Unification des mémoires réparties dans les systèmes hétérogènes

Erwan Lenormand

université PARIS-SACLAY

Unification des mémoires réparties dans les systèmes hétérogènes

Unifying Distributed Memories in Heterogeneous Systems

Thèse de doctorat de l'université Paris-Saclay

École doctorale n° 580, sciences et technologies de l'information et de la communication (STIC)
Spécialité de doctorat : Informatique
Graduate School : Informatique et science du numérique,
Référent : Faculté des sciences d'Orsay

Thèse préparée dans l'unité de recherche **Institut LIST** (Université Paris-Saclay, CEA), sous la direction d'**Henri-Pierre CHARLES**, Directeur de recherche, le co-encadrement de **Loïc CUDENNEC**, Ingénieur de recherche, et le co-encadrement de **Stéphane LOUISE**, Directeur de recherche

Thèse soutenue à Paris-Saclay, le 31 janvier 2022, par

Erwan LENORMAND

Composition du jury

Marc PÉRACHE Directeur de recherche, CEA DAM, Université Paris-Saclay	Président
Tanguy RISSET Professeur, Laboratoire CITI, INSA-Lyon	Rapporteur & Examinateur
Samuel THIBAULT Professeur, INRIA, Université de Bordeaux	Rapporteur & Examinateur
Catherine DEZAN Maître de conférences, Lab-STICC, Université de Bretagne Occidentale	Examinatrice
Olivier MULLER Maître de conférences, Laboratoire TIMA, Université Grenoble Alpes	Examinateur
Soraya ZERTAL Maître de conférences, LI-PARAD, Université Paris-Saclay	Examinatrice
Henri-Pierre CHARLES Directeur de recherche, CEA LIST, Université Grenoble Alpes	Directeur de thèse

THESE DE DOCTORAT
NNT : 2022UPASG007

université PARIS-SACLAY

ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre : Unification des mémoires réparties dans les systèmes hétérogènes
Mots clés : Systèmes hétérogènes, Mémoire Virtuellement Partagée, Systèmes répartis

Résumé : Les ordinateurs du futur, embarqués dans une voiture ou dans des super calculateurs, auront une hiérarchie mémoire très complexe. Il s'agit de relier des dizaines (centaines) de cœurs de calcul à des tera-octets de mémoire. Les cœurs de calculs peuvent être hétérogènes (CPU, GPU, DSP, FPGA), la mémoire également (DRAM, NVRAM, FLASH). Beaucoup d'architectures existent (mémoire distribuées, mémoires partagées, NUMA), plusieurs possibilités existent pour leur exploitation matérielle (cohérence de cache, protocoles de communication) ou logicielles (parallélisme de thread, OpenMP, transactions), plusieurs options existent pour la communication entre cœur et mémoire (MPI, RDMA, RoCE, CCIX, GenZ). Aucune approche ne donne ni de modèle de programmation clair, ni de modèle de mémoire simple dans le cadre d'applications parallèles. L'objectif de cette thèse de doctorat est d'étudier la possibilité d'exploiter les technologies émergentes autour des ressources d'accélération de calcul, des mémoires hybrides (persistantes ou non) et des standards de communication distants (par exemple le RDMA) afin de rendre plus performant le partage de données sur les plateformes hétérogènes et de fournir un modèle simple pour le programmeur.

Title : Unifying Distributed Memories in Heterogeneous Systems
Keywords : Heterogeneous systems, Distributed Shared Memory, Distributed system

Abstract : Future computers in high-performance and embedded systems lead to complex memory hierarchies. Hundreds of computing nodes will have to be connected to tera-bytes of memories. In such systems, both the processing units (CPU, GPU, DSP, FPGA) and the memories (DRAM, NVRAM, FLASH) can be heterogeneous. Several architectures exist (distributed memory, shared memory, NUMA) with different hardware implementations (cache coherence, communication protocols), software implementations (thread parallelism, OpenMP, transactions) and communication technologies between processing units and memory (MPI, RDMA, RoCE, CCIX, GenZ). None of the approaches above offer a simple, unified programming model and memory model for parallel applications. The purpose of this Ph.D. Thesis is to study the possibility of using emerging technologies related to computing units, hybrid memories (persistent or not) and remote communication standards in order to accelerate data sharing onto heterogeneous platforms and provide a convenient programming model.

Remerciements

Mon travail de thèse s'est déroulé en grande partie pendant une période de pandémie. La qualité de l'encadrement académique et des relations humaines y prenaient une importance particulière et ont été un soutien indispensable de mon travail.

Pour cela, je tiens à remercier mon directeur de thèse, Henri-Pierre Charles, pour son attention constante et bienveillante. Henri-Pierre a su me faire confiance tout en me guidant avec justesse. Également, je remercie Henri-Pierre pour tout le soutien qu'il m'a apporté pendant plus de trois ans.

Mes remerciements vont également à mes encadrants, Loïc Cudennec qui m'a accompagné dès l'origine, à Thierry Goubier et Stéphane Louise dont les conseils m'ont été précieux. Leur participation à ce projet de recherche en parfaite cohésion avec mon directeur de thèse m'a permis de mener mes travaux de recherche sereinement.

J'exprime ma gratitude envers mes rapporteurs, Tanguy Risset et Samuel Thibault, dont les remarques m'ont permis d'éclaircir et de compléter certains points fondamentaux.

Je remercie chaleureusement Catherine Dezan, Olivier Muller, Marc Pérache et Soraya Zertal d'avoir accepté d'examiner mes travaux lors de la soutenance de thèse. Je remercie une seconde fois Olivier Muller pour avoir également participé au comité de suivi de thèse à mi-parcours et pour avoir organisé un séminaire durant lequel j'ai pu présenter mes travaux. Ses conseils m'ont été d'une aide précieuse.

J'exprime ma reconnaissance au département DSCIN de l'institut LIST et ses collaborateurs de m'avoir accueilli pour mon stage de master, puis pendant ma thèse de doctorat.

Je tiens enfin à marquer toute ma gratitude envers ma compagne, ma famille et mes amis qui ont su me faire surmonter mes doutes et ont cru en moi.

Résumé

Suite à la perte de la mise à l'échelle de Dennard, les systèmes de calcul deviennent de plus en plus hétérogènes, notamment par l'ajout de coprocesseurs. Grâce à leur architecture optimisée pour traiter massivement les données, les coprocesseurs permettent à la fois d'accélérer le traitement des applications et d'en augmenter l'efficacité énergétique. Cependant, cette amélioration des performances par l'hétérogénéité se paie au prix d'un modèle de programmation hybride et distribué rendant l'utilisation des systèmes plus complexe. Cette thèse souhaite apporter une solution au problème de la gestion des données dans les systèmes hétérogènes. Beaucoup d'approches ont été proposées pour répondre à cette problématique. Si elles présentent toutes des caractéristiques différentes, la majorité repose sur un partitionnement à gros grains des structures de données en blocs réguliers. Les blocs sont transférés avant le lancement des noyaux de calcul pour satisfaire les dépendances de données. Ces approches permettent de gérer efficacement des structures de données régulières dont le traitement génère des motifs d'accès réguliers et prévisibles. Cependant les applications de calcul utilisent de plus en plus des structures de données irrégulières dont le traitement provoquent des accès aléatoires et imprévisibles. Tel que l'illustre le banc de test *High Performance Conjugate Gradient*, les performances de ces applications sont limitées par celles des systèmes mémoires et d'interconnexions. Ainsi, l'optimisation de ces applications nécessitent de limiter les échanges de données entre les différentes mémoires. Une mémoire partagée offre un paradigme pratique pour la programmation de ce genre d'applications en résolvant les problématiques de la localisation, de l'orchestration des transferts et du maintient de la cohérence des données. Grâce à leur architecture programmable, les accélérateurs reconfigurables semblent être les ressources de calcul les plus adaptées pour traiter les applications irrégulières. Ainsi, dans le cadre des travaux de cette thèse, nous avons proposé d'étudier

l'intégration d'accélérateurs reconfigurables dans une Mémoire Virtuellement Partagée (MVP). Ce positionnement se distingue de l'état de l'art par la capacité donnée aux accélérateurs d'initier les accès aux données distantes. Pour ce faire, nous avons proposé un modèle d'intégration des accélérateurs dans une MVP logicielle déjà existante. Cette intégration repose sur un système de proxy permettant de faire communiquer les noyaux de calcul, implémentés sur la logique programmable de l'accélérateur, avec l'environnement logiciel de la MVP. Nous avons proposé un modèle de programmation pour ce système qui répond aux besoins des applications dont les motifs d'accès aux données sont aléatoires. Pour résoudre le problème de la latence d'accès aux données distante, le modèle de programmation utilise la superposition temporelle des flux de données transférées avec les flux de données traitées. Pour ce faire, il s'appuie sur un partitionnement des structures de données irrégulières en blocs de tailles adaptables permettant de masquer l'irrégularité des structures et de précharger les données sous forme de flux réguliers. Nous avons développé un outil de modélisation et de simulation permettant de valider le système et le modèle de programmation proposés. Deux cas d'études, la multiplication de matrices creuses et un code de simulation de tsunami, ont été utilisés pour mener les expérimentations. Ils ont permis d'évaluer le modèle de programmation pour deux structures de données irrégulières largement utilisées : les matrices creuses compressées et les maillages non structurés. Les résultats obtenus ont montré que le modèle de programmation permet de cacher efficacement les latences d'accès aux données et de quasiment atteindre les performances maximales permises par la bande passante du système mémoire du FPGA modélisé.

Abstract

Following the loss of Dennard scaling, computing systems have become increasingly heterogeneous by the integration of coprocessors. Thanks to their massively parallel architecture, coprocessors can both speed up processing and increase the energy efficiency of applications. However, this performance improvement through heterogeneity comes at the cost of a hybrid and distributed programming model that makes systems more complex to program. This thesis aims to address the data management problem in heterogeneous systems. Many works have been proposed to address this issue. While they all have different features, most rely on coarse-grained partitioning of data structures into regular blocks. Blocks are transferred before the compute kernels are executed to resolve data dependencies. These works make it possible to efficiently manage regular data structures whose processing generates regular and predictable access patterns. However, scientific applications increasingly use irregular data structures resulting in random and unpredictable data accesses. As illustrated by the High Performance Conjugate Gradient benchmark, the performance of these applications is limited by those of memory and interconnect systems. Thus, the optimization of these applications requires limiting the data exchange between the different memory systems. Shared memory is a convenient paradigm to program such applications by solving the problems of data localization, data transfers orchestration and data coherency. Thanks to their reconfigurable architecture, Field-programmable gate arrays (FPGAs) are particularly suitable for processing irregular applications. Thus, as part of the work of this thesis, we have proposed to study the integration of FPGAs in a Distributed Shared Memory (DSM). This work positioning differs from the state of the art by the ability given to accelerators to initiate access to remote data. To do this, we have proposed a integration model of FPGAs into Software-Distributed Shared Memory (S-DSM). This is based on a proxy

system enabling the compute kernels, implementing on the programmable logic of the accelerator, to communicate with the software environment of the S-DSM. We have proposed a programming model for this system that meets the requirements of applications whose randomly access data. To solve the problem of remote data access latency, the programming model relies on a temporal overlay of the transferred data flow with the processed data flow. To do this, it relies on irregular data structures partitioning into blocks of adaptable sizes in order to hide structures irregularities and to be able to prefetch data. We have developed a modeling and simulation tool to validate the proposed system and programming model. General Sparse Matrix-Matrix Multiplication (SpGEMM) and a tsunami simulation code were used as case studies to conduct experiments. They allowed to assess the programming model for two widely used irregular data structures : compressed sparse matrix and unstructured mesh. The obtained results showed that the programming model enables to effectively hide data access latencies and to almost reach the maximum performance allowed by the FPGA local memory bandwidth.

Table des matières

Introduction générale

La loi de Moore [79] prédisant une croissance exponentielle du nombre de transistors intégrés par circuit a été un moteur pour le développement de l'informatique. Cette prédiction s'est vérifiée notamment grâce à la miniaturisation des transistors et la mise à l'échelle de Dennard [39] qui expliquent comment la densité d'intégration des transistors et leur fréquence de fonctionnement ont pu augmenter tout en limitant l'enveloppe thermique (*Thermal Design Power* (TDP)) des circuits. Les limites de la physique ont entrainé une perte de la mise à l'échelle de Dennard [45] qui a eu pour conséquence la stabilisation de la fréquence d'horloge des processeurs et l'augmentation du nombre d'unités de traitement par processeur (cœurs de processeur). Dans le même temps, les applications informatiques sont devenues de plus en plus complexes, conduisant au développement de systèmes informatiques plus grands et de ce fait plus énergivores. L'hétérogénéité apporte une réponse à ce problème de consommation énergétique [42] par l'ajout de coprocesseurs dont les architectures sont optimisées pour traiter des jeux de données larges. Ainsi, les coprocesseurs permettent à la fois d'accélérer le traitement et d'augmenter l'efficacité énergétique d'une application intense en calcul.

Cependant, cette amélioration des performances par l'hétérogénéité se paie au prix d'une complexité de programmation plus importante. L'utilisation de ressources de traitement hétérogènes dans un même système conduit à un modèle de programmation hybride et distribué. Les applications doivent être analysées et segmentées pour extraire les portions dont l'intensité arithmétique est élevée, appelées noyaux de calcul, et dont l'exécution par un coprocesseur serait bénéfique. Chaque type de ressource de traitement est associé à un modèle de programmation, voire à un langage de programmation, spécifique à son architecture. L'exécution d'un noyau de calcul par un coprocesseur nécessite donc une transformation du code qui est une première source de difficulté

dans l'utilisation des systèmes hétérogènes. Une seconde source de difficulté est la distribution du traitement et des données entre les différentes ressources qui amène de nombreux problèmes, tels que l'équilibrage de la charge de travail, la synchronisation des tâches, l'orchestration des transferts de données ou le maintient de la cohérence des données. Pour des applications dont les noyaux de calcul font des accès aux données avec des motifs irréguliers, appelés noyaux de calcul irréguliers, l'utilisation d'un système hétérogène peut se révéler périlleuse.

Ce type de noyaux de calcul provoque beaucoup d'accès mémoire par rapport à leur nombre d'opérations ce qui stresse les systèmes mémoires et les systèmes d'interconnexion. Les coprocesseurs sont connectés au système par des bus de communications dont les bandes passantes peuvent former des goulots d'étranglement. Les mémoires embarquées des coprocesseurs sont limitées en capacité, ce qui peut nécessiter l'éviction de certaines données au fil de l'exécution et par la suite provoquer des transferts de données redondants.

Lorsque l'exécution est répartie entre plusieurs ressources de calcul, le partage des données est nécessaire. Les dépendances de données d'un noyau de calcul irrégulier peuvent être difficiles à identifier de manière fine au cours de l'exécution. Pour être certain de satisfaire ces dépendances, la stratégie de partage des données peut être de communiquer des blocs de données plus larges que nécessaire. Cette stratégie conduit à une mauvaise utilisation de la bande passante et augmente le risque de formation de goulot d'étranglement. Il est donc essentiel d'optimiser la gestion des données pour exploiter au mieux les performances des systèmes.

Le portage de noyaux de calcul irréguliers sur l'architecture des coprocesseurs peut aussi être complexe. Chaque type d'architecture offre un profil de performances différent (principalement caractérisé par la vitesse de calcul et la consommation énergétique). Par exemple, des processeurs graphiques (GPU) sont optimisés pour opérer sur des vecteurs ou des matrices denses. Le traitement de noyaux de calcul irréguliers par un GPU résulte en une baisse d'efficacité. Les accélérateurs reconfigurables (*Field Programmable Gate Arrays (FPGA)*) grâce à leur architecture programmable peuvent s'adapter et offrir un meilleur profil de performances pour les noyaux de calcul irréguliers [44].

L'objectif de cette thèse est d'étudier l'utilisation d'une mémoire virtuellement partagée (MVP) logicielle pour accélérer des noyaux de calcul irréguliers avec des FPGA. Une MVP logicielle agit comme un cache de données

et permet d'apporter de l'intelligence dans la gestion des données de manière transparente pour l'utilisateur. Les travaux de la thèse ont aboutit à trois contributions :

1. Un modèle d'intégration d'accélérateurs reconfigurables dans une MVP logicielle. Il permet à des noyaux de calcul instanciés sur l'accélérateur d'initier l'accès à l'ensemble des données allouées dans l'espace d'adressage de la MVP.
2. Un modèle de programmation pour noyaux de calcul irréguliers sur accélérateur reconfigurable dans un système distribué hétérogène. Il utilise la première contribution pour orchestrer le partage des données.
3. Le développement d'un outil de modélisation et de simulation utilisé pour valider le modèle de programmation. Celui-ci a permis de mener des expérimentations sur deux cas d'études : la multiplication de matrices creuses et un code de simulation de tsunami.

Ce manuscrit est organisé en trois parties :

— la partie I présente le contexte, les motivations et la problématique de la thèse.
— la partie II expose les contributions de la thèse.
— la partie III décrit les expérimentations menées pour valider le modèle de programmation et les résultats obtenus.

Première partie

Contexte, motivations et problématique

Chapitre 1

L'hétérogénéité dans les systèmes de calcul

1.1 Introduction

Les lois empiriques de Moore ont permis le développement des systèmes d'information qui occupent une place croissante dans notre société depuis l'invention du premier circuit intégré, en 1958, par Jack Kilby [66]. En 1965, Gordon E. Moore partageait [78] sa vision du futur de l'industrie de la microélectronique. À une époque où les systèmes de calcul sont pratiquement réservés à des usages militaires, ses propos se sont avérés particulièrement justes. En effet, il prédisait que les circuits intégrés permettraient l'avènement de l'ordinateur personnel, des voitures autonomes ou encore du téléphone portable. Mais, cet article est principalement connu comme étant l'établissement de la première loi de Moore. À ce moment, il prédisait que le nombre de transistors intégrés par circuit doublerait tous les ans et que le coût de fabrication par composant allait décroître pendant 10 ans. En 1974, Moore est invité à se prononcer à nouveau sur sa vision de l'évolution de l'industrie de la microélectronique [79]. Il corrige alors sa prédiction, en annonçant un doublement du nombre de transistors intégrés par circuit tous les deux ans. Ce postulat se révèlera particulièrement proche de la réalité et restera comme la loi de Moore. En effet, nous pouvons observer sur la figure 1.1 que cette loi (symbolisée par la ligne bleue) suit de près le nombre de transistors intégrés par microprocesseurs haut de gamme (symbolisés par les points bleus) produits au fil des années. Cette corrélation a été rendue possible par la miniaturisation

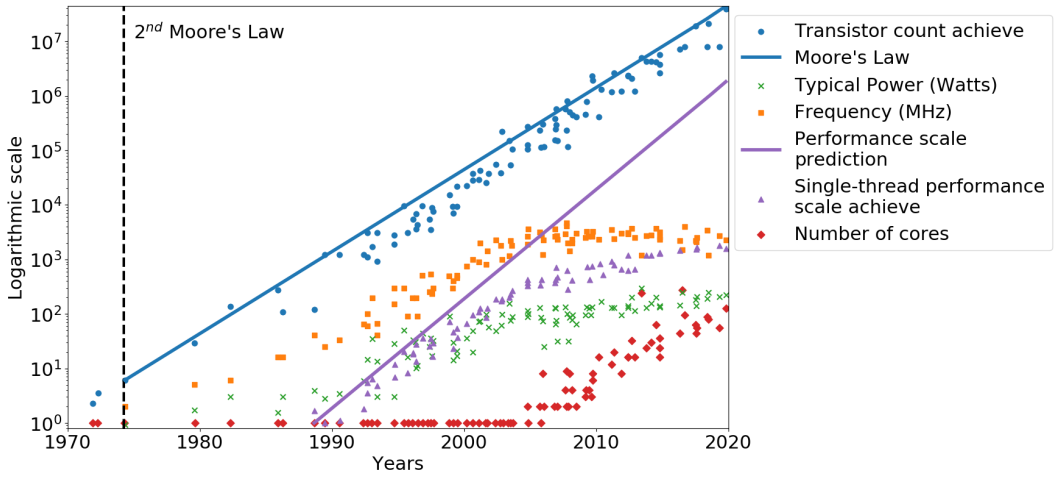


FIGURE 1.1 – 48 ans de données sur les tendances des microprocesseurs. Les données ont été collectées jusqu'en 2010 par M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, et C. Batten, puis entre 2010 et 2019 par K. Rupp.

des transistors qui a permis d'augmenter leur densité par surface et grâce à l'augmentation de la taille des puces. Ainsi, lors de l'énoncé de la deuxième loi de Moore, la surface de la puce du microprocesseur *Intel 8080* était de 20 mm^2 dont les transistors étaient gravés avec une finesse de 6 µm. Aujourd'hui, la finesse de gravure pour les processeurs haut de gamme est inférieure à 10 nm et la surface des puces peut être de l'ordre du centimètre carré grâce à l'empilement de circuits (*i.e.* les circuits intégrés en trois dimensions).

La mise à l'échelle de Dennard est une observation partagée par Rober H. Dennard *et al.* [39] en 1974. Celle-ci conclut que grâce à la miniaturisation des transistors, il est possible d'augmenter leur densité d'intégration et la fréquence de fonctionnement des circuits sans accroître la consommation énergétique. Cette observation s'appuie sur le fait que la réduction de la taille des transistors permet de baisser leur tension d'alimentation et de raccourcir leur délai de commutation. David House, un employé de l'entreprise *Intel*, utilisa la loi de Moore et la mise à l'échelle de Dennard pour annoncer que la puissance de calcul des microprocesseurs doublerait tous les 18 mois. Cette prédiction (souvent associée à tort à la loi de Moore) se révéla juste pendant plusieurs années. Sur la figure 1.1, les triangles violets représentent la puissance de traitement des microprocesseurs à partir de 1988. Celle-ci est mesurée pour un

seul fil d'exécution avec le banc de test *SPECint*[1]. Les valeurs représentées ont été normalisées par le score obtenu par le microprocesseur le moins performant. Sur cette même figure, la droite violette représente, à partir de 1988, une croissance exponentielle de la puissance de calcul des microprocesseurs telle qu'exprimée par David House. Nous pouvons observer que la courbe et les triangles suivent la même croissance pendant une dizaine d'années, ensuite, la puissance des *threads* tend à stagner.

La perte de la mise à l'échelle de Dennard survenue dans les années 2000, explique le ralentissement de l'accroissement de la puissance de traitement des processeurs. Avant cette date, l'augmentation du nombre de transistors dans les puces permettait de concevoir des processeurs plus complexes et donc plus efficaces avec une seule unité de traitement, notamment grâce à des évolutions architecturales telles que : le pipeline d'instructions [4], l'exécution dans le désordre des opérations flottantes avec l'algorithme de Tomassulo [106], l'intégration d'une mémoire cache dans le processeur [54] ou encore la prédiction des branchements [97]. Cependant, l'augmentation des performances des processeurs était principalement soutenue par l'accélération de leur cadence d'horloge permise par la mise à l'échelle de Dennard. En atteignant des limites physiques, celle-ci a fortement ralenti. Les concepteurs de circuits intégrés ont du faire face à ce problème. En continuant d'augmenter la fréquence d'horloge des processeurs, la consommation énergétique allait croître. L'énergie consommée par un circuit est dissipée sous forme de chaleur. En dépassant un certain seuil, celle-ci peut endommager le circuit. Comme nous pouvons l'observer sur la figure 1.1, la fréquence d'horloge des processeurs (symbolisée par des carrés oranges) a été "gelée" à partir du milieu des années 2000. Pour continuer d'exploiter la loi de Moore, une révolution architecturale a été adoptée : les processeurs multi-cœurs [52] (*i.e.* des processeurs qui intègrent plusieurs unités de traitement dans un même circuit). Ainsi, en 2001, *IBM* commercialisait le premier processeur multi-cœurs : le *IBM POWER 4*. Nous pouvons observer sur la figure 1.1 que le nombre de cœurs de processeurs (symbolisé par les losanges rouges) ne cesse de croître au fil des années.

$$S(n) = \frac{1}{1 - F_a + \frac{F_a}{n}} \tag{1.1}$$

1. `https://en.wikipedia.org/wiki/SPECint`

La loi d'Amdahl décrite par l'équation 1.1 permet de calculer l'accélération théorique en latence de l'exécution parallèle d'un programme par rapport à une exécution séquentielle. Dans cette équation, S donne l'accélération théorique, n indique le nombre de cœurs du processeur et F_a correspond au pourcentage du temps d'exécution séquentiel qui est parallélisé. Ainsi, $1 - F_a$ correspond à la partie d'un code qui ne peut pas être exécutée en parallèle. La loi d'Amdahl traduit le fait que l'accélération d'un programme par son exécution parallèle est limitée par la portion séquentielle de son code. Il en résulte que les architectures multi-cœurs conduisent à une baisse d'efficacité.

Les systèmes hétérogènes sont des systèmes de calcul qui intègrent plusieurs composants ayant la même fonction, mais dont les architectures varient. Il existe plusieurs sources d'hétérogénéité dans un système de calcul, par exemple :

- pour un système de stockage de données avec l'association d'un disque dur (*Hard Disk Drive* (HDD)) et d'un disque statique à semi-conducteur (SSD).
- pour un système mémoire avec l'association d'une mémoire vive volatile (*Random Access Memory* (RAM)) et d'une mémoire vive non volatile (*Non-volatile RAM* (NVRAM))
- un processeur multi-cœurs dont les cœurs utilisent des jeux d'instructions différents.
- un système de calcul par l'association de processeurs et de coprocesseurs.

Les systèmes hétérogènes étudiés dans le cadre de cette thèse s'inscrivent dans cette dernière catégorie. Cette forme d'hétérogénéité dans un système de calcul permet d'apporter une réponse au problème de la consommation énergétique [42]. Ce problème est lié, d'une part, à la perte d'efficacité des processeurs multi-cœurs et, d'autre part, à l'augmentation de leur consommation énergétique. Tel que nous pouvons l'observer sur la figure 1.1, malgré le gel de leur fréquence d'horloge, les microprocesseurs ont vu leur enveloppe thermique (TDP) légèrement croître (symbolisée par des croix vertes). Pour contenir cette chaleur, des techniques de refroidissement actives sont utilisées. Celles-ci participent à l'augmentation de la consommation énergétique globale d'un système de calcul. Dans le même temps, les besoins applicatifs ont augmenté eux aussi. Pour satisfaire ces besoins, les systèmes de calcul deviennent de plus en plus

grands et donc plus énergivores. Les enjeux climatiques actuels nous poussent à concevoir des systèmes de calcul économes en énergie. Ainsi, aujourd'hui, l'efficacité énergétique est devenue une métrique aussi importante que la vitesse de traitement pour évaluer les performances d'un système de calcul. Un coprocesseur est une ressource de calcul dont l'architecture est optimisée pour réaliser certains types de traitement. Ils sont intégrés dans les systèmes de calcul pour agir comme des accélérateurs matériels. Leur usage typique consiste à déporter les portions d'une application dont l'intensité arithmétique est élevée.

Le reste de ce chapitre s'organise de la façon suivante : la section 1.2 présente différentes architectures de processeurs et de coprocesseurs utilisées pour le calcul générique et la section 1.3 conclut ce chapitre et présente la problématique du manuscrit.

1.2 Architectures de processeurs et coprocesseurs pour le calcul générique

L'architecture d'un ordinateur tel qu'on le connait est basée sur l'architecture de von Neumann (nommée en référence au mathématicien John von Neumann) illustrée sur la figure 1.2. Celle-ci est organisée autour d'un processeur (*Central Processing Unit* (CPU)) qui contrôle l'ordinateur. Les données et instructions des programmes sont stockées dans une mémoire principale. L'utilisateur interagit avec l'ordinateur par des périphériques d'entrée (*Input*) et de sortie (*Output*). Pour exécuter un programme, le CPU charge les instructions depuis la mémoire. Ces instructions décrivent le traitement qui doit être appliqué aux données du programme. Le CPU accède à ces données et les stocke temporairement dans ses registres internes, puis réalise des opérations dessus avec son unité arithmétique et logique (*Arithmetic Logic Unit* (ALU)). Les périphériques sont également utilisés pour augmenter les fonctionnalités de l'ordinateur, par exemple : inclure un système de stockage permanent, interconnecter l'ordinateur à d'autre systèmes d'information ou ajouter des coprocesseurs.

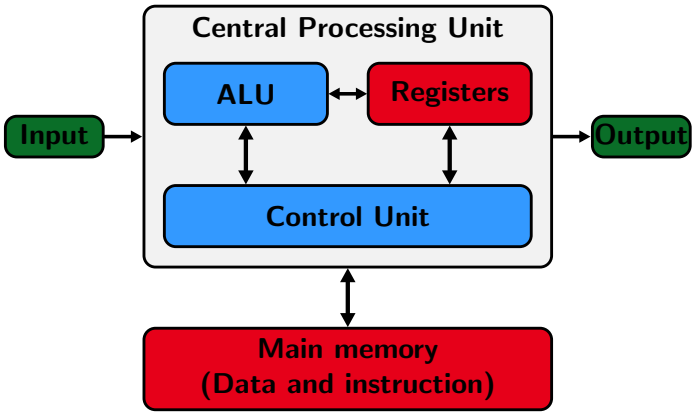


FIGURE 1.2 – Représentation de l'architecture de von NeuMann.

1.2.1 Architecture d'un processeur pour les serveurs de calcul

Depuis le début des années 2000, les processeurs deviennent de plus en plus parallèles en intégrant plusieurs cœurs de calcul. Aujourd'hui, les CPU conçus pour les serveurs de calcul intègrent plusieurs dizaines de cœurs. La figure 1.3 représente une architecture de ce type de processeur. Trois classes de composants se distinguent : les unités de calcul (en bleu), la mémoire (en rouge) et les systèmes de communication (en vert). Les unités de calcul (cœurs du processeur) sont regroupées en plusieurs grappes (*clusters*) reliées par un réseau sur puce (*Network on Chip* (NOC)). La mémoire est organisée en plusieurs niveaux de caches. Chaque cœur possède un cache privé appartenant au premier niveau (*Cache L1*) qui est segmenté pour contenir les données et les instructions des programmes. Chaque grappe intègre un cache du deuxième niveau (*Cache L2*) qui est partagé entre ses cœurs. Un troisième niveau de cache (*Cache L3*) peut être utilisé pour unifier tous les caches *L2*. Le cache L3 peut également être partitionné et physiquement réparti. Dans ce cas, chaque partition est directement interfacée avec une ou plusieurs grappes. Cependant, toutes les grappes peuvent accéder à l'ensemble des partitions. Le réseau sur puce est utilisé pour transférer les données. Plusieurs topologies de NoC existent, par exemple : une grille, un anneau ou un *crossbar* permettant de faire une connexion point-à-point entre toutes les grappes. La répartition physique du cache L3 conduit à des accès à la mémoire non uniformes (*Non Uniform Memory Access* (NUMA)). Une grappe a un accès plus rapide aux données contenues dans la partition à laquelle elle est interfacée qu'à une partition

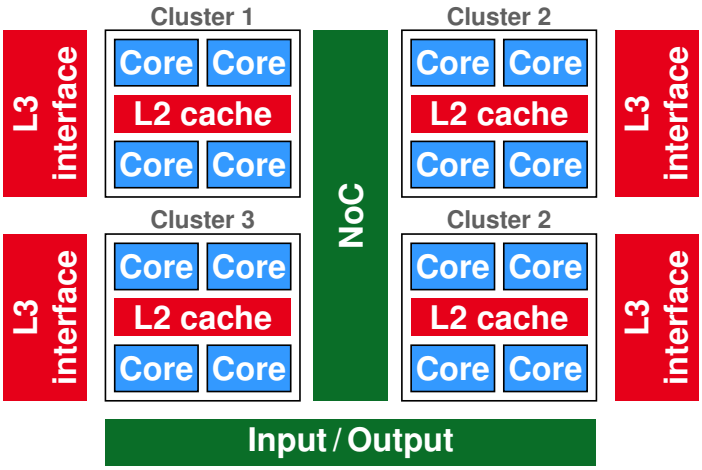


FIGURE 1.3 – Architecture d'un CPU pour les serveurs de calcul.

distante.

Aujourd'hui, nous pouvons observer que les processeurs haut de gamme tirent profit de l'intégration sur silicium en trois dimensions [2] pour externaliser le dernier niveau de cache sur d'autres circuits. L'intérêt est notamment d'offrir un plus grand volume d'espace mémoire cache et un plus grand débit de données.

L'architecture des cœurs a également évolué dans le temps. Ainsi, un cœur de processeur autorise le traitement de plusieurs fils d'exécution en parallèle et intègre des unités de calcul vectorielles. Ces dernières réalisent une opération sur un vecteur de données en une seule instruction (*Single Instruction, Multiple Data* (SIMD)). Ces évolutions augmentent la puissance de calcul des processeurs.

Les processeurs intègrent également des modules leur permettant de communiquer avec le reste du système. Pour répondre aux exigences du calcul haute performance (*High Performance Computing* (HPC)), ces interfaces de communication ont également évolué dans le temps. Aujourd'hui, elles permettent d'échanger des données avec un plus grand débit et une plus faible latence.

En 2021, le processeur qui affiche la puissance de calcul la plus élevée est le *A64FX* conçu par l'entreprise japonaise *Fujitsu*. Il s'agit d'une architecture *ARMv8.2-A*. Ce processeur est notamment utilisé par le superordinateur

2. L'intégration en 3D consiste à empiler plusieurs circuits et les interconnecter avec des vias traversants.

Fugaku qui est le plus puissant du TOP500 [3] en 2021. Il intègre 48 cœurs de processeur cadencés entre 1.8 GHz et 2.2 GHz, qui sont organisés en 4 grappes. Il est doté de 32 Mio de mémoire cache L2 (4× 8 Mio). Ses cœurs de processeurs permettent d'exécuter des instructions vectorielles sur 512 bits. Une des innovations de ce CPU est d'intégrer (par une intégration en 3 dimensions) 32 Gio de mémoire à haute bande passante (*High Bandwidth Memory* (HBM) [4]) offrant 1 Tio/s de bande passante mémoire. Ces innovations lui permettent d'atteindre en théorie [5] une vitesse de calcul de 3.4 TFLOP/s en double précision (64 bits). Son enveloppe thermique (TDP) est de seulement environ 165 W, ce qui en fait un processeur efficace énergétiquement. Malgré cela, le superordinateur *Fugaku* est seulement 26^{me} au classement des systèmes les plus efficaces énergétiquement du TOP500 en novembre 2021 [6]. Ce résultat s'explique par l'absence de coprocesseur. Un processeur est généraliste. Il est capable de réaliser tous types de traitement nécessaire au fonctionnement de l'ordinateur. Tandis que, un coprocesseur est conçu et optimisé pour réaliser des opérations de calcul. Ainsi, avec le même budget énergétique un coprocesseur peut traiter un plus grand nombre d'opérations de calcul qu'un processeur.

1.2.2 Architecture d'un processeur *manycore*

Un processeur *manycore*, également appelé processeur massivement parallèle, est un type spécifique de processeur multi-cœur. Il se distingue par l'intégration d'un très grand nombre de cœurs de processeur. La figure 1.4 représente ce modèle d'architecture, basée sur celle d'un processeur *MPPA-256* de l'entreprise française *Kalray*. Les cœurs de processeur sont regroupés dans des grappes reliées par un réseau sur puce utilisant une topologie en grille. Chaque grappe intègre un cache L2. Pour certains processeurs la cohérence des données entre les grappes peut être orchestrée par les caches L2 et le réseau sur puce. C'est la cas par exemple de l'architecture TSAR (*Tera-Scale ARchitecture*) [7]. Dans ce cas, chaque grappe constitue un domaine NUMA. D'autres architectures, comme celle du *MPPA-256*, ne supportent pas la co-

3. Classement des 500 ordinateurs les plus puissants du monde : `https://www.top500.org/`

4. Les mémoires HBM utilise l'empilement de circuit pour offrir un grand espace de stockage et une large bande passante.

5. Données fournies par *Fujitsu*

6. `https://www.top500.org/lists/green500/list/2021/11/`

7. `https://www-soc.lip6.fr/trac/tsar`

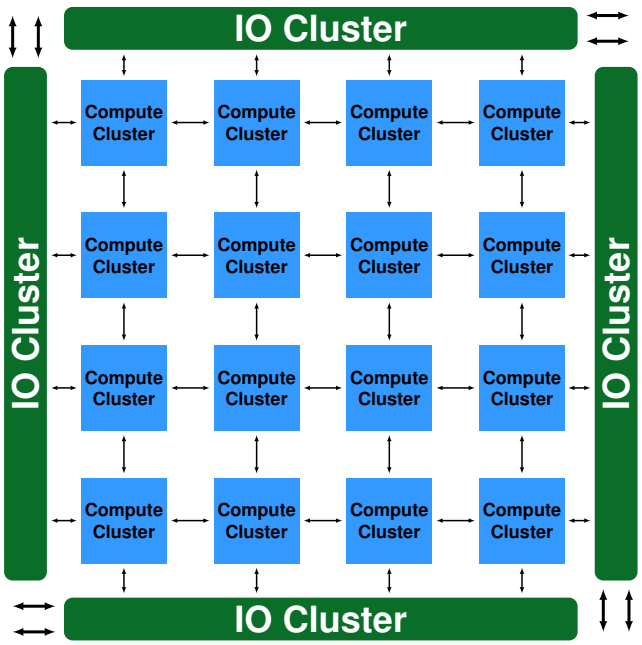


FIGURE 1.4 – Architecture d'un processeur *manycore* utilisant une topologie en grille.

hérence entre les grappes. Le processeur *MPPA-256* intègre 16 grappes de 16 cœurs, portant le nombre total de cœurs à 256.

Les processeurs *manycore* peuvent être utilisés en tant que coprocesseurs dans les serveurs de calcul ou en tant qu'unité centrale, notamment pour les systèmes embarqués. Pour le premier cas, le nombre important d'unités de calcul permet d'accélérer les tâches de calcul d'une application. Pour le second cas, leur architecture permet d'exécuter en parallèle plusieurs applications intensives en calcul.

1.2.3 Architecture d'un processeur graphique

Les images étant représentées numériquement sous la forme de matrices, leur traitement exhibe naturellement un large parallélisme. Les processeurs graphiques (*Graphics Processing Unit* (GPU)) sont des ressources de calcul qui ont été conçues pour accélérer ce traitement. Puisque beaucoup de systèmes physiques peuvent être modélisés sous la forme de matrices ou de vecteurs, aujourd'hui les GPU sont utilisés comme accélérateurs pour réaliser du calcul générique (*General-purpose computing on Graphics Processing Unit*

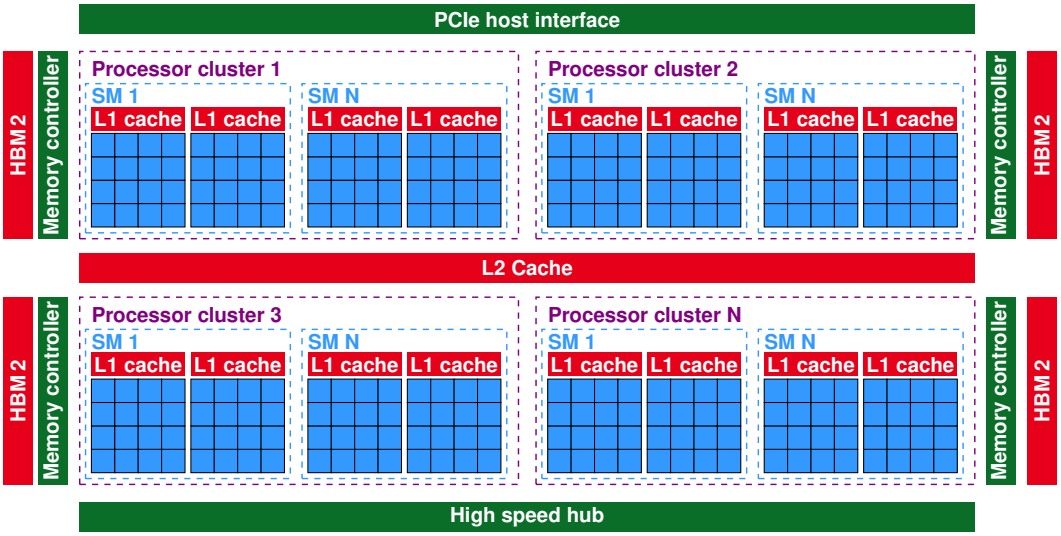


FIGURE 1.5 – Architecture d'un processeur graphique.

(GPGPU)). Pour répondre à ce nouveau besoin, des architectures de processeurs graphiques sont spécifiquement conçues pour le calcul haute performance (HPC). Ces processeurs sont intégrés sur des cartes qui elles aussi sont adaptées aux besoins du calcul HPC. Le marché des cartes graphiques pour le calcul HPC est très largement dominé par l'entreprise *Nvidia*.

La figure 1.5 représente l'architecture d'un processeur graphique moderne. Elle se compose de plusieurs grappes de calcul (*Processor cluster*) qui chacune intègre plusieurs *Streaming Multiprocessor* (SM). Les SM exécutent des *threads* et intègrent un cache de données et d'instructions. Ils se composent d'un grand nombre :

— d'unités de calcul flottant en simple précision (32 bits) et en double précision (64 bits) ;
— d'unités de calcul entier ;
— d'unités de calcul tensoriel ;
— d'unités de lecture et d'écriture mémoire.

Contrairement à un CPU où l'activité de chaque cœur est indépendante, le contrôle de l'ensemble des unités d'un SM est centralisé. Ainsi, chaque SM intègre un ordonnanceur de *threads* qui est en charge de répartir les instructions à exécuter entre les différentes unités.

Un cache L2 unifie les mémoires des grappes de calcul. Aujourd'hui, les GPU intègrent des mémoires HBM dont les interfaces sont réparties entre les différentes grappes. Des interfaces de communication sont également intégrées

pour faire des échanges de données externes à la carte. Ainsi, une interface *PCI Express* est disponible pour relier le processeur à un système de calcul en tant que périphérique. Des interfaces permettent également de relier plusieurs cartes graphiques, via un hub de communication rapide, pour former des grappes de GPU. Pour les cartes *Nvidia*, ces interfaces sont nommées *NVlink*.

Ampere 100 est la dernière gamme de cartes graphiques *Nvidia* et la plus puissante commercialisée en 2021. Les cartes graphiques *GA 100* en font partie. Leur processeur se compose de 8 grappes de calcul cumulant au total 8192 unités de calcul flottant en simple précision. Ces cartes intègrent des mémoires HBM offrant plus de 1.5 To/s de bande passante. Leur puissance de calcul maximale théorique sur des nombres flottants 64 bits est de 9.7 TFLOP/s. L'enveloppe thermique (TDP) d'une de ces cartes est de 400 W.

1.2.4 Architecture d'un accélérateur reconfigurable

Un *Field-Programmable Gate Array* (FPGA) est un circuit logique programmable. Leur architecture permet d'implémenter des accélérateurs matériels pouvant être reconfigurés. Grâce à cette caractéristique, les FPGA offrent beaucoup de libertés pour concevoir des accélérateurs répondant aux besoins spécifiques d'une application. Ces circuits sont intégrés sur des cartes dotées de nombreux périphériques leur permettant de communiquer.

La figure 1.6 illustre l'architecture d'un FPGA. Elle se compose d'un réseau de composants :

- des blocs de logique configurable (*Configurable Logic Block* (CLB)). Ils intègrent des tables de correspondance, des multiplexeurs et des bascules. Les tables de correspondance sont programmées pour réaliser une fonction logique.
- de la mémoire embarquée (*Block RAM* (BRAM)). Elle est distribuée en plusieurs blocs et permet de stocker des données volatiles. Typiquement, le volume de stockage d'un bloc est de 18 kbits ou 36 kbits.
- des *Digital signal processor* (DSP) qui permettent d'effectuer des opérations arithmétiques.
- des blocs d'entrée et sortie (*IO*) qui permettent d'interagir avec les périphériques de la carte FPGA.

Le réseau est configurable. Il permet de connecter différents composants pour créer des modules plus complexes. Plusieurs horloges peuvent être utilisées

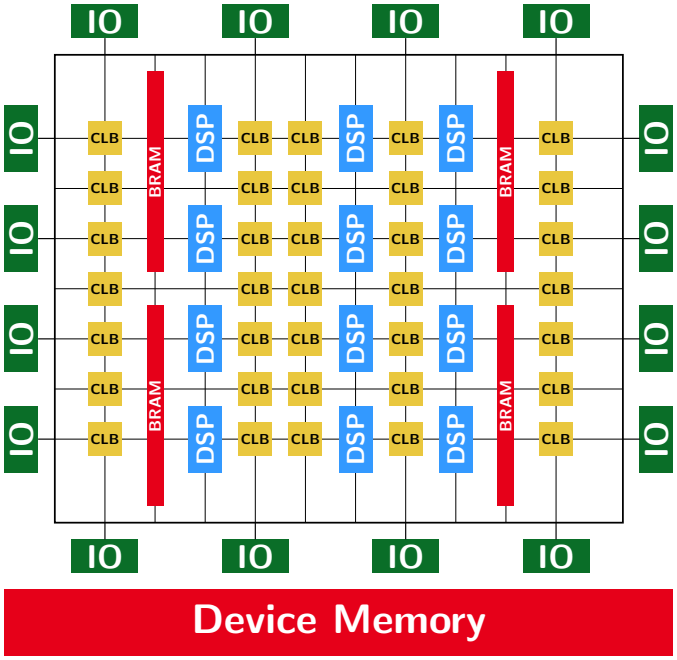


FIGURE 1.6 – Représentation de l'architecture d'un FPGA.

pour cadencer le fonctionnement des modules. Un FPGA est configuré avec un fichier *bitstream* qui permet notamment de remplir les tables de correspondance des CLB et de décrire la topologie du réseau. Aujourd'hui, comme pour les GPU haut de gamme, les cartes FPGA intègrent des mémoires HBM.

1.3 Conclusion

L'emploi de coprocesseurs dans les systèmes de calcul est devenu très courant. Ceux-ci permettent d'augmenter la puissance de calcul d'un système et d'en améliorer l'efficacité énergétique. Cette popularité se retranscrit notamment dans les superordinateurs du TOP500, où l'on peut observer depuis le début des années 2010 une augmentation de la proportion d'architectures intégrant des coprocesseurs [51], principalement des GPU et quelques processeurs *manycore*. Avec les enjeux climatiques, la consommation énergétique des systèmes de calcul est devenue une préoccupation majeure. Le Green500 [8] classe les ordinateurs du TOP500 en fonction de leur efficacité énergétique. Pour le classement de novembre 2021 [9], les 21 ordinateurs les plus efficaces intègrent

8. https://www.top500.org/lists/green500/
9. https://www.top500.org/lists/green500/2021/11/

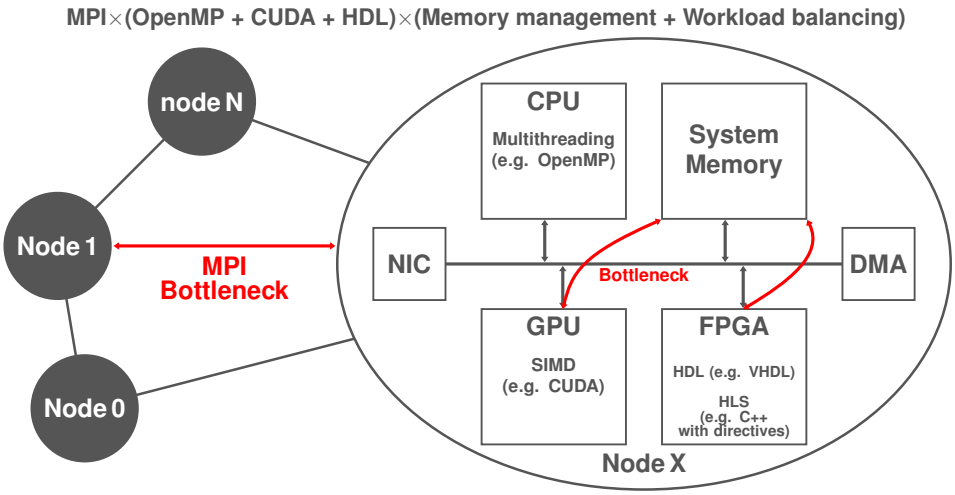


FIGURE 1.7 – Représentation d'un système de calcul distribué et hétérogène.

des coprocesseurs et seuls 4 systèmes parmi les 50 premiers ne sont pas hétérogènes.

1.3.1 Problématique de la thèse

L'hétérogénéité apporte une réponse au problème de la consommation énergétique des systèmes de calcul en augmentant leur efficacité. Cependant, ce gain se paye au prix d'une plus grande complexité de programmation. Ceci amène un nouveau problème illustré sur la figure 1.7.

Premièrement, l'architecture des ressources de calcul étant variée, chacune se programme différemment. Cela implique pour un développeur de maîtriser plusieurs modèles de programmation et parfois plusieurs langages de programmation. Chaque type de ressource de calcul peut traiter un même problème à une vitesse différente. Pour exploiter efficacement le système, le développeur doit distribuer la charge de travail de manière équitable au vu des performances de chaque ressource.

Deuxièmement, les coprocesseurs sont intégrés dans les systèmes par des bus de données (typiquement un bus *PCI Express*) et disposent d'une mémoire locale. L'utilisation d'un coprocesseur pour décharger une partie de la charge de travail d'une application nécessite de transférer les données entre la mémoire principale du nœud de calcul et les mémoires locales des coprocesseurs. Ce transfert implique une latence qui doit être prise en compte lors de l'équilibrage de la charge de travail. Le bus de données peut former un

goulot d'étranglement car il peut être partagé avec plusieurs périphériques et que sa bande passante est limitée. De ce fait, le transfert des données doit être optimisé. Une politique de gestion des données est adoptée pour limiter ces transferts. Généralement, elle consiste à conserver les données d'une application le plus longtemps possible dans les mémoires. Cependant, cette politique peut être difficile à mettre en œuvre, d'une part, à cause de l'hétérogénéité des ressources de calcul et, d'autre part, à cause des dépendances de données. Les mémoires de coprocesseurs ont une taille inférieure à celle de la mémoire principale d'un nœud de calcul. Par conséquent, elles ne sont pas toujours en capacité de stocker l'ensemble des données d'une application. L'asymétrie d'un système hétérogène peut s'observer également sur le dimensionnement des mémoires des coprocesseurs. Cette asymétrie oblige à adopter une politique de gestion des données non uniforme, plus complexe à mettre en œuvre, notamment de manière portable [10]. Lorsque l'exécution est répartie, les dépendances de données obligent leur transfert entre les ressources de calcul. Pour certains algorithmes les dépendances sont facilement prévisibles et impliquent des blocs de données réguliers, rendant l'orchestration des transferts des données relativement simple à gérer. Par opposition, des algorithmes peuvent avoir des dépendances de données difficilement identifiables et impliquer des blocs irréguliers. Dans ce cas, le transfert des données peut devenir complexe et inefficace car mal orchestré. Lorsque le système est distribué sous la forme de plusieurs nœuds de calcul, la politique de gestion des données devient encore plus complexe avec la multiplication des ressources de calcul et des mémoires physiquement distribuées. Un nouveau modèle de programmation doit être adopté pour faire communiquer les nœuds. Il implique généralement l'utilisation d'un intergiciel de passage de messages tel que MPI et conduit à un modèle de programmation hybride. De plus, le réseau utilisé pour faire communiquer les nœuds peut être la source d'un nouveau goulot d'étranglement.

La problématique de cette thèse est d'apporter une solution à la complexité de programmation des systèmes hétérogènes dont les mémoires sont physiquement réparties au travers du prisme de la gestion de données partagées et de l'importance d'amener les bonnes données au bon endroit et au bon moment.

Le reste de cette partie du manuscrit s'organise de la manière suivante : le

10. Pouvant s'adapter à différentes architectures de système.

chapitre 2 présente une classification des algorithmes utilisés dans les applications de calcul scientifique et fait une analyse de l'adéquation de l'architecture des coprocesseurs pour traiter ces algorithmes, le chapitre 3 expose des solutions de l'état de l'art pour organiser le partage des données entre les différentes mémoires d'un système distribué et hétérogène, enfin une conclusion est proposée en fin de partie.

Chapitre 2

Adéquation des architectures matérielles pour le traitement d'algorithmes de calcul scientifique

2.1 Introduction

Le chapitre 1 a exposé la tendance des systèmes de calcul à devenir de plus en plus hétérogènes, notamment par l'intégration de coprocesseurs dans les nœuds de calcul. L'objectif de cette évolution architecturale est notamment d'améliorer les performances d'un système qui sont principalement évaluées par deux critères :

- la vitesse de calcul qui peut être exprimée en fonction du temps de résolution d'un problème (l'unité de mesure est la seconde) ou en nombre d'opérations exécutées par seconde (*e.g* nombre d'opérations flottantes (FLOPS), nombre d'instructions (IPS) ou encore en nombre d'images traitées (frame/s)).
- la consommation énergétique qui est mesurée en watt (W).

Ces critères peuvent être combinés pour mesurer l'efficacité énergétique (FLOP/J) ou pour définir un front de Pareto.

Chaque classe de coprocesseurs traite les données différemment et offre un profil de performances spécifique pour la résolution d'un même problème. Typiquement, un CPU est bon pour faire des tâches de contrôle, un GPU est adapté aux calculs sur des données denses (vecteurs ou matrices) et un FPGA permet d'implémenter des accélérateurs matériels avec une architecture spé-

cifique à la résolution d'un problème. Les FPGA sont également utilisés pour faire l'acquisition et le traitement de données proche des capteurs dans les systèmes embarqués ou dans les systèmes de mesures scientifiques. L'utilisation d'un système hétérogène implique, pour le développeur, de définir la manière dont l'exécution d'une application doit être répartie sur les différentes ressources de calcul. Cette tâche nécessite une analyse du code qui est utilisée pour définir les portions dont l'intensité arithmétique est la plus élevée. Celles-ci sont désignées comme des noyaux de calcul. Ensuite, le développeur doit déterminer quelle ressource du système est la plus adaptée pour traiter chaque portion de l'application. Typiquement, les tâches de contrôle d'une application sont traitées par le ou les CPU et les noyaux de calcul sont exécutés sur des coprocesseurs agissant comme accélérateurs matériels.

La section 2.2 présente une classification des algorithmes de calcul scientifique et l'adéquation des coprocesseurs pour les traiter. La section 2.3 conclut ce chapitre.

2.2 Adéquations des architectures de coprocesseurs pour le traitement des algorithmes de calcul scientifique

Différents algorithmes de calcul peuvent être regroupés dans une famille lorsqu'ils montrent des ressemblances dans leur méthode pour opérer les calculs et accéder aux données. Ainsi, en 2004, Phillip Colella a identifié 7 familles d'algorithmes dans les applications de calcul scientifique [28]. Parmi celles-ci, la méthode de Monte-Carlo est assimilée à *Map Reduce* qui se rapproche davantage d'un modèle de programmation que d'une famille d'algorithmes de calcul. Nous utilisons la classification de Phillip Colella restreinte à 6 familles pour présenter les algorithmes utilisés dans les applications de calcul scientifique.

1–L'algèbre linéaire dense est une des familles d'algorithmes parmi les plus importantes pour les applications HPC et est utilisée dans beaucoup de solveurs. Ces algorithmes sont notamment employés par le banc de test *LINPACK* qui évalue les performances des superordinateurs du TOP500. Les données traitées sont généralement des entiers ou des nombres flottants orga-

nisées en vecteurs ou en matrices. *Basic Linear Algebra Subprograms* (BLAS) est un standard qui définit un ensemble de fonctions pour réaliser des opérations de l'algèbre linéaire creuse. Ces opérations sont réparties en trois niveaux de complexités calculatoires :

Niveau 1 : les opérations entre deux vecteurs ($\mathcal{O}(n)$)

Niveau 2 : les opérations entre une matrice et un vecteur ($\mathcal{O}(n^2)$)

Niveau 3 : les opérations entre deux matrices ($\mathcal{O}(n^3)$)

Plus le niveau de complexité d'une opération est élevé plus son intensité arithmétique est grande. Le parallélisme de données de ces opérations est facilement identifiable, ce qui rend les algorithmes faciles à paralléliser. L'architecture des processeurs graphiques étant optimisée pour traiter des matrices et des vecteurs, ils sont le type de coprocesseur le plus adapté pour exécuter les opérations d'algèbre linéaire dense. Leurs performances en termes de vitesse de calcul et de consommation énergétique dépassent celles pouvant être atteintes avec un FPGA ou un CPU.

2–L'algèbre linéaire creuse est une des familles d'algorithmes parmi les plus utilisées dans les applications de calcul scientifique. Elle répond aux mêmes problèmes mathématiques que ceux de l'algèbre linéaire dense, à la différence que les opérandes sont majoritairement peuplés de valeurs nulles. Cette sporadicité requiert d'utiliser une structure de données qui s'éloigne de la représentation d'un vecteur ou d'une matrice. Ces structures permettent de stocker uniquement les valeurs non nulles, mais dans le même temps obligent à mémoriser des informations permettant de reconstruire le motif de la matrice ou du vecteur. Les opérations d'algèbre linéaire creuse conduisent à des accès mémoires indirects et irréguliers. Étant donné que leurs opérandes contiennent majoritairement des éléments nuls, leur intensité arithmétique est beaucoup plus faible que celle des opérations de l'algèbre linéaire dense. Par conséquent, les performances d'exécution de ces algorithmes sont limitées par la bande passante du système mémoire (*memory-bound*). L'optimisation de ces algorithmes requiert d'apporter une attention particulière à la gestion des données. Ainsi, la taille des données peut être réduite en compressant les indices de colonnes ou de rangées. Le parcours des structures de données peut être optimisé pour limiter les échanges de données entre les différents niveaux de la hiérarchie mémoire. Ces deux optimisations permettent d'économiser la bande passante mémoire. Un FPGA peut adapter son architecture pour faciliter la mise en

œuvre de ces optimisations, par exemple en implémentant un décodeur d'indices compressés et en instanciant un cache de données adapté aux motifs d'accès que ces noyaux de calcul génèrent. Cependant, les cartes graphiques offrent une très large bande passante qui les rendent compétitives par rapport à un FPGA. Le banc de test *High Performance Conjugate Gradient* (HPCG) [1] permet d'évaluer l'efficacité d'une architecture à traiter des noyaux de calcul de l'algèbre linéaire creuse. Zeni *et al.* [119] ont comparé les performances de ce banc de test sur trois types de ressources de calcul : un CPU *Intel Xeon*, des GPU *Nvidia Telsa V100*, et un FPGA *Xilinx Alveo U280*. L'étude conclut que le FPGA atteint une vitesse de traitement 104.1 fois plus rapide que le CPU (lors d'un exécution utilisant les 8 cœurs en *hyperthreading*) et que le GPU traite les données 1.35 fois plus rapidement que le FPGA. Cependant, au regard de la consommation énergétique le FPGA est 2.7 fois plus efficace que le GPU et 408 fois plus que le CPU. Il faut également noter que la carte graphique *Nvidia Tesla V100* offre une plus grande bande passante mémoire que le FPGA. La même étude montre que le FPGA peut atteindre une vitesse de traitement supérieure à un GPU si la bande passante est similaire (*Nvidia Telsa P100*).

3–Les méthodes spectrales permettent de résoudre des équations différentielles pour des applications variées, telles que celles de dynamiques des fluides, de mécaniques quantiques ou de prévisions météorologiques. La résolution de ces équations se fait dans le domaine spectral et implique fréquemment l'emploi de la transformation de Fourier rapide. Pour un vecteur de taille n, cet algorithme a une complexité en $\mathcal{O}(nlog(n))$. Il est récursif et ses motifs d'accès aux données sont compliqués. Chaque donnée d'entrée de l'itération $i+1$ dépend d'une donnée de sortie de l'itération i. Cependant, les dépendances varient d'une itération à l'autre. Plus la dimension du problème physique est élevée (1D, 2D, 3D), plus les motifs d'accès aux données sont complexes. Le traitement de ces algorithmes se prête bien au flux de données et, par conséquent, un FPGA peut offrir de bonnes performances. Ainsi, Corda *et al.* [30] ont évalué les performances d'exécution d'un algorithme de transformation de Fourier rapide en $2D$ sur trois architectures : un CPU *IBM Power 9*, un GPU *Nvidia Tesla V100* et un FPGA *AlphaData 9H7*. Malgré une bande passante

1. `https://www.hpcg-benchmark.org/`

mémoire beaucoup plus élevée pour le GPU, l'étude conclut que le FPGA peut atteindre les mêmes vitesses de traitement que le GPU qui sont jusqu'à 120 fois plus rapides que celles obtenues avec le CPU. Cette étude n'apporte pas d'évaluation comparative de la consommation énergétique, cependant le FPGA a une enveloppe thermique (TDP) inférieure aux autres ressources et devrait donc consommer moins d'énergie. Sanaullah et *et al.* [93] ont réalisé une étude similaire pour la transformation de Fourier en 3D. Pour ce faire, ils ont comparé les performances d'un CPU *Intel Xeon* utilisant 14 cœurs, d'un GPU *Nvidia Tesla P100* et d'un FPGA *Intel Arria 10*. Les résultats obtenus montrent que le FPGA est, respectivement, 29 et 4.1 fois plus rapide que le CPU et le GPU. Cette étude n'apporte pas de précision sur la consommation énergétique. Cependant nous pouvons penser que le FPGA est plus efficace, d'une part, parce qu'il traite les données plus vite et, d'autre part, parce qu'un FPGA est plus économe en énergie.

4–Le problème à N-corps consiste à résoudre des équations du mouvement de Newton de N corps interagissant entre eux par force gravitationnelle. Pour un système à n corps, les algorithmes traditionnels de cette méthode ont une complexité calculatoire en $(\mathcal{O}(n^2))$. La simulation de Barnes-Hut permet de réduire la charge de travail pour obtenir une complexité en $(\mathcal{O}(nlog(n)))$. Pour ce faire, l'espace est découpé en cubes de manière récursive. Les cubes sont organisés en un arbre, où une feuille contient au plus une particule et un nœud (appelé pseudo-particule). Ce dernier représente le centre de masse de l'ensemble des particules qu'il réunit. La simulation se fait par l'itération de quatre étapes dans un intervalle de temps simulé [60] :

1. la construction de l'arbre.
2. le calcul des centres de masses et de la position des pseudo-particules.
3. le calcul des forces exercées entre particules.
4. la mise à jour de la position et de la vélocité de toutes les particules.

Le format de stockage de l'arbre en mémoire conduit à des accès aux données fortement irréguliers. De plus, la mise à jour de l'arbre entre les itérations nécessite de modifier sa structure, ce qui complexifie la gestion des données. Del Sozzo *et al.* [101, 100] ont implémenté un accélérateur matériel sur FPGA pour la simulation de problèmes à N-corps. Ils ont comparé les performances de cette implémentation à celles de plusieurs autres de l'état de l'art, dont une

sur un GPU *Nvidia Tesla K80* [88] et une sur un CPU *Intel Xeon* utilisant un seul des 8 cœurs [88]. Cette comparaison conclut que le FPGA a une vitesse de traitement 33.9 fois supérieure à celle du CPU et que le GPU est 2.7 fois plus rapide que le FPGA. Au regard de la consommation énergétique, le FPGA est, respectivement, $37,4$ et 1.8 fois plus efficace que le CPU et que le GPU.

5–Les maillages structurés permettent de modéliser dans un tableau à n dimensions la géométrie d'un objet par une discrétisation de son espace et d'y associer des valeurs physiques. Ces maillages sont notamment utilisés pour résoudre des équations par méthode des éléments finis. Typiquement, la valeur des éléments est mise à jour en appliquant une fonction de stencil : pour chaque case du tableau la même fonction est appliquée à partir des valeurs des éléments du voisinage. Étant donné que le maillage est structuré, sa topologie est implicite. Il en résulte des accès à la mémoire réguliers et un parallélisme de données facilement identifiable. Comme pour les algorithmes d'algèbre linéaire dense, la difficulté d'optimisation du traitement des maillages structurés est notamment liée à la taille des données. Lorsque les maillages sont trop volumineux pour être contenus dans les mémoires proches des unités de calcul, le parcours du maillage doit être optimisé pour limiter les échanges de données entre les différents niveaux de la hiérarchie mémoire du système. L'architecture d'un FPGA peut être adaptée au motif du stencil et ainsi permettre d'obtenir de bonnes performances. Kamalakkannan *et al.* [63] ont implémenté sur un FPGA *Xilinx Alveo U280* un accélérateur pour les solveurs numériques traitant des maillages structurés. Ils ont comparé les performances de cet accélérateur avec l'exécution d'un solveur sur un GPU *Nvidia Tesla V100*. Les résultats obtenus montrent que le FPGA atteint la même vitesse de traitement que le GPU tout en consommant deux fois moins d'énergie. Zohouri *et al.* [123] ont comparé les performances d'exécution de plusieurs noyaux de calcul utilisant un stencil sur des maillages réguliers bidimensionnels et tridimensionnels, lorsqu'ils sont traités par un FPGA *Intel Arria 10* commercialisé en 2014 ou un GPU *Nvidia Tesla K40c* commercialisé en 2013. Ils ont observé une vitesse de traitement 1.2 fois plus rapide et une efficacité énergétique 2.6 fois meilleure pour le FPGA. Dans cette même étude, ils utilisent les données obtenues pour estimer les performances d'un FPGA *Intel Stratix 10* et les comparer à celles d'un GPU *Nvidia Tesla V100*. L'étude conclut que ce FPGA serait 25% moins rapide que le second GPU, mais serait 1.6 fois plus efficace énergétiquement.

6–Les maillages non structurés ont la même fonctionnalité que les maillages structurés mais permettent de modéliser des formes géométriques plus complexes ou des problèmes physiques plus grands. Typiquement, un maillage non structuré utilise des éléments triangulaires dans un espace bidimensionnel et des tétraèdres dans un espace tridimensionnel. La résolution du maillage peut varier dans l'espace afin que les arêtes des éléments épousent les courbes d'un objet de forme irrégulière. Avec un maillage structuré, l'augmentation de la résolution doit s'appliquer à tous les éléments ce qui conduit à une augmentation exponentielle du volume de données et de la charge de travail. La modélisation d'un objet avec un maillage non structuré permet d'adapter la résolution dans l'espace ou dans le temps, et ainsi de réduire le volume de données. Cependant, le maillage étant non structuré, sa topologie doit être explicite. Comme pour les matrices et les vecteurs creux, une structure de données spécifique doit être utilisée pour décrire la topologie. Il en résulte des accès à la mémoire par indirections. Par conséquent, l'intensité arithmétique pour le traitement d'un maillage non structuré est plus faible que pour un maillage structuré. De plus, la topologie des maillages étant irrégulière, les accès aux données le sont aussi. Comme pour l'algèbre linéaire creuse, l'optimisation nécessite de concentrer les efforts sur la gestion des données. Les indices utilisés pour décrire la topologie peuvent être compressés pour réduire la quantité de données à transférer. Les données du maillage peuvent être réorganisées pour améliorer la localité des accès. Un FPGA permet de profiter de ces optimisations en implémentant des opérateurs pour décompresser les indices ou en implémentant un cache de données spécifiquement conçu pour le parcours d'un maillage non structuré. C'est ce que Barrio *et al.* [13] ont proposé de réaliser sur un FPGA, sans comparer les résultats obtenus avec d'autres types d'architectures. En 2009, Andres *et al.* [5] ont proposé une étude comparative des performances d'une application de dynamique des fluides sur des maillages non structurés en utilisant un FPGA *Xilinx Virtex 5*, un GPU *Nvidia Tesla S870* et un CPU *Athlon X2 4800+*. Avec un format à virgule flottante pour représenter les nombres réels, ils ont observé une vitesse de traitement 150 fois plus rapide avec le GPU et 236 fois plus rapide avec le FPGA que celle obtenue avec le CPU. En utilisant un format à virgule fixe, le FPGA obtenait une vitesse de traitement 778 fois plus rapide que le CPU. Cette étude n'apporte pas d'estimation de l'efficacité énergétique. Cependant, en traitant les données plus rapidement, le FPGA devrait avoir la consommation énergétique la plus faible.

Famille	Vitesse de traitement	Efficacité énergétique
1–Algèbre linéaire dense	GPU	GPU
2–Algèbre linéaire creuse	GPU	FPGA
3–Méthodes spectrales	GPU ≈ FPGA	FPGA
4–Problèmes à N-corps	GPU	FPGA
5–Maillages structurés	GPU ≈ FPGA	FPGA
6–Maillages non structurés	FPGA	FPGA

Tableau 2.1 – Synthèse de l'adéquation des coprocesseurs pour le traitement des 6 familles d'algorithmiques de calcul scientifique en fonction de la vitesse de traitement et de l'efficacité énergétique qu'ils permettent à partir de l'état de l'art connu

2.3 Conclusion

Le tableau 2.1 synthétise l'analyse présentée dans la section 2.2. Nous pouvons observer que, hormis pour l'algèbre linéaire dense, les FPGA permettent d'améliorer l'efficacité énergétique des applications scientifique. En ce qui concerne la vitesse de traitement, les GPU restent globalement plus performants que les FPGA. Pour être juste, l'étude comparative des performances d'exécution d'une application sur différents types de ressources de calcul requiert d'utiliser des composants de la même génération et du même niveau de gamme. La majorité des études présentées dans la section 2.2 est récente et utilise des FPGA et des GPU représentatifs du matériel intégré dans les systèmes contemporains. Cependant, il est important de noter que les GPU offrent des performances crêtes plus grandes en termes de bande passante mémoire et de puissance de calcul que des FPGA du même niveau de gamme. Malgré cette différence, l'architecture reprogrammable d'un FPGA permet de s'adapter aux besoins d'une application lorsque les motifs de calcul ou d'accès à la mémoire deviennent irréguliers.

Le banc de test *LINPACK* est utilisé pour classer les 500 superordinateurs les plus puissants (TOP500). Cependant, il est considéré insuffisamment représentatif des algorithmes de calcul utilisés par les applications de calcul scientifique "du monde réel". Le banc de test HPCG se présente comme une alternative jugée plus en adéquation avec ces applications. Il est également utilisé pour classer les 500 ordinateurs les plus puissants [2]. Ainsi, l'étude de Zeni *et al.* [119] se révèle particulièrement pertinente pour juger l'adéquation

2. `https://www.top500.org/lists/hpcg/`

des coprocesseurs pour le traitement des applications de calcul scientifique. En utilisant les conclusions de cette étude (présentée en section 2.2/2), nous pouvons considérer que les GPU restent plus performants en termes de vitesse de calcul et que les FPGA sont plus économes en énergie.

Au regard des objectifs du calcul hétérogène (augmenter l'efficacité énergétique des systèmes), les FPGA se présentent comme une ressource en passe de devenir essentielle. Un des freins à l'adoption générale des FPGA dans les systèmes de calcul est l'effort à fournir pour les programmer. Faciliter la programmation de ces ressources est donc un enjeu majeur.

Chapitre 3

Partage de données dans les systèmes de calcul à mémoires physiquement réparties

Un système de calcul hétérogène peut avoir une architecture parallèle unifiée par une mémoire physiquement partagée. C'est notamment le cas pour certains systèmes sur puces hétérogènes. L'architecture d'un système hétérogène peut également être distribuée, où chaque ressource de calcul dispose d'une mémoire privée. La figure 3.1 illustre ce deuxième type d'architecture où l'on peut observer une distribution hiérarchique des unités de traitement et des mémoires physiques. Ainsi, les nœuds de calcul ont une architecture distribuée et hétérogène. Le système se compose de plusieurs nœuds interconnectés par un réseau de communication et disposant chacun d'une mémoire privée. Ce type de système permet d'exécuter en parallèle un programme dans l'objectif d'en augmenter les performances, notamment en termes de vitesse de traitement et de consommation énergétique. Cependant, cette amélioration des performances entraine une augmentation de la complexité de programmation. Un programme traite, sous la forme d'un algorithme, des données structurées. L'exécution parallèle d'un programme sur un système distribué requiert de répartir sa charge de travail entre les différentes unités de traitement et de partager les données à traiter entre les différentes mémoires physiquement réparties.

Dans ce chapitre, et plus généralement dans le cadre des travaux de cette thèse, nous considérons la question de la gestion des données partagées entre

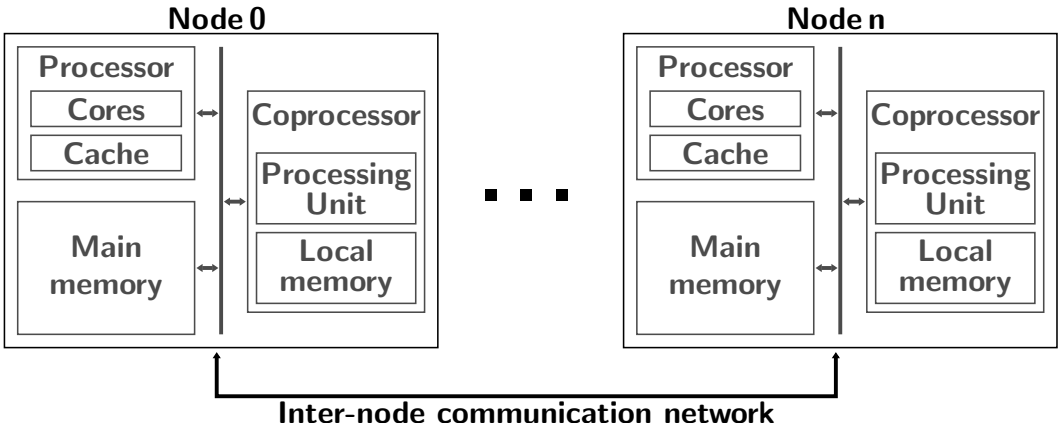
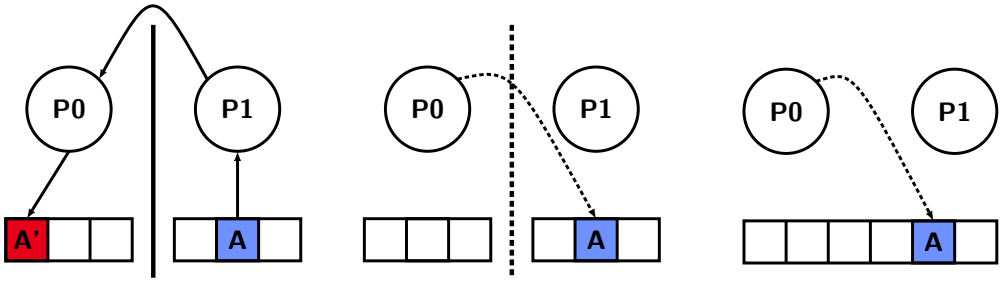


FIGURE 3.1 – Représentation haut niveau d'une architecture de système distribué hétérogène. Le système se compose de n nœuds de calcul reliés par un réseau de communication. Chaque nœud intègre une mémoire principale, un processeur et un coprocesseur disposant d'une mémoire locale.



(a) Passage de messages. (b) mémoire partitionnée. (c) Mémoire partagée.

FIGURE 3.2 – Modèles mémoires caractérisant le mécanisme d'accès aux données

des mémoires physiquement réparties.

L'évolution rapide des architectures de systèmes de calcul et des besoins applicatifs a conduit à la proposition de nombreux modèles de programmation reposant sur différents modèles mémoires. En considérant que l'entité qui exécute une portion d'un programme est un processus, un modèle mémoire décrit, d'un point de vue logique, comment un processus peut accéder aux données partagées. Historiquement, deux modèles mémoires s'opposent : le passage de messages et la mémoire partagée. Pour le modèle passage de messages, chaque processus possède un espace mémoire privé et le partage de données se fait par l'échange de messages. Tel que représenté sur la figure 3.2a, pour accéder à la donnée A contenue dans la mémoire du processus $P1$, le processus $P0$

doit attendre la réception d'un message contenant la donnée et provenant du processus $P1$ pour la copier dans son espace mémoire. Tel qu'illustré sur la figure 3.2c, pour le modèle mémoire partagée, l'espace d'adressage mémoire est unifié entre tous les processus qui peuvent lire ou écrire directement les données. Ce modèle semble plus pratique pour programmer une application dont plusieurs processus accèdent en parallèle à des données partagées. Cependant, lorsque les mémoires du système de calcul sont physiquement réparties, le modèle passage de messages est plus fidèle aux mécanismes opérant le partage des données. En abstrayant la répartition physique des mémoires, le modèle mémoire partagée peut conduire à des effets d'accès mémoire non uniformes (NUMA) parce que, pour un processus, l'accès à une mémoire physiquement distante est plus long que l'accès à une mémoire locale. Pour essayer de profiter des avantages des deux modèles sans subir leurs inconvénients, un troisième modèle mémoire a été proposé : la mémoire partitionnée. Pour ce modèle, l'espace d'adressage mémoire du programme est partitionné (PGAS). Il exploite le concept de mémoire locale et mémoire distante. Chaque processus dispose d'un espace mémoire local, mais peut également accéder directement à un espace mémoire distant (la mémoire locale d'un autre processus). Ainsi, le modèle mémoire partitionnée offre un espace d'adressage mémoire unifié, tout en laissant apparaitre la répartition physique des mémoires. Enfin, certains modèles de programmation reposent sur des modèles mémoires hybrides, par exemple en utilisant le passage de messages entre les nœuds et la mémoire partagée au sein des nœuds.

Ce chapitre s'organise de la façon suivante : la section 3.1 présente la mise en œuvre des trois modèles mémoires pour le partage de données entre les processeurs de plusieurs nœuds d'un système de calcul distribué, la section 3.2 présente des solutions pour partager des données dans un nœud de calcul hétérogène, la section 3.3 expose des solutions pour partager des données dans un système de calcul distribué avec plusieurs nœuds hétérogènes, enfin la section 3.4 conclut ce chapitre.

3.1 Partage des données entre les processeurs d'un système distribué multi-nœud

Les premiers systèmes de calcul étaient très coûteux, encombrants et manquait de moyens d'interconnexions. De ce fait, les premiers ordinateurs opéraient principalement de manière indépendante. À partir du milieu des années 1980, deux avancées technologiques ont permis l'apparition des systèmes de calcul distribués. La première est le progrès de l'industrie de la microélectronique, grâce à la Loi de Moore, rendant les microprocesseurs plus performants et moins coûteux. La seconde est l'invention des réseaux locaux (*Local-Area Network* (LAN)) permettant d'interconnecter des ordinateurs avec une vitesse de communication élevée. L'apparition des systèmes de calcul distribués a été suivie par la proposition de plusieurs modèles de programmation pour exécuter en parallèle des applications sur les différents processeurs d'un système de calcul. Pour mettre en œuvre ces modèles de programmation, plusieurs solutions de partage de données ont été proposées. La sous-section 3.1.1 illustre la paradigme du passage de messages avec le standard MPI. La sous-section 3.1.2 présente des langages de programmation exploitant le modèle mémoire partitionnée. La sous-section 3.1.3 expose des systèmes logiciels permettant de créer une mémoire virtuellement partagée pour un système distribué.

3.1.1 Le passage de messages avec le standard MPI

MPI, pour *Message Passing Interface*, est un standard de bibliothèque de fonctions conçu pour écrire des programmes s'exécutant en parallèle sur des systèmes de calcul à mémoires distribuées et utilisant le passage de messages pour communiquer. La première version de ce standard a été publiée en 1994 [48]. Depuis, le standard MPI a été largement adopté pour la programmation des systèmes de calcul distribués. Plusieurs implémentations de MPI, libres ou propriétaires, on été proposées telles que : OpenMPI [50], MPICH [57], IBM Spectrum MPI ou Intel MPI.

La parallélisation d'un programme avec MPI se fait par la création de plusieurs processus. Chaque processus est exécuté sur un cœur de processeur distinct et dispose d'un espace mémoire privé. Le partage de données entre processus se fait par l'envoi de blocs de données sous la forme de messages.

Le modèle de communication standard, proposé dans la première version de MPI, est un modèle point-à-point bilatéral. Cela signifie que la communication s'effectue entre deux processus, un processus émetteur et un processus récepteur, tous deux impliqués dans l'échange du message. Ce modèle requiert que les deux processus se synchronisent. Pour ce faire, le processus émetteur appelle une fonction d'envoi en spécifiant le processus destinataire, l'adresse et la taille du bloc de données à transférer. Le processus récepteur appel une fonction de réception en spécifiant le processus émetteur, la taille du bloc de données attendue et l'adresse de l'emplacement mémoire où copier les données. La synchronisation de l'envoi et de la réception du message doit, d'une part, permettre au processus émetteur de ne pas modifier le bloc de données avant que le message soit stocké en toute sécurité et, d'autre part, permettre au processus récepteur de ne pas accéder à l'emplacement mémoire destinataire avant que les données soient totalement copiées.

Deux modes de synchronisation sont proposés : bloquant et non bloquant. Avec le mode bloquant, l'exécution du processus émetteur est interrompue par l'appel de la fonction d'envoi jusqu'à la réception d'une notification d'acquittement indiquant que le message est stocké en toute sécurité. De la même manière, l'exécution du processus récepteur est bloquée par la fonction de réception jusqu'à la complétion de la copie des données dans son espace mémoire privé. Ce mode de synchronisation peut conduire à une grande latence de communication, notamment lorsque l'appel des fonctions par les deux processus n'est pas bien synchronisé et que le réseau n'est pas performant. Avec le mode non bloquant, le retour à l'exécution est immédiat après l'appel aux fonctions d'envoi et de réception, même si les données n'ont pas été postées ou reçues. La synchronisation est assurée, d'une part, pour le processus émetteur par l'appel à une fonction d'attente qui est bloquante jusqu'à la complétion de l'envoi et, d'autre part, pour le processus récepteur par l'appel à une fonction de test qui est bloquante jusqu'à la complétion de la réception. De cette manière, les deux processus peuvent masquer les latences de communication en poursuivant le fil de leur exécution pendant que le transfert des données s'opère.

Avec la première version de MPI, et pour les deux modes de synchronisation, quatre modes de communication point-à-point sont possibles. Ces modes définissent le mécanisme de transmission du message et le mécanisme de synchronisation. Pour les décrire, le concept de tampon mémoire est utilisé. Un tampon mémoire peut être l'espace mémoire destinataire du processus

récepteur ou un espace mémoire du système, c'est-à-dire de l'environnement d'exécution MPI. Un tampon système n'est pas visible pour les processus de l'application. La copie d'un message dans un tampon du système permet de découpler l'envoi du message de sa réception. Dès la copie du message dans un tampon, l'envoi est acquitté. Ce mécanisme peut permettre de réduire les latences de communication pour le processus émetteur, mais conduit à utiliser un espace mémoire plus grand et oblige une copie des données supplémentaire. Les quatre modes sont les suivants :

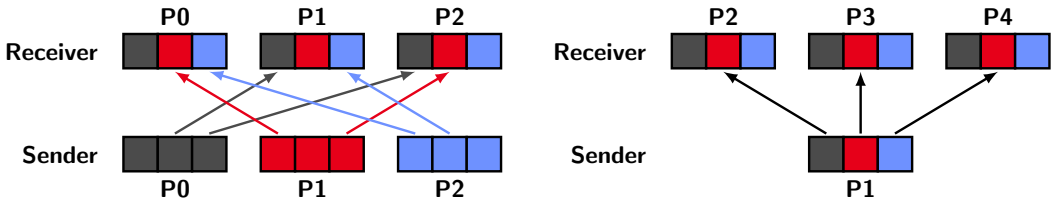
Standard : le message peut être copié directement dans le tampon destinataire ou dans un tampon du système local au processus destinataire. Le choix d'utiliser un tampon système est laissé à la discrétion de l'environnement MPI. Ce choix peut être basé sur la disponibilité ou l'indisponibilité d'un tampon système. Il peut être également basé sur des critères de performance. Typiquement, l'utilisation d'un tampon système est adapté à l'envoi d'un message de petite taille, car il requiert moins d'espace mémoire et engendre un mouvement de données moins important lors de la copie des données.

Tamponné : systématiquement, le message est stocké dans un tampon système local au processus émetteur. Ce mécanisme permet d'acquitter rapidement l'envoi d'un message, même si la fonction de réception n'a pas été appelée. Cependant, si l'espace mémoire alloué pour les tampons systèmes ne permet pas de contenir le message, parce qu'il est saturé ou mal dimensionné, l'envoi du message provoque une erreur.

Synchrone : ce mode utilise un protocole de communication en trois étapes : un premier message est envoyé au processus destinataire pour lui demander s'il est prêt à recevoir les données, ce processus répond lorsqu'il est prêt, puis les données sont envoyées par le processus émetteur. Lorsque la fonction d'envoi et la fonction de réception sont bloquantes, ce mode permet de créer un *rendezvous* dans l'exécution des deux processus.

Prêt : l'envoi d'un message peut débuter uniquement lorsque la fonction de réception correspondante a été postée. À défaut, l'envoi provoque une erreur. Ce mode permet d'alléger le protocole de communication.

En plus des communications point-à-point, MPI propose un ensemble de communications impliquant un groupe de processus MPI. Un groupe est appelé communicateur et doit contenir au moins deux processus. Les communications



(a) Communication All-to-All. (b) Communication Broadcast.

FIGURE 3.3 – Illustrations des communications collectives MPI.

collectives peuvent être utilisées pour envoyer un message à l'ensemble des processus du communicateur. Ainsi, une communication tous-vers-tous (*All-to-all*), tel qu'illustrée sur la figure 3.3a, permet de commuter des données entre tous les processus du communicateur. Pour chaque processus, la donnée d'indice i dans son bloc de données local est envoyée au processus i et la donnée provenant du processus j est stockée dans le bloc de données à l'indice j. Une communication de diffusion (*Broadcast*), tel que représentée sur la figure 3.3b, permet d'envoyer l'ensemble du bloc de données d'un processus à tous les autres processus du communicateur. À partir de la troisième version de MPI (MPI-3), les communications collectives peuvent être bloquantes ou non bloquantes.

Les communications bilatérales peuvent se révéler couteuses en termes de latence de communication, parce qu'elle implique l'intervention des processus émetteur et récepteur, et d'usage de la mémoire, parce qu'elles peuvent nécessiter l'utilisation de tampons systèmes. Pour pallier ce problème, des mécanismes d'accès distants à la mémoire (*Remote Memory Access* (RMA)) ont été ajoutés à la deuxième version de MPI (MPI-2). Ces mécanismes permettent de faire des communications unilatérales et ainsi de diminuer les coûts de communication en réduisant les coûts de synchronisation et en évitant l'emploi de tampons systèmes. De plus, les accès RMA peuvent être gérés sans surcoût logiciel grâce à l'émergence des réseaux d'interconnexions supportant les accès directs aux mémoires distantes (*Remote Direct Memory Access* (RDMA)), tels que les réseaux *RDMA over Converged Ethernet* (RoCE), *InfiniBand* (IB), *Omni-Path* ou *Myrinet*. Le partage des données avec les communications unilatérales se fait par la création d'un objet mémoire appelé fenêtre. Une fenêtre correspond à un espace dans la mémoire d'un processus qui est rendu accessible aux autres processus. Les accès RMA à la fenêtre sont encadrés par des primitives de synchronisation, en ouvrant la fenêtre pour y autoriser les accès

et en la fermant pour les interdire. D'un point de vue logique, les communications unilatérales se rapprochent d'un modèle mémoire partitionnée. En effet, l'espace mémoire est partitionné sous la forme de fenêtres et les accès aux données sont directs. Cependant, d'un point de vue sémantique, le partage de données se fait par l'envoi de messages.

Par rapport aux communications bilatérales, les communications unilatérales sont moins coûteuses en termes de latence, d'usage de la mémoire et de ressources du processeur, notamment grâce aux réseaux d'interconnexion de nouvelle génération. Cependant, les accès RMA peuvent être plus complexes à utiliser dans certains contextes. L'avantage des communications bilatérales est que chaque processus gère son espace mémoire. Pour une communication unilatérale, le processus initiateur d'un accès doit être suffisamment informé sur l'état de la mémoire distante pour gérer seul la communication.

> Lorsque le système de calcul est hétérogène et que les données sont irrégulières, cette gestion peut être coûteuse et complexe. Coûteuse, parce que les processus doivent stocker et traiter des informations pour gérer les mémoires distantes. Complexe, parce que l'irrégularité engendrée par l'hétérogénéité du système et les structures de données est entièrement exposée au développeur.

3.1.2 Les langages de programmation *Partitioned Global Address Space*

À la fin des années 1990, trois nouveaux langages de programmation ont exploité le concept du modèle mémoire à espace d'adressage globalement partitionné (*Partitioned Global Address Space* (PGAS)) :

- *Co-Array Fortran* [84],
- *Titanium* [118],
- *Unified Parallel C* [21].

Ces trois langages utilisent le modèle d'exécution parallèle "unique programme, multiple données" (*Single Program, Multiple Data (SPMD)*). Au lancement du programme, un nombre fixe de processus est généré. Chaque processus a un identifiant unique et est exécuté sur une unité de traitement spécifique. L'espace mémoire de chaque processus est divisé en deux catégories : un espace mémoire partagé et un espace mémoire privé. Les espaces mémoires partagés

de tous les processus sont unifiés dans un unique espace d'adressage global qui est partitionné. Chaque processus est lié à une partition qui est dite locale à ce processus. L'objectif est de tenir compte des domaines NUMA d'un système de calcul distribué. Ainsi, logiquement, la partition liée à un processus correspond à un espace de la mémoire physique du nœud de calcul sur lequel il s'exécute. De cette manière, l'accès à la partition locale d'un processus a un coût relativement faible et uniforme. Par opposition, un processus peut accéder, en lecture ou en écriture, à une partition distante, mais avec un coût plus élevé. Ainsi, le coût d'accès à la mémoire peut être défini en fonction de l'architecture matérielle sous-jacente et du partitionnement de l'espace d'adressage. Ce partitionnement permet d'abstraire l'architecture du système de calcul en décrivant une topologie d'interconnexion des partitions. La topologie la plus simple est une topologie plane : les partitions sont ordonnées par un indice compris dans l'intervalle $[0, n[$, où n est le nombre de partitions. La topologie des partitions est utilisée pour distribuer les données et y accéder.

Le Langage *Co-Array Fortran* est une extension du langage *Fortran*. Avec celui-ci, le partage de données se fait par la création de *coarrays*, dont la déclaration et l'accès se fait par l'utilisation syntaxique de crochets. La création d'une variable par la déclaration d'un *coarray* dans un programme à n processus aura pour effet l'allocation d'une instance différente de la variable dans les n partitions. L'accès, en lecture ou en écriture, à une instance de la variable dans une partition se fait par l'usage de l'index de la partition (i.e l'index de son processus). Par exemple, l'accès à la donnée x située dans la partition p se fait par la syntaxe suivante : $x[p]$. La syntaxe de la déclaration d'un tableau *coarray* se fait par l'usage de parenthèses (i.e la syntaxe normale du langage *Fortran*). Le développeur d'un programme avec *Co-Array Fortran* n'a pas de contrôle explicite sur la distribution des données. Le langage *Titanium* utilise le paradigme de la programmation orientée objet et un dialecte *Java*. Le parallélisme se fait avec des *threads*. *Titanium* utilise une syntaxe particulière pour la déclaration d'un objet local ou partagé. La création d'un objet partagé, de manière similaire à un *coarray*, se fait par l'instanciation de l'objet dans plusieurs partitions. Cependant, contrairement à *Co-Array Fortran*, *Titanium* permet de choisir les partitions dans lesquelles seront créés les objets partagés. Un objet instancié dans n partitions peut être vu comme un tableau à n cases, où chaque thread modifie une case spécifique. L'échange de données se fait par l'invocation de méthodes sur les objets : *exchange* et *broadcast*. La

méthode *exchange* permet à tous les *threads* de partager à tous les threads (tous-vers-tous) leurs données. La méthode *broadcast* permet à un seul thread de propager sa donnée à tous les autres *threads*. Le Langage *Unified Parallel C* est une extension du langage *C*. Le partage de données se fait également par une syntaxe spécifique lors de la déclaration en préfixant le type de l'objet déclaré par le qualificatif *shared*. Avec ce langage, tout type d'objet peut être partagé. Le développeur peut exprimer la distribution des données lors de la déclaration en spécifiant une taille de bloc. Cette taille est utilisée pour découper l'objet partagé en n blocs distribués dans les partitions de manière cyclique. L'accès aux données partagées est transparent : aucun élément de syntaxe ne permet de distinguer l'accès à une donnée locale d'un accès à une données distante.

Au milieu des années 2000, trois autres langages de programmation PGAS ont été proposés dans l'objectif d'augmenter la productivité de développement des programmes et de s'adapter aux nouvelles architectures des systèmes de calcul : *Chapel* [23], *X10* [24] et *Fortress* [2]. Ces langages offrent une gestion plus implicite et dynamique du parallélisme d'exécution. Ils permettent aux développeurs de gérer explicitement la distribution des données, que la structures des données soit régulière ou irrégulière. Enfin, ces langages utilisent un adressage global des données, contrairement aux *coarrays* qui sont adressés par l'indice de la partition.

Plus récemment, le langage *XCalableMP* [69] a été proposé comme une extension des langages *C* et *Fortran* sur le modèle de OpenMP. L'objectif est de pouvoir étendre le modèle de programmation de OpenMP à un système de calcul distribué. Le partage et la distribution des données se fait par l'emploi d'un ensemble de directives au compilateur (*#pragma*). De cette manière, *XCalableMP* permet d'exprimer le parallélisme de tâches et le parallélisme de données avec la même logique de syntaxe.

3.1.3 Les Mémoires Virtuellement Partagées logicielles

Le paradigme de la mémoire partagée est pratique pour écrire une application dont plusieurs processus accèdent en parallèle à des données partagées. Pour un système de calcul où la mémoire est centralisée, tel qu'une architecture de Von-Neumann classique, le partage de l'espace mémoire peut être réalisé par des mécanismes matériels fiables. Pour ce type d'architecture, l'ac-

cès aux données partagées se fait de manière transparente du point de vue du programme. Lorsque l'architecture du système de calcul est distribué et que les mémoires sont physiquement réparties, la mise en place d'une mémoire partagée, permettant aux processus d'accéder de manière transparente aux données, requiert l'emploi supplémentaire d'un système logiciel ou matériel. Dans ce manuscrit, un système partageant des données par du logiciel est désigné en français comme une Mémoire Virtuellement Partagée (MVP) logicielle et en anglais par *Software* Distributed Shared Memory (DSM).

Une MVP fournit un espace d'adressage virtuel qui est partagé entre tous les processus du programme exécutés sur le système distribué. L'espace d'adressage permet d'unifier les espaces mémoires physiquement répartis. N'importe quel processus peut accéder à n'importe quel espace de la mémoire directement en utilisant l'espace d'adressage virtuel. Un protocole de consistance mémoire est utilisé pour maintenir les données cohérentes. La mise à jour des données est réalisée par la Mémoire Virtuellement Partagée (MVP) et est rendue transparente pour le programme. Au fil de l'évolution des architectures de systèmes de calcul distribués, différentes MVP logicielles ont été proposées. Elles se distinguent par des choix d'implémentations différents, répondant à des besoins spécifiques, et par la manière dont l'espace mémoire partagé est exposé au développeur.

L'étude des MVP logicielles a suscité de l'intérêt à la fin des années 1980 avec la démocratisation et la multiplication des ordinateurs dans les entreprises. Ces ordinateurs reliés entre eux permettaient de former des grappes de calcul. Ainsi, le système *IVY* [75] est la première MVP logicielle à avoir été proposé. Il est implémenté par une modification du système d'exploitation. Avec ce système, le partage des données se fait à la granularité d'une page mémoire et la cohérence des données partagées implique l'unité de gestion mémoire (*Memory Management Unit* (MMU)). Une page peut être accédée en lecture par plusieurs processus en même temps et par un unique processus en écriture. Une politique d'invalidation est employée pour assurer la cohérence. Ainsi, toutes les copies d'une page accédée en lecture sont invalidées jusqu'à ce que l'écriture de la page soit réalisée. La MMU permet de détecter l'accès aux pages. De cette manière, lorsqu'un accès provoque un défaut de page, une routine du système d'exploitation est appelée pour orchestrer la mise à jour des données. Cette mise à jour implique un module qui centralise la gestion des copies des pages. Le système *IVY* permet de partager l'espace mémoire

des processus en toute transparence. Cependant, l'utilisation des pages pour orchestrer le partage des données peut avoir des effets néfastes sur les performances. Par exemple, si un processus modifie un seul octet dans une page qui est accédée en lecture par tous les autres processus, toutes ces pages devront être invalidées, puis mises à jour. Le bon dimensionnement des pages est donc critique. L'utilisation de pages trop petites conduira à une augmentation du nombre de défauts de pages pour l'usage normal du cache, tandis que l'utilisation de pages trop grandes peut conduire à de la contention sur le réseau d'interconnexion et augmenter les risques de faux partages [1]. Sur le même modèle de partage de pages, les systèmes *Munin* [22] et *Midway* [15] ont proposé chacun l'implémentation d'une MVP pour les grappes d'ordinateurs. *Munin* a introduit l'utilisation de plusieurs implémentations d'un même protocole de consistance mémoire (*release consistency*). Ce système repose sur l'analyse du développeur pour annoter le code en indiquant les motifs attendus d'accès aux données partagées. Ces annotations sont utilisées pour choisir l'implémentation du protocole de consistance le plus adapté à chaque page. Le système *Midway* a introduit l'utilisation du modèle de consistance mémoire *entry consistency* et la possibilité d'utiliser plusieurs modèles de consistances dans le même programme pour différentes données : *processor consistency*, *release consistency* ou *entry consistency*. Avec le modèle *entry consistency* l'accès aux données partagées est encadré par des instructions d'acquisition et de libération.

TreadMarks [3] est une *Application Programming Interface* (API) qui instancie une MVP entre un ensemble de processus exécutés sur un système distribué, toujours par le partage de pages, mais qui ne requiert pas de modification du système d'exploitation. Le système *TreadMarks* utilise un modèle de consistance mémoire *lazy release consistency* : les données sont mises à jour au moment où une demande d'accès est émise. *DSM-PM2* [6] est une plateforme étendant le modèle de programmation multitâche POSIX aux systèmes de calcul distribués en construisant une MVP logicielle. Cette plateforme logicielle a été développée dans l'intention de concevoir, d'implémenter et d'expérimenter des protocoles de cohérence *multithread*. Ainsi, *DSM-PM2* permet à un développeur de définir un protocole de cohérence spécifique. La communication

1. Le faux partage est une situation où plusieurs processus accèdent à des données différentes mais qui sont contenues dans le même bloc de données (ici une page). Les mécanismes de cohérence de données étant appliqués à la granularité d'un bloc, cette situation conduit à mettre à jour inutilement des données.

entre les nœuds se fait par des *Remote Procedure Call* (RPC), permettant à un processeur d'invoquer une routine sur un processeur distant.

Mermaid [122] et *Jade* [90] sont deux MVP logicielles développées pour les systèmes de calcul hétérogènes, où l'hétérogénéité provient notamment de la diversité des architectures de processeurs. Ces systèmes traitent le problème de l'hétérogénéité en se concentrant en particulier sur la conversion des données pour s'adapter à différents types de représentations en mémoire. Le système *SAT* [31] est une API permettant de créer une MVP pour les systèmes de calcul distribués intégrant des processeurs hétérogènes. Ce système est utilisé dans le cadre des travaux de cette thèse et son fonctionnement est détaillé dans la section 4.1.

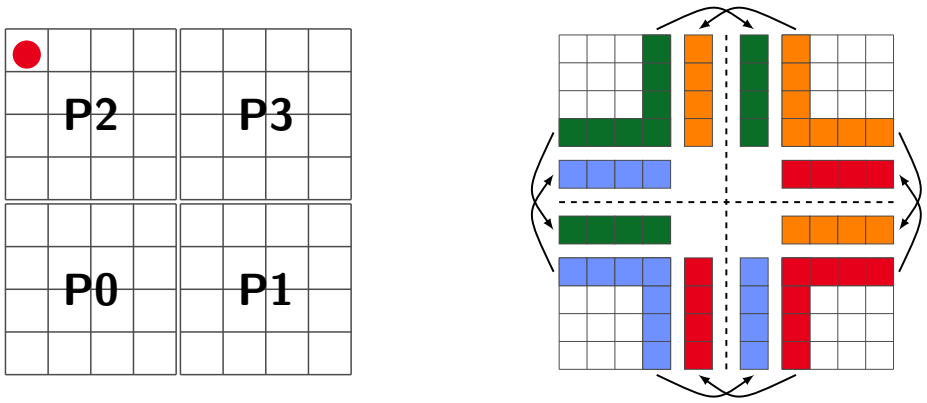
Plus récemment, les systèmes *Grappa* [83] et *Argo* [64] ont été proposés pour créer une MVP entre les processeurs de systèmes largement distribués. Ces deux systèmes exploitent les progrès des réseaux d'interconnexions pour réduire le coût des communications en exploitant les capacités RDMA. Ces deux systèmes utilisent MPI comme couche de communication. *Grappa* permet le partage de données avec une granularité fine. Cette granularité implique un nombre de messages plus important pour mettre à jour les données. Pour réduire le coût des communications, le système *Grappa* utilise l'agrégation de messages. Le système *Argo* partage les données entre des tâches à la granularité d'une page mémoire (typiquement 4 kio). Avec ce système, la gestion des données partagées est distribuée avec une politique *home-based* : la gestion de chaque page est assignée à un nœud de calcul en particulier. Pour limiter le coût du maintien de la cohérence des données, chaque nœud utilise un cache local pour mémoriser les pages distantes. Le mécanisme de cohérence repose sur l'auto invalidation des pages. Toutes les tâches peuvent lire n'importe quelle page partagée tant qu'elles s'engagent à l'invalider avant un certain point de synchronisation. De cette manière, une page lue n'a pas besoin d'être traquée par le nœud en charge de sa gestion pour être invalidée avant une écriture. L'objectif de cette stratégie est d'éliminer le goulot d'étranglement qu'une gestion centralisée peut provoquer et de permettre au protocole de cohérence de passer à l'échelle lorsque le nombre de tâches augmente.

3.1.4 Discussion

Le passage de messages avec le standard MPI, les langages de programmation PGAS et les mémoires virtuellement partagées logicielles sont trois moyens de partager des données entre les processeurs d'un système distribué multi-nœud. Chacune de ces approches exploite un modèle mémoire différent. Le passage de messages peut être vu comme le paradigme la plus plus fidèle aux moyens de communications permettant le partage de données entre des mémoires physiquement réparties. Pour preuve, plusieurs solutions présentées dans cette section, mettant en place une mémoire partitionnée ou virtuellement partagée, utilisent MPI comme couche de communications. Cependant, les communications bilatérales avec MPI peuvent devenir complexes à utiliser. Dans un premier temps, pour une question de performance, les exécutions du processus émetteur et du processus récepteur doivent être bien synchronisées, de sorte à minimiser les latences de communications et à éviter le recours à l'utilisation de tampons systèmes qui sont coûteux en termes d'usage mémoire et de trafic de données. Dans un deuxième temps, une erreur dans l'écriture du code peut rapidement conduire à une situation d'interblocage. Cette assertion est vraie d'une manière générale lorsqu'il est question de programmation parallèle. Cependant, la bijection entre l'appel à une fonction d'émission et l'appel à celle de réception peut amplifier les risques d'interblocages. Enfin, l'émission d'un message suppose de connaître les besoins d'un autre processus et donc son activité. De manière similaire, la réception d'un message nécessite de connaître l'activité du processus émetteur. Lorsque le programme réalise un traitement régulier des données, ce point n'est pas forcément problématique. Cependant, lorsque le traitement est irrégulier, suivre l'activité de l'ensemble des processus peut être soit complexe, soit impossible à réaliser de manière raisonnable [2].

Pour illustrer le problème, prenons l'exemple d'un cas applicatif : la simulation de la propagation d'un évènement sur une surface plane modélisée par une grille bidimensionnelle. Considérons que la méthode de calcul soit une convolution : la valeur d'une case de la grille (représentant une zone de la surface) est mise à jour en fonction de la valeur des cases voisines sur plusieurs itérations. Considérons également que l'exécution est répartie entre quatre processus qui traitent chacun un quart de la grille, tel que représenté sur la figure 3.4a.

2. Le caractère raisonnable peut être déterminé pour une analyse coût-bénéfice.



(a) Représentation d'une grille divisée en 4 sous-domaines.

(b) Illustration du mécanisme d'échange de données en *Halo*.

FIGURE 3.4 – Représentation de la décomposition d'une grille en sous-domaines et de l'application du mécanisme d'échange en *Halo*.

Maintenant, considérons que l'évènement simulé se propage de manière non uniforme à partir de la case située en haut à gauche, tel qu'illustré par le rond rouge sur la figure 3.4a. Tant que les voisins d'une case n'ont pas modifié leur valeur, la case ne doit pas être modifiée. Pour chaque itération, un processus peut savoir quelles cases ont été modifiées dans son domaine. Cependant, un processus ne peut pas savoir quelles cases ont été modifiées dans un autre domaine. Pourtant, entre chaque itération un processus doit connaître la nouvelle valeur de l'ensemble des cases voisines à son domaine. En utilisant des communications bilatérales, il parait compliqué d'échanger uniquement les valeurs des cases qui ont été mises à jour à la dernière itération. D'une part, parce qu'un processus peut attendre la réception d'un message alors qu'aucune case voisine n'a été mise à jour et donc qu'aucun message ne devrait être émis. D'autre part, parce que le nombre de cases mises à jour peut varier et, dans ce cas, le processus récepteur n'est pas en mesure de connaître la taille du message par anticipation. Pour répondre à ce problème, une technique populaire est d'utiliser un mécanisme de communication dit d'échange en *Halo*. Tel que représenté sur la figure 3.4b, à chaque itération, chaque processus envoie la rangée et la colonne voisine aux autres sous-domaines et de la même manière reçoit une rangée et une colonne. Ce mécanisme est simple à mettre en œuvre avec des communications bilatérales mais peut conduire à envoyer des données inutiles. Des communications unilatérales permettent de résoudre le problème de l'attente d'un message qui ne devrait pas être émis mais n'apportent pas

plus d'informations sur les données à transférer. De la même manière, des communications unilatérales permettent la mise en œuvre de mécanisme d'échange en *Halo* mais le transfert de données inutiles reste présent.

Les langages PGAS permettent d'abstraire les communications entre des nœuds de calcul nécessaires aux partages des données. Cependant, la prise en charge efficace de structures de données irrégulières et sporadiques reste un problème ouvert [38]. Le problème provient notamment de la topologie de partitionnement de la mémoire et de la distribution des données liée à cette topologie.

> Une mémoire virtuellement partagée logicielle permet à la fois d'abstraire les communications entre les nœuds de calcul et la répartition physique des mémoires. Une MVP apporte un support pratique pour la programmation parallèle et bien adapté pour l'allocation de structures de données irrégulières. Cependant, en abstrayant la répartition physique des mémoire, ces systèmes ne laissent pas apparaître le coût des accès mémoires distants.

3.2 Partage des données dans un nœud de calcul hétérogène

Comme exposé en introduction de ce manuscrit, les nœuds des systèmes de calcul deviennent de plus en plus hétérogènes par l'intégration de coprocesseurs. Cette évolution architecturale permet d'améliorer l'efficacité énergétique FLOP/W des systèmes. Les coprocesseurs sont utilisés comme des accélérateurs matériels pour exécuter les portions des applications dont l'intensité arithmétique est élevée. Cette amélioration des performances du système par l'hétérogénéité se paie au prix d'une plus grande complexité de programmation. Premièrement, le code d'une application doit être réécrit pour être adapté à l'architecture matérielle du coprocesseur. Deuxièmement, les données à traiter doivent être rendues accessibles au coprocesseur et le résultat doit pouvoir être récupéré. Cette section se concentre en particulier sur l'utilisation de processeurs graphiques (GPU), de processeurs massivement parallèles (*many-core*) et de FPGA comme accélérateurs matériels. Les GPU, qui à l'origine était conçus pour le traitement d'images, sont devenus largement populaires

pour le calcul générique (GPGPU). Ces dernières années, les FPGA suscitent de plus en plus d'intérêt pour la communauté HPC grâce à leur grande efficacité énergétique pour plusieurs classes d'applications. La communauté scientifique et les groupes industriels ont fourni beaucoup d'efforts pour faciliter l'usage de ces ressources de calcul en proposant des langages de programmation, des compilateurs, des API ou des environnements de développement. Dans cette section, nous allons nous intéresser à la manière dont ces propositions traitent le partage des données entre un processeur et ses coprocesseurs.

Du point de vue du système, un coprocesseur apparait comme un périphérique. Il en résulte un modèle de programmation maître-esclave : un coprocesseur (l'esclave) est sous l'autorité d'un processeur (le maître), appelé processeur hôte. Avec ce modèle, le processeur hôte ordonne l'exécution des noyaux de calcul sur l'accélérateur et orchestre les transferts de données. D'un point de vue logique, la gestion des données avec ce modèle maître-esclave se rapproche du modèle passage de messages. Pour beaucoup d'applications, notamment celles traitant des blocs de données réguliers, cette conception de l'accélérateur est suffisante et efficace. Cependant, pour des applications traitant des structures de données irrégulières et riches en pointeurs, ce modèle maître-esclave montre des limites en engendrant un coût de gestion des accélérateurs trop important. Pour cette raison, ces dernières années des propositions ont été faites pour rendre les accélérateurs plus autonomes, notamment en étendant leur capacité à accéder à la mémoire du processeur.

La sous-section 3.2.1 présente le partage de données avec le modèle maître-esclave et la sous-section 3.2.2 expose des solutions permettant de partager la mémoire d'un processeur aux accélérateurs matériels.

3.2.1 Le partage de données avec le modèle maître-esclave

Pour illustrer le modèle maître-esclave, considérons un système composé d'un processeur, d'une mémoire centrale et d'un coprocesseur appelé accélérateur. La mémoire centrale est gérée par le processeur, ainsi elle est appelée mémoire du processeur. L'accélérateur dispose de sa propre mémoire, appelée mémoire de l'accélérateur. Le paradigme du modèle maître-esclave retranscrit le fait que l'accélérateur travaille avec sa propre mémoire et que la mémoire centrale est sous l'autorité du processeur. Ainsi, l'utilisation de ce modèle requiert l'allocation d'un espace dans chacune des mémoires et le transfert des

données entre ces deux espaces. Typiquement, la gestion des données pour l'exécution d'un noyau de calcul sur l'accélérateur avec ce modèle se fait en 7 étapes :

1. l'allocation d'un espace dans la mémoire du processeur pour chaque structure de données (opérandes et résultat).
2. l'initialisation des structures de données opérandes.
3. l'allocation d'un espace dans la mémoire de l'accélérateur pour chaque structure de données (opérandes et résultat).
4. le transfert des structures de données opérandes de la mémoire du processeur vers la mémoire de l'accélérateur.
5. le lancement du noyau de calcul sur l'accélérateur et l'attente de la complétion.
6. le transfert de la structure de données résultat de la mémoire de l'accélérateur vers la mémoire du processeur.
7. la désallocation des espaces mémoires alloués dans la mémoire de l'accélérateur.

Pour simplifier la gestion des données pour les systèmes hétérogènes, de nombreuses solutions ont été proposées afin d'abstraire en partie ces étapes et de les optimiser. Nous classifions ces solutions en deux groupes :

— les solutions basées sur le modèle de *OpenCL*, où les données sont représentées comme des objets et où ces étapes apparaissent explicitement dans le code.
— les solutions basées sur le modèle de *OpenMP*, où la gestion des données repose sur l'emploi de directives de compilations.

Le modèle *OpenCL*

OpenCL [65, 105] (*Open Computing Language*) a été proposé en tant que standard en 2008, par le groupe Khronos, pour permettre la portabilité des codes et faciliter la programmation des systèmes hétérogènes. L'objectif est de fournir au développeur la même vision abstraite de la plateforme matérielle pour tous types de systèmes hétérogènes. Le développement résulte en un code source qui est portable sur toute plateforme compatible avec *OpenCL*. Pour ce faire, le modèle de plateforme matérielle utilisé par *OpenCL*, appelé environnement de calcul, consiste en un hôte (le processeur hôte) connecté à un

```
1  /* Initialize environment */
2  cl_platform_id plt[1]; // platform ID
3  clGetPlatformIDs(1,plt,NULL);
4
5  /* Connect to a compute device */
6  cl_device_id dev[1]; // compute device id
7  clGetDeviceIDs(plt[0], CL_DEVICE_TYPE_DEFAULT,1,dev,NULL
       );
8
9  /* Create a context */
10 cl_context context; // compute context
11 context = clCreateContext(NULL,1,dev,NULL,NULL,NULL);
12
13 /* Create a command queue */
14 cl_command_queue command; // compute command queue
15 cmd = clCreateCommandQueue(context,dev[0],0,NULL);
```

Code 3.1 – Création de l'environnement, du *context* et de la file de commandes *OpenCL*

ou plusieurs périphériques (les accélérateurs). Chaque périphérique est associé à une file de commandes (*command queue*) qui permet d'ordonner l'exécution des noyaux de calcul, d'orchestrer le transfert des données et de synchroniser l'hôte et le périphérique. L'exécution d'un noyau de calcul se fait dans le cadre d'un *context* qui est créé avec un ou plusieurs périphériques. Un *context* est utilisé par le support exécutif de *OpenCL* pour gérer les files de commandes, la mémoire et les noyaux de calcul.

Pour illustrer la programmation avec *OpenCL*, prenons l'exemple du programme décrit dans les codes 3.1, 3.2 et 3.3 qui exécute un noyau de calcul quelconque traitant un vecteur opérande et produisant un vecteur résultat. La première partie du programme (code 3.1) consiste à mettre en place l'environnement de la plateforme matérielle (ligne 3, ici une plateforme avec un seul périphérique), à connecter le périphérique à la plateforme (ligne 7, un périphérique de n'importe quel type) et créer une file de commandes pour le périphérique (ligne 15). La deuxième partie du programme (code 3.2) est l'allocation des tampons dans la mémoire du processeur hôte (lignes 17 et 18), l'initialisation des données (ligne 19) et l'allocation des tampons dans la mémoire de l'accélérateur (lignes 22 à 24). Pour le tampon opérande, l'allocation se fait avec le paramètre `CL_MEM_READ_ONLY` qui indique que le noyau peut

```
16 /* Allocate and initialize host memory */
17 float * h_A = (float*)calloc(VECSIZE, sizeof(float));
18 float * h_B = (float*)calloc(VECSIZE, sizeof(float));
19 INIT(A,B);
20
21 /* Create memory buffers on the device */
22 cl_mem d_A, d_B; // device memory
23 d_A = clCreateBuffer(context,CL_MEM_READ_ONLY |
       CL_MEM_COPY_HOST_PTR,VECSIZE*sizeof(float),h_a,NULL);
24 d_B = clCreateBuffer(context, CL_MEM_WRITE_ONLY, VECSIZE
       *sizeof(float), NULL, NULL);
```

Code 3.2 – Allocation des tampons mémoires et transfert des données sources.

```
25 /* Create a program from the kernel source */
26 cl_program program; // compute program
27 program = clCreateProgramWithSource(context, 1,
              (const char**)&KernelSource, NULL, NULL);
28
29 /* Build the program */
30 clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
31
32 /* Create the compute kernel from the program */
33 cl_kernel kernel; // compute kernel
34 kernel = clCreateKernel(program, "kernel", NULL);
35
36 /* Set the arguments of the compute kernel */
37 clSetKernelArg(kernel,0,sizeof(cl_mem),(void *)&d_A);
38 clSetKernelArg(kernel,1,sizeof(cl_mem),(void *)&d_B);
39
40 /* Launch the kernel */
41 size_t global = VECSIZE;
42 clEnqueueNDRangeKernel(cmd, kernel, 1, NULL, &global,
       NULL, 0, NULL, NULL);
43
44 /* Read back the result from the compute device */
45 clEnqueueReadBuffer(cmd, d_B, CL_TRUE, 0, VECSIZE*sizeof
       (float), h_B, 0, NULL, NULL);
46
47 /* Then release all OpenCL allocated objects and host
       buffers */
```

Code 3.3 – Création du programme, lancement du noyau de calcul et transfert des données résultats.

accéder au tampon en lecture uniquement et `CL_MEM_HOST_PRT` qui indique que les données situées à l'adresse `h_A` doivent être copiées immédiatement dans le tampon. Pour le tampon résultat, le paramètre `CL_MEM_WRITE_ONLY` indique que le noyau peut accéder au tampon en écriture seulement. Enfin, la troisième partie du programme (code 3.3) consiste à exécuter le noyau de calcul et récupérer le résultat. Pour ce faire le noyau de calcul doit être créé, ce qui requiert 3 étapes : le chargement du code source du noyau dans un objet *program* (ligne 27), la compilation du noyau de calcul (ligne 30) et la création d'un objet *kernel* (ligne 34). La compilation à la volée introduit un surcoût en termes de temps d'exécution mais permet d'augmenter la portabilité du code. Après la création de l'objet *kernel*, ses arguments lui sont communiqués (lignes 37 et 38), puis il est exécuté (ligne 42). À la fin de l'exécution, le résultat est transféré dans la mémoire du processeur hôte (ligne 45). Le paramètre `CL_TRUE` indique que le transfert est bloquant.

Alpaka [120] et *Kokkos* [41] sont deux bibliothèques de fonctions proposées pour faciliter le développement de noyaux de calcul sur des accélérateurs de types *manycore* et GPU. Ces deux bibliothèques se démarquent de *OpenCL* sur plusieurs points : elles utilisent un seul code source, elles ne font pas de compilation à la volée, le style de programmation est beaucoup moins verbeux et elles se concentrent sur l'optimisation des accès aux données. Ces deux bibliothèques utilisent des `templates` *C++* pour allouer des structures de données complexes dans la mémoire des accélérateurs. Cependant, comme pour *OpenCL*, le développeur utilisant ces bibliothèques doit exprimer l'allocation de tampons dans la mémoire de l'accélérateur et ordonner le transfert des données de la mémoire du processeur vers celle de l'accélérateur et vice versa.

Le modèle *OpenMP*

OpenMP (*Open Multi-Processing*) [49] est une API reposant sur un modèle de tâche pour paralléliser des applications de calcul sur des architectures à mémoire partagée. Son utilisation est très populaire car elle nécessite peu de modifications d'un code séquentiel pour le rendre parallèle. En effet, seul l'emploi d'une directive au compilateur (`#pragma`) est nécessaire pour paralléliser une boucle `for`. Cependant, avant la version 4.0 sortie en 2013, *OpenMP* ne supportait pas les accélérateurs. Pour répondre à ce manque de support hétérogène, plusieurs API ont été proposées avant 2013 pour étendre le modèle de

```
1  // Initialization: |x[i]| < 1, i=0 ,... , size-1
2  #pragma acc data copy (x[0:size])
3  {
4    while(error > eps){
5      error = 0.0;
6  #pragma acc parallel present (x[0:size])
7  #pragma acc loop gang vector reduction (+:error)
8      for (int i=0; i < size; ++i) {
9        x[i] *= x[i];
10       error += fabs(x[i]);
11     }
12   }
13 }
```

Code 3.4 – Exemple de l'utilisation de directives *OpenACC*. Ce code est extrait de la publication [113].

programmation de *OpenMP* aux processeurs graphiques et aux accélérateurs *manycore* :

- *PGI* [58] développée par *The Portland Group*.
- *HMPP*[40] proposée par l'entreprise française *CAPS*.
- *OpenMPC* [70] permettant la génération d'un code *CUDA* par l'interprétation d'un code *OpenMP*.
- *StarSs* [10] qui a été proposée par le *Barcelona SuperComputing Center* comme une extension de *OpenMP*
- *OpenACC* [20] développée par des membres du comité *OpenMP Language*, dont les entreprises *CAPS* et *The Portland Group*. L'API *PGI* a servi de fondement à la spécification de *OpenACC*.

Pour illustrer l'application du modèle *OpenMP* à la programmation d'un système hétérogène CPU-GPU, nous prenons l'exemple de *OpenACC*. L'API est basée sur un modèle où la mémoire du processeur hôte et la mémoire des accélérateurs sont séparées et ne se synchronisent pas automatiquement. La gestion des données est guidée par le développeur avec l'emploi de directives. Pour illustrer l'utilisation de ces directives prenons l'exemple du code 3.4. À la ligne 2, la directive `acc data copy` permet d'indiquer que les données spécifiées dans la liste (ici le vecteur x) doivent être copiées dans la mémoire de l'accélérateur avant de rentrer dans la région matérialisée par le premier niveau d'accolades. La directive `acc parallel present` (ligne 6) indique que

l'exécution doit être parallélisée et que les données spécifiées dans la liste sont présentes en mémoires. Les directives `copy` et `present` peuvent être remplacées par d'autres directives permettant d'appliquer une gestion différente des données entre la mémoire du processeur et celle de l'accélérateur :

1. pour indiquer que les données doivent être transférées ou non en sortie de région.
2. pour indiquer qu'un espace doit être alloué dans la mémoire de l'accélérateur sans le peupler.
3. pour tester si des données sont présentes dans la mémoire de l'accélérateur et, dans la négative, appliquer la stratégie 1 ou 2.

L'utilisation des API citées précédemment permet d'augmenter significativement la productivité des développeurs à la fois en rendant le parallélisme du traitement des données presque transparent et en permettant d'orchestrer le transfert des données avec une simple ligne de code. Elles sont particulièrement bien adaptées au traitement de blocs de données réguliers (des vecteurs denses ou des matrices denses) et ainsi à la programmation de GPU ou de *manycore*. Cependant, avec le gain en intérêt des FPGA pour effectuer du calcul générique, Lee et al. [71] ont proposé un environnement de développement permettant d'étendre l'utilisation de *OpenACC* aux FPGA. Cet environnement utilise un compilateur source-à-source (*OpenARC*) qui prend en entrée un code *OpenACC* et génère un code *OpenCL* pour FPGA. L'environnement utilise ensuite le compilateur *Altera OpenCL* pour générer le fichier de configuration du FPGA (*bitstream*).

3.2.2 Le partage de données avec le modèle mémoire partagée

L'exploitation du modèle mémoire partagée pour un nœud de calcul hétérogène peut être permis par une approche logicielle permettant d'abstraire les mouvements de données entre la mémoire du processeur et la mémoire de l'accélérateur, afin de données au développeur la vision d'une mémoire partagée. C'est ce que Gelado et al. [53] ont proposé en partie avec l'API *A-SDSM* (*Asymmetric Distributed Shared Memory*). Cette API met en place une mémoire virtuellement partagée pour permettre au CPU d'accéder aux données allouées dans la mémoire de l'accélérateur, mais pas l'inverse (*i.e.* le GPU ne

peut pas accéder à la mémoire du CPU). De ce fait, la MVP est asymétrique. L'intérêt de cette API est de simplifier la gestion des données pour le développeur. Pour ce faire, elle repose sur un support d'exécution qui se superpose à celui de CUDA. Avec cette API, l'allocation d'un objet mémoire nécessite un seul appel de fonction qui a pour effet d'allouer l'objet dans la MVP. Lorsque c'est nécessaire (*i.e.* lorsqu'un noyau de calcul est lancé), le support d'exécution alloue un espace dans la mémoire de l'accélérateur et orchestre le transfert des données. Ainsi, le développeur n'a pas à gérer le transfert des données. La MVP repose sur un modèle de consistance mémoire *release consistency* : les données sont libérées lorsque le processeur invoque un noyau de calcul, puis sont acquises par le processeur lorsque l'accélérateur termine sa tâche. La principale limitation de l'API *ASDM* est son asymétrie qui ne permet pas à un noyau de calcul d'accéder à des structures de données irrégulières et riches en pointeurs sans qu'elles soient entièrement transférées dans la mémoire de l'accélérateur.

L'exploitation du modèle mémoire partagée pour un nœud de calcul hétérogène peut également être permis par une approche matérielle et logicielle en donnant la capacité à un accélérateur d'accéder à l'ensemble de l'espace d'adressage d'un processus exécuté sur le processeur hôte. C'est notamment ce qu'a proposé Pirmin Vogel dans le cadre des travaux de sa thèse de doctorat soutenue en 2018 [110]. Ces travaux ont proposé l'utilisation d'une *IOMMU (I/O Memory Management Unit)* pour permettre le partage de pages mémoires entre un processeur hôte et des noyaux de calcul implémentés sur un FPGA ou un processeur *manycore* programmable, c'est-à-dire implémenté sur un FPGA. Ce système permet de partager de manière transparente des données entre les deux ressources de calcul. Cependant, il est limité aux systèmes sur puces hétérogènes, c'est-à-dire des puces sur lesquelles sont implémentés la logique programmable du FPGA et un processeur embarqué.

QuickPath Interconnect (QPI), *Ultra Path Interconnect (UPI)* ou *Open Coherent Accelerator Processor Interface* (OpenCAPI) sont des nouvelles technologies de bus qui utilisent une connexion point-à-point entre un processeur et un accélérateur, notamment des FPGA. Ces bus permettent aux accélérateurs d'accéder directement à la mémoire du processeur sans perte de cohérence. Du côté de l'accélérateur, une couche protocole est implémentée pour lui permettre d'émettre des requêtes d'accès à l'espace mémoire du processeur. Le bus est directement interfacé au dernier niveau de cache du processeur. De

cette manière, il peut bénéficier des mécanismes du cache pour assurer la cohérence des données et pour accéder à la mémoire. La principale contrainte de ces technologies est d'utiliser des ports d'interconnexion spécifiques, ce qui limite leur utilisation. Les connecteurs *QPI* et *UPI* sont disponibles seulement sur certaines gammes de processeur *Intel*. Tandis que *Open Coherent Accelerator Processor Interface* (OpenCAPI), malgré le fait d'être proposé comme un standard par le consortium du même nom, n'est accessible que pour les processeurs *Power 9* et *Power 10* de *IBM*. L'accès à ces technologies concerne donc un marché de niche et ne favorise pas la portabilité.

3.3 Partage des données dans un système distribué multi-nœuds hétérogènes

Les systèmes de calcul peuvent être à la fois distribués sous la forme de plusieurs nœuds de calcul et hétérogènes avec l'intégration d'accélérateurs matériels dans les nœuds. La programmation de ces systèmes requiert de prendre en compte tout le parallélisme qu'ils offrent, que ce soit celui des unités de traitement ou celui des mémoires physiquement réparties. Ces systèmes exposent le développeur à une hiérarchie mémoire complexe et accumulent la complexité de programmation d'un système distribué avec des nœuds homogènes et d'un système de calcul avec un nœud hétérogène.

Dans cette section, trois types de solutions adressant la gestion des données partagées dans un système distribué hétérogène sont présentées : le modèle de programmation hybride (sous-section 3.3.1), l'utilisation d'un langage PGAS étendu aux systèmes hétérogènes (sous-section 3.3.2) et un support exécutif pour les systèmes distribués hétérogènes (sous-section 3.3.3).

3.3.1 Le modèle de programmation hybride

Comme exposé en introduction de cette section, l'utilisation de systèmes distribués et hétérogènes amène une grande complexité de programmation. D'une part, le développeur doit être en mesure d'exploiter le parallélisme d'un système qui peut compter plusieurs milliers d'unités de traitement hétérogènes. D'autre part la hiérarchie mémoire est en forme d'arbre :

1. les nœuds de calcul sont reliés par un réseau d'interconnexion.

2. chaque nœud intègre une mémoire centrale, un ou plusieurs processeurs et un ou plusieurs coprocesseurs, parfois de manière asymétrique (*i.e.* tous les nœuds de calcul du système n'ont pas le même architecture).
3. les coprocesseurs sont dépendants d'un processeur pour accéder à la mémoire centrale du nœud.
4. l'architecture de la mémoire interne aux processeurs et coprocesseurs est composée de domaines NUMA et exhibe également une hiérarchie en forme d'arbre (*e.g.* un processeur intègre généralement trois niveaux de cache) où la cohérence des données par des mécanismes matériels ne permet pas toujours d'obtenir les meilleures performances.

Face à cette complexité, l'usage d'un modèle de programmation hybride est devenu courant dans le domaine du calcul distribué et plus particulièrement du calcul distribué hétérogène. Un modèle de programmation hybride consiste à programmer un système de calcul en combinant plusieurs modèles de programmation. Pour le calcul distribué, cela consiste à utiliser un modèle pour gérer la programmation distribuée des nœuds de calcul et un second modèle ou plusieurs autres modèles pour gérer la programmation au sein des nœuds. L'intérêt de cette pratique est de diviser la complexité de programmation en sous problèmes et de traiter chaque sous problème avec un modèle de programmation adapté.

Le modèle de programmation hybride le plus fréquent pour le calcul distribué consiste à associer MPI à un autre modèle de programmation. Comme exposé dans la sous-section 3.1.1, MPI permet d'échanger des données par passage de messages entre des processus exécutés sur des nœuds distants. Cependant, MPI ne permet pas nativement de programmer des ressources hétérogènes. Ainsi, pour programmer ces ressources, une seconde API est employée. La programmation hybride avec *MPI* repose en partie sur le modèle *maître-esclave*. Les ressources de calcul sont programmées avec la seconde API au sein d'un processus *MPI*, appelé maître. Un ou plusieurs processus maîtres sont exécutés sur chaque nœuds de calcul. Ces processus sont en charge d'orchestrer les transferts des données entre les nœuds de calcul. Ensuite, la gestion des données au sein du nœud est à la charge de la seconde API. De cette manière, *MPI* peut être associé à *OpenACC* [46] ou *CUDA* [3] pour la programmation de grappes de GPU. *MPI* peut aussi être utilisé avec *OpenMP* [89] pour la pro-

3. `https://developer.nvidia.com/blog/introduction-cuda-aware-mpi/`

grammation de grappes de processeurs. Également, *MPI* peut être associé à un environnement de développement pour la programmation de nœuds de calcul intégrant CPU et FPGA afin de programmer des grappes de CPU-FPGA [91].

3.3.2 Extension d'un langage PGAS à la programmation hétérogène

Dans la section 3.1.2, *XcalableMP* a été présenté comme un langage de programmation PGAS développé sur le modèle de *OpenMP* pour permettre le partage de données entre des nœuds de calcul distants. Dans la sous-section 3.2.1, *OpenACC* a été présentée comme une API permettant d'étendre le modèle de programmation de *OpenMP* à la programmation de coprocesseur *manycore* et de GPU. *XcalableACC* [82] est une extension du langage *XcalableMP* utilisant *OpenACC* pour programmer des grappes intégrant des processeurs et des GPU. Ainsi, *XcalableACC* permet d'exprimer le parallélisme de données et les transferts de données sur l'ensemble de la plateforme par le simple usage de directives. Pour ce faire, *XcalableACC* regroupe les directives propres à *XcalableMP* et celles propres à *OpenACC*. Il en résulte un modèle d'exécution qui mélange celui de *XcalableMP* et *OpenACC*. Il considère un nœud de calcul comme étant composé d'un unique processeur (l'hôte) et d'un ou plusieurs accélérateurs. Chaque nœud exécute le même programme (SPMD) jusqu'à rencontrer une directive.

Comme pour un langage PGAS classique, les données peuvent être globales ou locales. Ces deux types de données peuvent être présentes dans la mémoire du processeur ou des accélérateurs. Les données globales sont celles qui sont distribuées dans les partitions mémoires entre les différents nœuds de calcul. Chaque partition est allouée sur un nœud en particulier. Les directives *OpenACC* permettent de transférer les données d'une partition d'un nœud vers la mémoire de ses accélérateurs. Les accès aux données des partitions distantes doivent être exprimés de manière explicite. L'accès aux données locales peut être effectué qu'au sein du nœud où elles résident.

XcalableACC apparait comme un langage pouvant grandement augmenter la productivité pour la programmation de systèmes de calcul distribués et hétérogènes.

3.3.3 Un support exécutif pour les systèmes distribués et hétérogènes

Un support exécutif peut être utilisé pour superviser l'exécution d'un programme et lui fournir des services. *StarPU* [9] est un support exécutif pour les systèmes de calcul hétérogènes qui a été développé dans un premier temps dans le cadre des travaux de la thèse de doctorat [7] de Cédric Augonnet. *StarPU* propose un modèle de programmation où une application est décrite sous la forme d'un algorithme séquentiel à base de tâches. Ces dernières sont un couple entre une *codelet* et des données. Une *codelet* est une structure qui modélise un noyau de calcul multi-versionné en précisant sur quelles architectures le noyau peut s'exécuter et en y associant les différentes implémentations. Les données sont déclarées sous la forme d'objets et sont liées aux noyaux de calcul. *StarPU* implémente une MVP logicielle qui n'est pas directement exposée au développeur. Elle est utilisée par le support exécutif pour traquer les données et les transférer ou les mettre à jour lorsqu'un noyau de calcul est lancé sur une des ressources. Pour ce faire, la MVP implémente un protocole de cohérence MSI (*Modified, Shared, Invalid*). Le support exécutif utilise les dépendances entre les noyaux pour créer un graphe orienté acyclique. Ce graphe est utilisé pour ordonnancer l'exécution des tâches sur les ressources. Différentes politiques d'ordonnancement peuvent être choisies ou implémentées par le développeur. Dans le cadre des travaux de la thèse de doctorat [27] de Georgios Christodoulis, l'intégration de FPGA dans le support exécutif *StarPU* a été proposée.

StarPU a connu plusieurs évolutions. Ainsi, *StarPU-MPI* [8] a été proposé pour permettre la programmation de systèmes hétérogènes distribués. Il propose un modèle de programmation hybride où une instance de *StarPU* est déployée sur chaque nœud et où MPI est utilisé pour faire communiquer les instances. Cependant, *StarPU-MPI* s'éloigne du paradigme de la programmation séquentielle à base de tâches. Plus récemment, *StarPU* a été étendu pour la programmation de systèmes de calcul hétérogènes et distribués, tout en gardant le paradigme original de programmation [1].

3.4 Conclusion

Dans ce chapitre, l'application de trois modèles mémoires comme paradigme pour la programmation de systèmes hétérogènes et distribués a été exposée. Chacun des modèles présente des avantages et des inconvénients.

Le modèle passage de messages est celui qui retranscrit le mieux l'architecture du système de calcul et les mécanismes mis en œuvre pour échanger des données. Cependant, son utilisation peut être complexe. D'une part, tel qu'illustré dans la sous-section 3.1.4, lorsque le traitement réalisé par une application est irrégulier, la gestion des données peut être difficile et inefficace. D'autre part, lorsque le système est distribué et hétérogène, le modèle passage de messages conduit à un modèle de programmation hybride. De ce fait, un développeur doit être en mesure de maitriser plusieurs modèles de programmation et les langages ou API qui les implémentent. Généralement, il en résulte une baisse de productivité et une augmentation du risque d'erreurs dans l'écriture du code.

Le modèle mémoire partitionnée est celui qui reflète le mieux la distribution des données. Il permet de prendre en considération les domaines NUMA d'une architecture avec des mémoires réparties et d'associer un coût aux accès distants. Le langage *XcalableACC* permet d'augmenter la productivité d'un développeur pour la programmation d'un système distribué et hétérogène, grâce à l'association du langage *XcalableMP* et de l'API *OpenACC*. Cependant, tel qu'indiqué dans la sous-section 3.1.4, la prise en charge efficace de structures de données irrégulières et sporadiques reste un problème ouvert.

Le modèle mémoire partagée est celui qui se rapproche le plus de la vue algorithmique d'une application en permettant le partage transparent et cohérent des données. Des systèmes logiciels, ont démontré depuis la fin des années 1980 la possibilité de mettre en œuvre une mémoire virtuellement partagée entre des mémoires physiquement réparties. Ces MVP logicielles ont su s'adapter à l'évolution de l'architecture des systèmes de calcul distribués dont le nombre d'unités de traitement a augmenté. Cependant, ces MVP n'apportent pas une solution complètement pour les systèmes hétérogènes actuels car les noyaux de calcul instanciés sur des accélérateurs n'ont pas un accès direct à l'espace mémoire partagée. Ces dernières années, des nouvelles technologies de bus d'interconnexions ont été proposées pour rendre accessible

l'espace mémoire d'un processeur depuis un FPGA sans perte de cohérence des données [25].

> L'apparition de ces technologies nous amène à penser qu'étendre l'espace d'adressage d'une MVP logicielle aux accélérateurs est une bonne perspective pour faciliter la programmation de systèmes de calcul distribués et hétérogènes.

Conclusion

Les systèmes de calcul deviennent de plus en plus hétérogènes. Une part de cette hétérogénéité provient de l'intégration de coprocesseurs agissant comme des accélérateurs matériels à la fois pour augmenter la puissance de calcul des systèmes et améliorer leur efficacité énergétique. Le TOP500 est une liste qui emploie le banc de test *LINPACK* pour classer les 500 ordinateurs les plus puissants du monde. Celui-ci utilise des noyaux de calcul d'algèbre linéaire dense afin d'évaluer la puissance de calcul maximale en FLOPS qu'un système peut atteindre (R_{max}). Le score obtenu peut alors être comparé à la puissance crête théorique du système (R_{peak}) pour en évaluer l'efficacité. Le Green500 est une liste qui classe les ordinateurs du TOP500 en fonction de leur efficacité énergétique (en FLOPS/W), c'est-à-dire, de leur R_{max} divisé par leur consommation énergétique. La conclusion du chapitre 1 (section 1.3) a souligné que la tendance à construire des systèmes de calcul hétérogènes s'observe dans l'architecture des superordinateurs du TOP500 et que le Green500 montre que cette hétérogénéité offre un gain en efficacité énergétique.

Cependant, le banc de test *LINPACK* n'est pas représentatif de toutes les applications de calcul. Le chapitre 2 expose une classification des algorithmes utilisés dans les applications de calcul scientifique en se basant sur leurs motifs d'accès aux données et leurs méthodes pour opérer les calculs. Cette classification met en avant que la majorité des familles d'algorithmes exhibe, soit des motifs d'accès aux données irréguliers, soit des motifs de calcul complexe, ou les deux. Le banc de test HPCG permet de mesurer les performances d'un système de calcul pour le traitement d'algorithmes dont les motifs d'accès à la mémoire sont irréguliers. Il provoque beaucoup d'accès aux données par rapport au nombre d'opérations effectuées conduisant à une faible intensité arithmétique. Ainsi, ce banc de test est *memory-bound*, c'est-à-dire que sa vitesse de traitement est limitée par les performances du système mémoire et du

système d'interconnexion. Il est souvent jugé comme étant plus représentatif des applications de calcul scientifique que le banc de test *LINPACK*. Il est également utilisé pour classer les ordinateurs du TOP500. La vitesse de calcul obtenue peut être comparée à la vitesse crête théorique pour évaluer l'efficacité d'un système à traiter des applications *memory-bound*. Ainsi, pour les résultats de novembre 2021 [4], les superordinateurs montrent une très faible efficacité comprise entre 0.21% et 5.57%. L'hétérogénéité de ces systèmes de calcul qui intègrent principalement des GPU ne semble pas apporter de gain sur ce critère, puisque l'efficacité la plus élevée pour un système hétérogène est seulement d'environ 2.24%. Par conséquent, l'efficacité énergétique des ordinateurs hétérogènes du TOP500 pour le traitement d'applications *memory-bound* est moins marquante que celle obtenue pour le banc de test *LINPACK*. La conclusion du chapitre 2 a mis en avant que les FPGA permettent d'atteindre une plus grande efficacité énergétique que les GPU pour le traitement de noyaux de calcul irréguliers, c'est-à-dire, des noyaux de calcul dont les motifs d'accès aux données ou de calcul sont irréguliers.

Les applications *memory-bound* stressent les systèmes mémoires et les systèmes d'interconnexion. L'optimisation des ces applications oblige à adopter une stratégie de gestion des données qui minimise les transferts entre les différents niveaux de la hiérarchie mémoire d'un système de calcul. Le chapitre 3 présente plusieurs méthodes pour partager les données entre les mémoires physiquement réparties d'un système de calcul. La mémoire partagée est un paradigme pratique pour la programmation d'applications dont plusieurs fils d'exécution accèdent en parallèle aux données de manières aléatoire et difficilement prévisible. Une MVP logicielle permet de construire un espace d'adressage unifiant des mémoires physiquement distribuées. Ces systèmes logiciels sont en charge de localiser les différentes répliques d'une donnée et des les maintenir cohérentes. Pour le développeur, leur utilisation permet d'optimiser la stratégie de gestion des données de manière transparente. En effet, une MVP agit comme un cache de données et ainsi transfère les données seulement lorsque c'est nécessaire. Dans ce sens, une MVP logicielle répond en partie à la problématique de la thèse : amener les bonnes données au bon endroit. Cependant, à notre connaissance, il n'existe pas de MVP logicielle qui réponde entièrement à la problématique de la thèse en étendant de manière uniforme son espace

4. `https://www.top500.org/lists/hpcg/2021/11/`

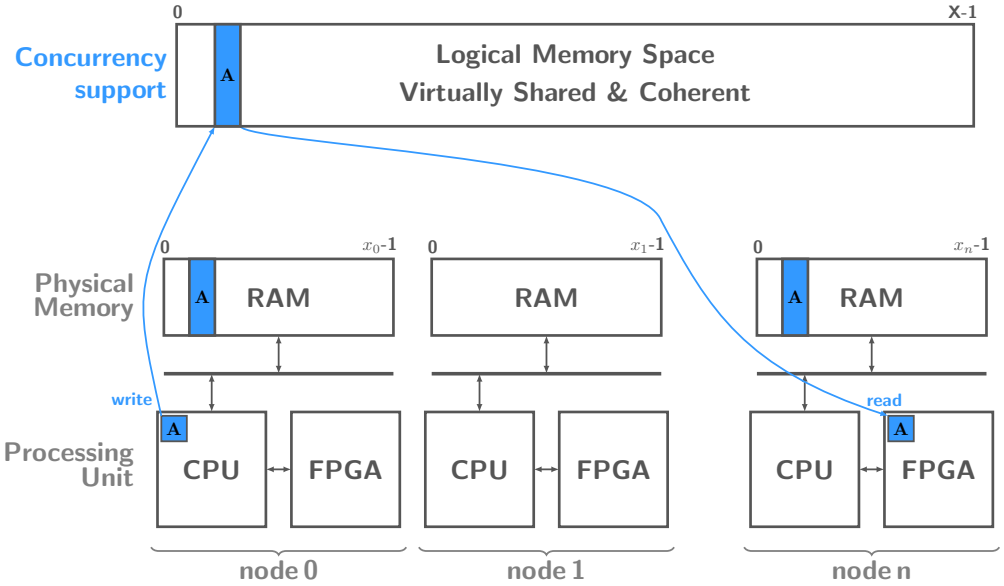


FIGURE 3.5 – Positionnement des travaux de la thèse : intégrer des accélérateurs reconfigurables dans une mémoire virtuellement partagée logicielle.

d'adressage aux coprocesseurs.

Positionnement des travaux de la thèse

Pour répondre à la problématique de cette thèse : "apporter une solution à la complexité de programmation des systèmes hétérogènes dont les mémoires sont physiquement réparties au travers du prisme de la gestion de données partagées et de l'importance d'amener les bonnes données au bon endroit et au bon moment", nous avons proposé d'étudier l'intégration d'accélérateurs reconfigurables dans une mémoire virtuellement partagée logicielle. Tel que représenté sur la figure 3.5, l'objectif de cette proposition est de permettre à des noyaux de calcul instanciés sur un accélérateur reconfigurable (FPGA) d'émettre des requêtes d'accès aux données allouées dans l'espace d'adressage de la MVP. Cette proposition s'adresse principalement à la programmation d'applications dont les noyaux de calcul sont irréguliers.

Deuxième partie

Intégration d'accélérateurs reconfigurables dans une mémoire virtuellement partagée logicielle : concept, modélisation et programmation

Introduction

Cette partie du manuscrit présente les contributions des travaux de la thèse. Comme introduit dans la conclusion de la partie I, le positionnement de ces travaux est l'étude de l'intégration d'accélérateurs reconfigurables dans une Mémoire Virtuellement Partagée (MVP) logicielle. La sous-section 3.1.3 à présente le principe d'une MVP logicielle. Pour mener les travaux de cette thèse, nous avons utilisé l'API SAT (*Share Among Things*) [31], une MVP logicielle dont le principe de fonctionnement est exposé dans la section 4.1. Le chapitre 4 présente la première contribution de cette thèse qui est un modèle d'intégration d'accélérateurs reconfigurables dans cette API. L'objectif est de faciliter la programmation de noyaux de calcul irréguliers sur des accélérateurs reconfigurables en les autorisant à émettre des requêtes d'accès aux données distribuées de la MVP.

Cette proposition nous a conduit à développer un modèle de programmation où l'accélérateur n'est plus vu comme un esclave du système et peut gérer de manière autonome l'accès aux données qu'il traite. Ce modèle de programmation présenté dans le chapitre 5 est la deuxième contribution de cette thèse.

Pour permettre la modélisation d'accélérateurs reconfigurables dans une MVP logicielle et le développement du modèle de programmation dans le temps de la thèse, nous avons utilisé le principe du manifeste pour le développement Agile de logiciels [14]. Celui-ci consiste à développer rapidement des prototypes incomplets mais validant certaines fonctionnalités. Pour y parvenir nous avons conçu un outil de simulation qui est la troisième contribution de cette thèse. Le fonctionnement et les motivations de la conception de cet outil sont présentés dans le chapitre 6. Il nous a permis de mener des expérimentations pour évaluer le modèle de programmation. Ces expérimentations et leurs résultats sont présentés dans la partie III de ce manuscrit.

Chapitre 4

Intégration d'accélérateurs reconfigurables dans une mémoire virtuellement partagée logicielle

Dans les travaux de cette thèse, nous avons utilisé une MVP logicielle, nommée SAT (*Share Among Things*) [31], qui est développée pour les micro-serveurs hétérogènes. Celle-ci a été conçue au CEA List à partir de 2015 dans le cadre du projet européen M2DC (*Modular Microserver DataCenter*) [85] pour répondre à la problématique de la gestion des données partagées sur des architectures de micro-serveurs. Le choix d'utiliser ce système a été motivé par plusieurs raisons. Premièrement, c'est un système qui est opérationnel et qui a été utilisé dans de nombreux travaux de recherche [31, 32, 107, 35, 104, 34, 33]. Deuxièmement, c'est un système qui est à notre disposition et qui a été déployé sur des plateformes de calcul de l'unité de recherche dans laquelle les travaux de cette thèse ont été effectués. Troisièmement, le système a été développé dans cette même unité de recherche, ce qui nous permet d'avoir une parfaite connaissance de son fonctionnement.

Le système SAT permet d'agréger dans un unique espace d'adressage logique les mémoires physiquement réparties d'un système distribué composé de plusieurs nœuds de calcul hétérogènes. Dans son état, les services du système SAT sont réservés à des processus logiciels exécutés par les processeurs intégrés dans les nœuds de calcul. Cette limitation est contraignante lorsque les données de la MVP doivent être partagées avec un coprocesseur. Pour ce faire, le développeur doit orchestrer manuellement le transfert des données, condui-

sant à une gestion des données hybride : le modèle mémoire partagée est utilisé entre les nœuds de calcul, à l'aide de la MVP, et le modèle maitre-esclave est utilisé entre le processeur hôte et l'accélérateur.

Les travaux de cette thèse ont pour objectif de rendre accessibles les services de la MVP à des accélérateurs reconfigurables afin de permettre une gestion des données homogène pour un système hétérogène intégrant de tels coprocesseurs. SAT est un système logiciel qui utilise *Message Passing Interface* (MPI) pour faire communiquer les processus et qui requiert certains services du système d'exploitation opérant sur les nœuds de calcul. Pour rendre accessible les services de la MVP aux accélérateurs reconfigurables, il est nécessaire de les intégrer dans l'environnement logiciel du système SAT. Nous proposons de réaliser cette intégration au moyen d'un intergiciel, qui se présente sous la forme d'un système d'interface permettant de faire le pont entre les tâches accélératrices implémentées sur la logique programmable d'un accélérateur et l'environnement logiciel de la MVP. Ce système d'interface repose sur un composant logiciel, appelé *FPGA-client*, un composant matériel, appelé *FPGA-server*, et un système permettant de faire communiquer ces deux composants. Le composant *FPGA-client* est un processus logiciel qui est exécuté sur un processeur et qui a pour rôle de représenter l'accélérateur auprès du système SAT. Le composant *FPGA-server* est un module implémenté sur la logique programmable de l'accélérateur et qui a pour rôle de reproduire les services de la MVP pour les tâches accélératrices.

Le chapitre s'organise de la manière suivante : la section 4.1 décrit les caractéristiques et le fonctionnement du système SAT, la section 4.2 présente le composant FPGA-server, la section 4.3 décrit le composant FPGA-client, la section 4.4 propose trois solutions pour interconnecter les deux composants et la section 4.5 expose les inconvénients et les propositions d'améliorations des travaux présentés dans ce chapitre.

4.1 Une mémoire virtuellement partagée logicielle pour micro-serveur hétérogène

Cette section présente le fonctionnement du système SAT utilisé comme MVP logicielle dans les travaux de cette thèse. Le développement de ce système ne fait pas partie des contributions de cette thèse. Cependant, la description de

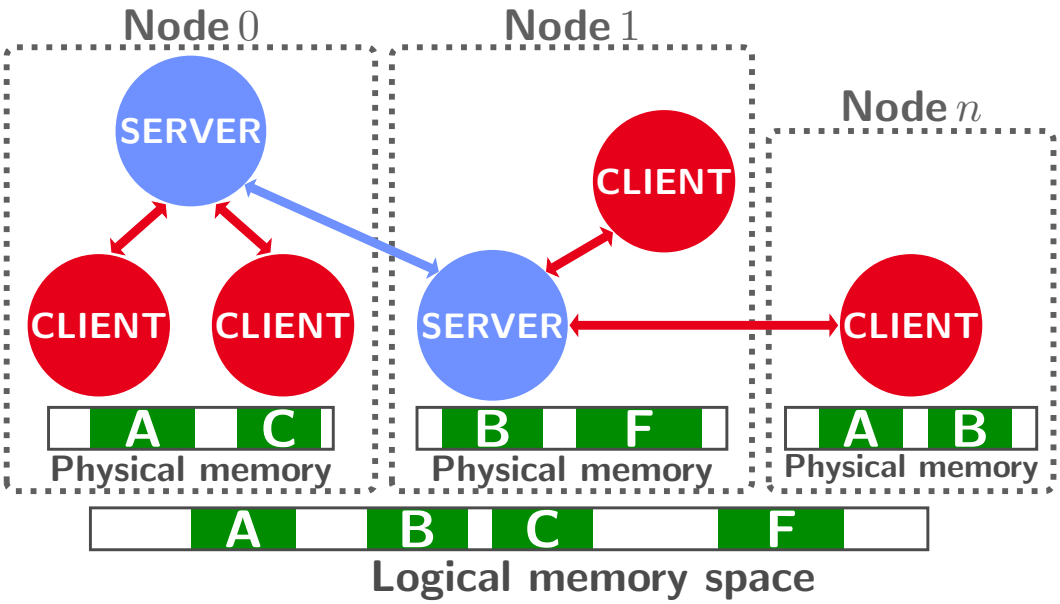


FIGURE 4.1 – Représentation de l'organisation du système SAT. La topologie du système est un réseau semi-structuré *super-peer*. Les cercles bleus représentent les processus serveurs connectés avec une topologie pair-à-pair. Les cercles rouges représentent les processus clients, où chaque client est connecté à un unique serveur. Les rectangles verts représentent les données partagées, sous forme de *chunks*, unifiées dans un même espace logique et distribuées dans les mémoires physiquement réparties.

son fonctionnement est nécessaire pour permettre une bonne compréhension de la suite du manuscrit.

SAT est une API qui construit une mémoire virtuellement partagée pour fédérer les mémoires physiquement réparties d'un système de calcul distribué. Cette MVP offre un espace d'adressage unifié où l'unité atomique de mémoire est appelée *chunk*. Un *chunk* est un bloc de données de taille variable auquel est associé des métadonnées et qui est identifié par un numéro unique qui constitue son adresse dans l'espace d'adressage de la MVP. La taille du bloc de données d'un *chunk* est définie lors de son allocation.

4.1.1 Topologie du système SAT

L'API repose sur un modèle de tâches où le code est organisé comme un ensemble de fonctions qui implémentent des rôles, qui peuvent être des fonctions du code utilisateur ou une fonction spéciale de l'API appelée serveur. Un serveur est en charge de l'allocation, de la réplication, de la localisation, du

transfert et de la suppression des *chunks*. Tel que le représente la figure 4.1, le système SAT est organisé comme un réseau semi-structuré dans lequel un ensemble de processus clients est connecté à un réseau pair-à-pair de processus serveurs. Les processus clients exécutent les rôles du code utilisateur et les processus serveurs exécutent le rôle de serveur. La communication entre les processus se fait par passage de messages. Pour ce faire, l'API utilise le standard MPI. Le système SAT utilise une description topologique de l'application pour répartir les processus sur le système de calcul. Cette description consiste à définir le nombre de processus instancié par rôle et à décrire leur connectivité. Cela signifie, pour un processus client, à définir le serveur auquel il est attaché et, pour un processus serveur, à définir l'ensemble des clients auxquels il est relié. La description topologique permet également de limiter l'espace mémoire physique utilisable pour chaque processus. L'intérêt est de pouvoir déployer la MVP sur des plateformes dont certains nœuds disposent de peu de mémoire vive et d'utiliser un stockage distant pour mémoriser l'ensemble des données de l'application. Pour premier exemple, une plateforme embarquée composée d'une mémoire centrale et d'une constellation de ressources de calcul dotées d'un faible espace mémoire. Pour deuxième exemple, un système distribué où la mémoire des coprocesseurs est beaucoup plus petite que la mémoire du processeur hôte.

4.1.2 Protocole de cohérence

La gestion de la MVP est répartie entre les différents processus serveurs. Pour ce faire, l'API adopte une approche dite *home-based* qui est largement répandue pour la conception de protocole de cohérence [68, 62, 17, 92]. Cette approche consiste à segmenter l'espace d'adressage et à placer chaque segment sous l'autorité d'un serveur en particulier. De manière similaire aux mécanismes matériels de cohérence d'un cache de données, un serveur est implémenté sous la forme d'un automate fini. L'API permet d'instancier plusieurs protocoles de cohérence mémoire et d'appliquer en même temps à chaque *chunk* un protocole différent. Par défaut, un protocole *MESI* est proposé. À l'état initial, l'automate du serveur est en attente d'une requête provenant soit d'un des processus clients auxquels il est attaché, soit d'un autre processus serveur. Lorsqu'un serveur reçoit une requête provenant d'un de ses clients pour un *chunk* dont il n'a pas l'autorité, il transmet la requête au serveur qui

```
1   /* SAT data strutures */
2   SAT_Consistency_t * cst    = newHomeBaseMESI();
3   SAT_Chunk_t       * chunk = NULL;
4
5   /* Chunk allocation */
6   uint32_t chunkid = 1, size = 10;
7   chunk = SAT_MALLOC(cst, chunkid, size);
8
9   /* Chunk access */
10  _SAT_READ(chunk);
11  foo(chunk->data);
12  _SAT_RELEASE(chunk);
```

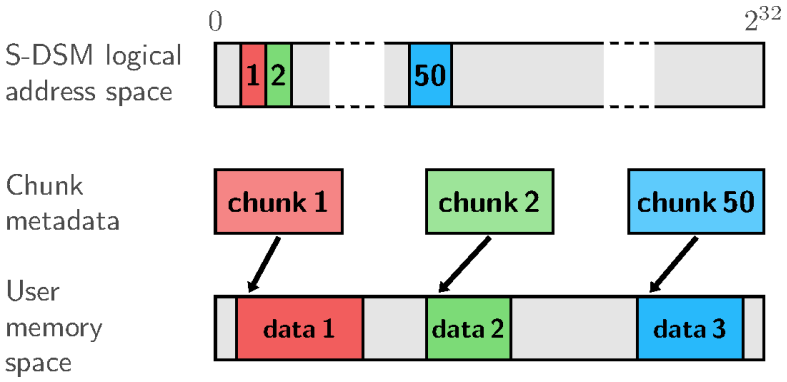
Code 4.1 – Exemples d'acquisition en lecture d'un chunk.

en a la gestion. La transition des états d'un serveur est définie par la nature de la requête et l'état de la MVP qui est en partie décrit par les métadonnées des *chunks*. Une requête correspond à une demande de service initiée par un processus client. La MVP utilise le modèle de consistance mémoire *scope consistency*. De cette manière, les accès aux *chunks* sont encadrés par quatre types d'instructions d'acquisitions et de libération : `read`, `readwrite`, `write` et `release`. Ces instructions exécutées par les processus clients encapsulent les appels aux services de la MVP pour mettre à jour l'état des *chunks*. Le code 4.1 illustre l'acquisition d'un *chunk* (ligne 10), l'accès à ses données (ligne 11), puis sa libération (ligne 12). Les instructions `read`, `readwrite` et `write` sont bloquantes tant que la transition d'état n'est pas possible. Les instructions `read` et `readwrite` ont pour conséquence de mettre à jour les données dans l'espace mémoire du processus appelant si le *chunk* est dans un état `Modified`.

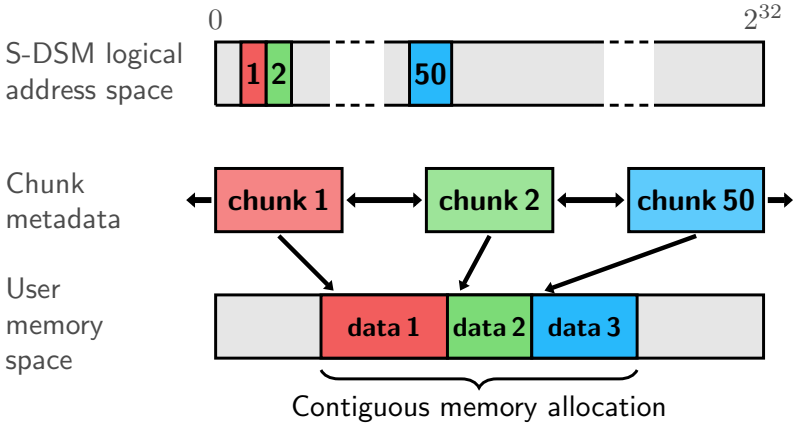
4.1.3 Allocation des données dans la mémoire partagée

L'allocation d'un *chunk* dans la MVP se fait par un appel de fonction en indiquant son identifiant, qui correspond à son adresse dans l'espace d'adressage de la MVP, et la taille du bloc de données à allouer. Les identifiants sont encodés avec des entiers *long non signés* offrant un espace de 4 milliards d'adresses[1]. Le bloc de données d'un *chunk* peut être de n'importe

1. Contrairement à une architecture de von Neumann classique, où l'adressage est une bijection entre une adresse et un octet mémoire, l'adressage dans la MVP est une bijection



(a) Allocation séparée de trois *chunks*.



(b) Allocation chaînée de trois *chunks*.

FIGURE 4.2 – Allocations des *chunks* dans l'espace mémoire physique et dans l'espace d'adressage logique de la MVP

quelle taille dans l'intervalle $[0, Taille_{max}]$. Ainsi, tous les *chunks* peuvent occuper une taille différente qui est mémorisée dans les métadonnées. La valeur $Taille_{max}$ peut être définie par l'utilisateur pour limiter la taille maximale que peut occuper un unique *chunk*.

Tel que le représente la figure 4.2b, l'allocation des *chunks* peut être réalisée de manière séparée ou de manière chaînée. Pour l'allocation séparée, illustrée sur la figure 4.2a, chaque *chunk* est indépendant. Ils occupent tous un emplacement différent dans l'espace d'adressage virtuel et leurs blocs de données sont alloués individuellement dans l'espace mémoire du processus utilisateur qui en a demandé l'allocation. Une allocation chaînée consiste à demander

entre un identifiant de *chunk* et un *chunk* qui permet d'accéder par indirection à un bloc d'octets.

```
1  /* SAT data strutures */
2  SAT_Consistency_t * cst   = newHomeBaseMESI();
3  SAT_Chunk_t       * chunk = NULL;
4
5  /* Separated allocation */
6  uint32_t chunkid = 1000;
7  uint32_t size = 197;
8  chunk = SAT_MALLOC(cst, chunkid, size);
9
10 /* Chained allocation */
11 uint32_t idlst[] = {1,2,50};
12 uint32_t sizelst[] = {1024,200,1000};
13 chunk = SAT_MALLOC_LST(cst,idlst,sizelst);
14
15 /* Separated lookup */
16 chunk = SAT_LOOKUP(cst, chunkid);
17
18 /* Chained lookup */
19 chunk = SAT_LOOKUP_LST(cst, idlst);
```

Code 4.2 – Exemples d'allocations de chunks.

l'allocation d'une liste de *chunks*. Pour ce faire, l'utilisateur fournit une liste d'identifiants de *chunks* et une liste de tailles de blocs. Tel que représenté sur la figure 4.2b, lors d'une allocation chaînée, un seul espace est alloué dans la mémoire du processus utilisateur. Cet unique espace est segmenté pour contenir les données de l'ensemble des *chunks* sujets à l'allocation. Pour chaque *chunk*, les métadonnées contiennent un pointeur qui indique le début de son segment de données. Les *chunks* sont reliés entre eux par une liste doublement chaînée qui est également mémorisée dans les métadonnées. Lors d'une allocation séparée, si la taille du bloc de données excède la taille maximale d'un *chunk*, l'allocation est segmentée en plusieurs *chunks* alloués de manière chaînée avec des identifiants consécutifs. L'allocation contigüe des données en mémoire permet de rendre cette segmentation transparente pour l'utilisateur, qui peut parcourir l'ensemble du bloc de données en utilisant le pointeur du premier *chunk* et en utilisant de l'arithmétique de pointeur.

Du point de vue du code utilisateur, un *chunk* est simplement une structure de données dont un des champs est un pointeur donnant accès à son bloc de données. Tel que le code 4.2 l'illustre, l'API fournit quatre fonctions per-

mettant d'allouer un *chunk* dans l'espace mémoire d'un processus utilisateur :

`SAT_MALLOC` permet d'allouer individuellement un *chunk* dans l'espace mémoire du processus appelant. Le *chunk* fait l'objet d'une allocation dans la mémoire virtuellement partagée s'il n'existe pas dans l'espace d'adressage. Si le *chunk* est déjà alloué dans l'espace mémoire du processus, la fonction renvoie une structure de données pointant sur le bloc de données existant. Sinon, un bloc de données correspondant à la taille passée en argument de la fonction est alloué.

`SAT_MALLOC_LST` permet de faire une allocation chaînée d'une liste de *chunks* dans l'espace mémoire du processus appelant. Comme pour la fonction `SAT_MALLOC`, chaque *chunk* qui n'est pas existant dans l'espace d'adressage virtuel fait l'objet d'une allocation dans la MVP. La fonction renvoie la structure de données du *chunk* positionné en tête de liste. Si les données de la chaîne sont allouées dans l'espace mémoire du processus appelant, la structure pointe vers cet espace. Sinon, un nouvel espace est alloué dans la mémoire du processus appelant.

`SAT_LOOKUP` fait une allocation séparée d'un *chunk* dans l'espace mémoire du processus appelant sans faire d'allocation dans la MVP. Si le *chunk* n'existe pas dans l'espace d'adressage virtuel, un pointeur **NULL** est renvoyé.

`SAT_LOOKUP_LST` réalise la même fonction que `SAT_LOOKUP` pour une liste d'identifiants de *chunks*. La fonction renvoie le *chunk* positionnée en tête de la liste.

4.2 Interface de l'API sur la logique programmable du FPGA

L'objectif de l'intégration d'accélérateurs reconfigurables dans la MVP est de permettre d'accélérer des applications en portant leurs noyaux de calcul sur les accélérateurs et en donnant la capacité aux noyaux d'accéder aux données partagées. Avec le système SAT, les processus clients accèdent aux données partagées en émettant des requêtes qui sont traitées par des processus serveurs. Pour permettre aux noyaux de calcul d'accéder aux données partagées, il est nécessaire de créer une interface qui leur permet d'émettre les requêtes et qui réalise les services demandés.

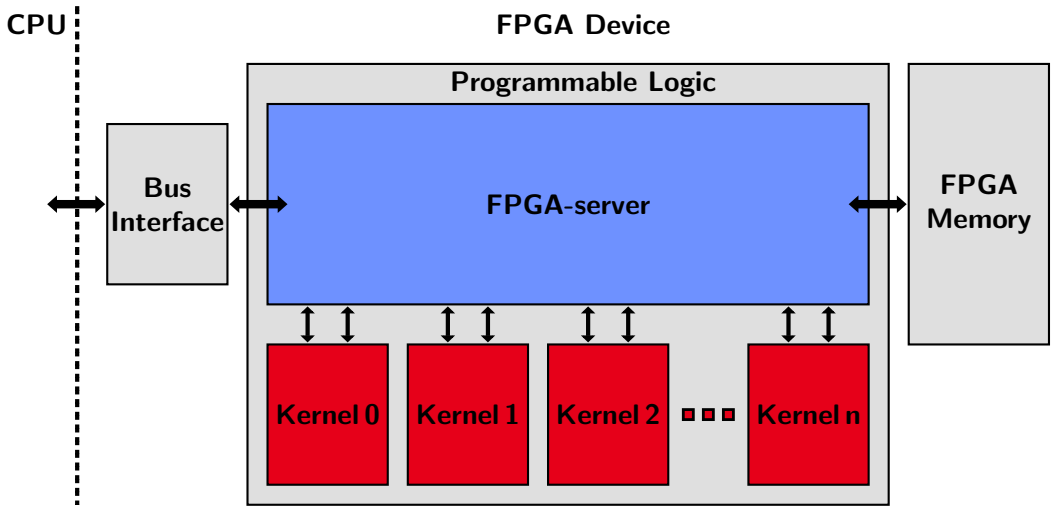
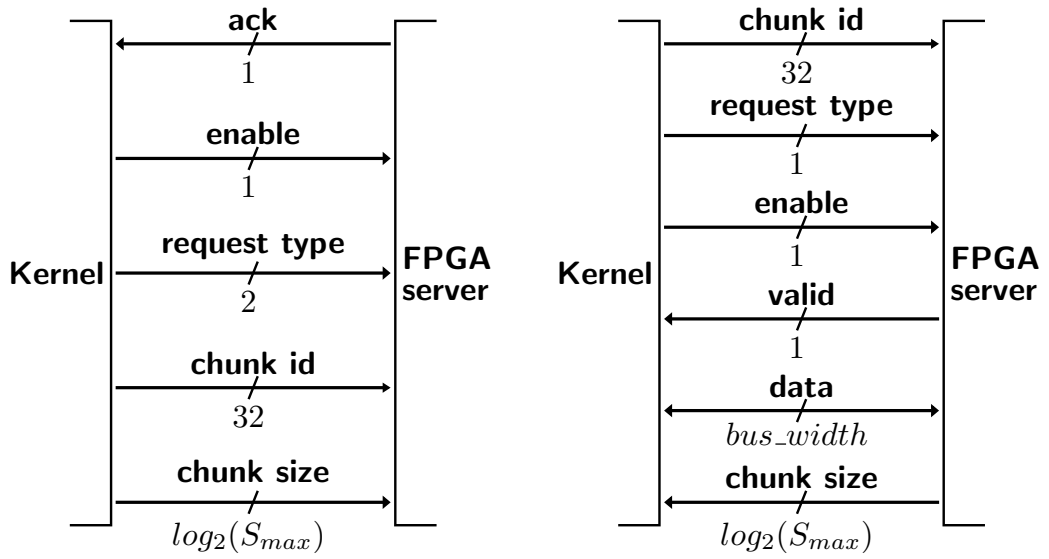


FIGURE 4.3 – Représentation logique de l'interface de l'API sur un FPGA. Les boîtes rouges représentent les noyaux de calcul du code utilisateur. La boîte bleue représente le module d'interface de l'API. Les boîtes grises symbolisent les ressources matérielles de la carte FPGA. Le module *FPGA-server* communique avec l'environnement logiciel de la MVP exécuté sur un CPU par un système d'interconnexion.

La figure 4.3 représente l'interface entre des noyaux de calcul instanciés sur un FPGA et l'API. Cette interface correspond à un module appelé *FPGA-server* et qui est implémenté sur la logique programmable de l'accélérateur. Ce module est en charge de reproduire le rôle d'un serveur pour les noyaux de calcul. Il occupe une position centrale en interagissant avec tous les éléments du FPGA. Il perçoit les requêtes provenant des noyaux de calcul, il communique avec l'environnement logiciel de la MVP et il orchestre les transferts de données de la mémoire locale du FPGA. Pour ce faire, il opère de manière analogue à un cache de données. La mémoire du FPGA est segmentée en emplacements de la taille maximale d'un *chunk*. Le module *FPGA-server* instancie un répertoire qui associe chaque emplacement de la mémoire à un *chunk*. Ce répertoire mémorise les métadonnées des *chunks* présents dans la mémoire. Dans le modèle d'intégration que nous proposons, le module *FPGA-server* communique avec un processus logiciel (*FPGA-client*) qui est exécuté sur un CPU. Pour établir cette communication, le module *FPGA-server* instancie un contrôleur lui permettant de piloter l'interface du système d'interconnexion reliant le FPGA et le CPU. L'implémentation de ce contrôleur dépend du type d'interface utilisée et donc du système d'interconnexion. Le type de système d'interconnexion



(a) Signaux de communication pour l'émission d'une requête d'accès à un *chunk*.

(b) Signaux de communication pour une demande de transfert de données.

FIGURE 4.4 – Signaux de communication entre un noyau de calcul et le module *FPGA-server* pour l'accès aux données d'un *chunk*. S_{max} désigne la taille maximale d'un *chunk*. bus_width désigne la largeur du bus de données interne au FPGA.

pouvant être employé est présenté dans la section 4.4. Pour manipuler la mémoire du FPGA, le module *FPGA-server* instancie un contrôleur mémoire qui permet de faire l'interface entre la logique programmable de l'accélérateur et la couche physique du module mémoire qui est externe à la logique programmable. L'implémentation de ce contrôleur mémoire peut être en partie réalisée à l'aide de solution proposée par les vendeurs de cartes FPGA (par exemple, le Memory Interface Generator (MIG) fourni dans l'environnement de développement Vivado de *Xilinx*). Enfin, le module *FPGA-server* instancie un bus de communication *Advanced eXtensible Interface* (AXI) pour transférer les données entre le contrôleur mémoire et les autres composants (les noyaux de calculs et l'interface du système d'interconnexion).

Pour un noyau de calcul, l'accès aux données d'un *chunk* correspond à un schéma de communication avec le module *FPGA-server* qui se divise en deux étapes : la requête d'accès au *chunk* et la requête de transfert des données.

L'émission d'une requête d'accès correspond à une instruction d'acquisition d'un *chunk* qui se fait au moyen d'un protocole d'interface dit *handshake*. La

figure 4.4a représente les signaux utilisés pour établir cette interface. Le noyau de calcul communique l'identifiant du *chunk*, le type de la requête d'accès parmi les trois instructions d'acquisition (`read`, `readwrite` ou `write`) et met le signal `enable` à la valeur booléenne 1 pour indiquer qu'une nouvelle requête est émise. Si la requête est de type `write` le noyau indique également la taille du bloc de données. Le module *FPGA-server* indique que la requête a été acceptée en faisant passer le signal `ack` à la valeur booléenne 1. La figure 4.5 décrit les conditions pour qu'une requête soit acceptée par le module *FPGA-server* :

- si le *chunk* est déjà présent dans le répertoire et qu'il n'est pas dans l'état `Exclusive`, la requête est acceptée et les métadonnées sont mises à jour en fonction de l'état du *chunk* (`Modified`, `Shared` ou `Invalid`) et du type de la requête. Lorsque la requête est une lecture et que le *chunk* est dans l'état `Shared`, le nombre de lecteurs est incrémenté, sinon le module *FPGA-server* transmet la requête d'accès à la MVP logicielle pour changer l'état du *chunk* et mettre à jour les données.
- si le *chunk* n'est pas présent dans le répertoire et qu'une entrée est disponible, celle-ci est initialisée pour accueillir les métadonnées et la requête est transmise à la MVP logicielle pour être complétée. Dans ce cas, la demande est acceptée par le module *FPGA-server*.
- si le *chunk* n'est pas présent dans le répertoire et qu'aucune entrée n'est disponible ou que le *chunk* est présent dans l'état `Exclusive`, la demande n'est pas acceptée et l'interface du noyau qui a fait la requête est bloquée jusqu'à la libération d'une entrée.

Une requête de transfert de données peut être émise seulement après que la requête d'accès au *chunk* correspondant ait été acceptée. L'émission d'une requête de transfert utilise également un protocole d'interface *handshake* dont les signaux sont représentés sur la figure 4.4b. Le noyau de calcul communique l'identifiant du *chunk* et le type de requête : une lecture pour un transfert depuis la mémoire du FPGA où une écriture pour un transfert vers la mémoire. Il émet la requête en faisant passer le signal `enable` à la valeur booléenne 1. Lorsque le module *FPGA-server* est prêt à transférer les données, il met le signal `valid` à la valeur booléenne 1 et maintient cette valeur jusqu'à la fin de la transmission. Si le transfert est une lecture, il indique la taille du bloc de données avec l'ensemble de signaux `chunk size` et transfert les données avec l'ensemble de signaux `data`. Si la requête est une écriture, c'est le

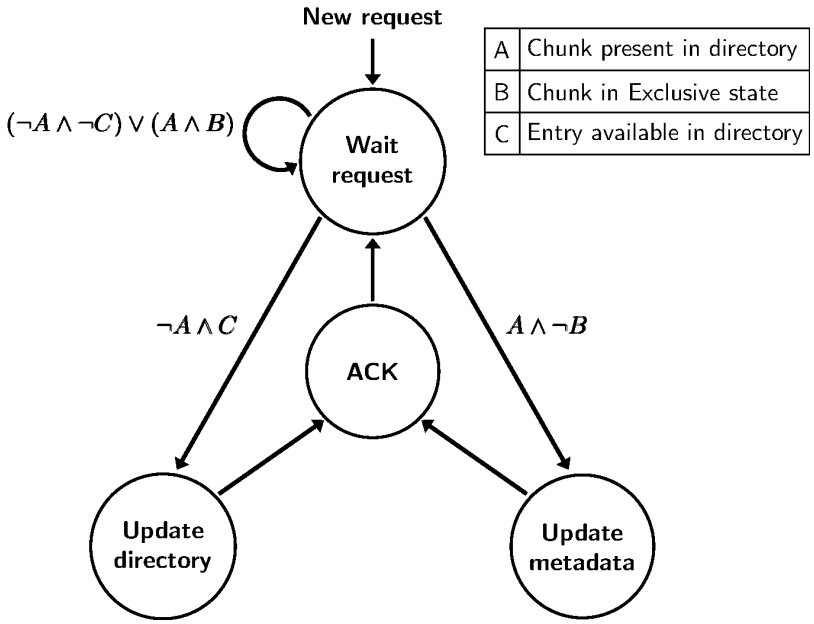


FIGURE 4.5 – Représentation sous la forme d'un automate fini de l'acceptation d'une requête d'accès à un *chunk* par le module *FPGA-server*.

noyau de calcul qui transfère les données toujours en utilisant l'ensemble de signaux `data`. Pour que le module *FPGA-server* soit prêt à effectuer un transfert de données, il faut que le *chunk* soit dans un état valide, c'est-à-dire que la requête d'accès soit complétée, et que le contrôleur mémoire du FPGA soit disponible. Pour ne pas créer de famine de données dans les noyaux, le module *FPGA-server* partage la disponibilité du contrôleur mémoire en utilisant l'algorithme du tourniquet (*Round Robin*). L'instruction de libération du *chunk* est implicitement effectuée lorsque le transfert des données est complété. Pour libérer une entrée du répertoire, le module *FPGA-server* utilise par défaut un algorithme de remplacement *Least Recently Used* (LRU) afin de profiter de la localité temporelle des données. Le remplacement d'une entrée est possible lorsqu'un *chunk* accédé en lecture à un nombre de lecteur égal à 0 ou lorsque les données d'un *chunk* accédé en écriture ont été répliquées par le processus *FPGA-client*.

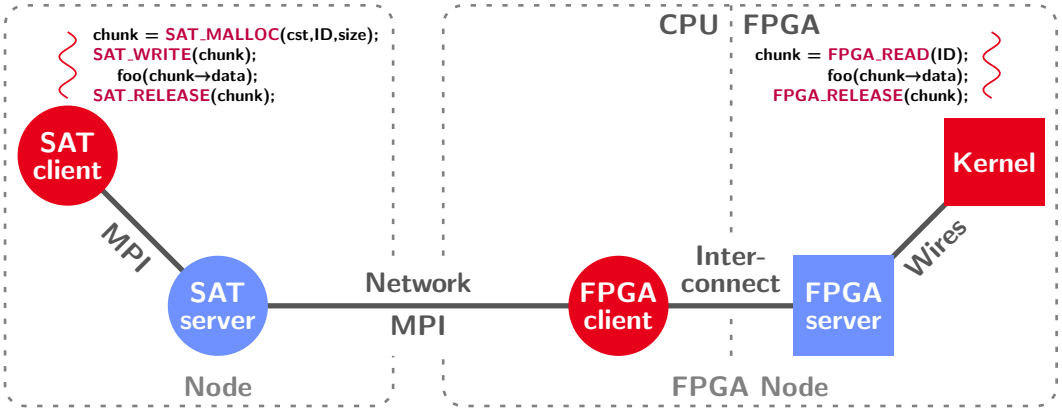


FIGURE 4.6 – Représentation de la topologie du système de proxy pour l'intégration d'un FPGA dans la MVP. Les cercles représentent des processus logiciels de la MVP exécutés par des CPU. Les boîtes représentent les modules implémentés sur la logique programmable du FPGA.

4.3 Mandataire logiciel du FPGA pour l'API

Du point de vue applicatif, un accélérateur a pour rôle de traiter des portions du code utilisateur. Pour représenter un FPGA auprès de la MVP, un processus logiciel, appelé *FPGA-client* est utilisé comme mandataire. Ce dernier est un client du système SAT, ce qui permet de faire apparaître le FPGA comme un processus client pour l'ensemble des processus de la MVP. L'intérêt de cette approche est d'offrir une abstraction de la plateforme matérielle où les données sont accédées par des processus clients, quelle que soit la nature du matériel, depuis un espace d'adressage unifié qui est géré par des processus serveurs.

La figure 4.6 illustre la topologie du modèle d'intégration d'un FPGA dans la MVP. Le module *FPGA-server* est interfacé avec le processus *FPGA-client* par les moyens d'un système d'interconnexion présenté dans la section 4.4. Le processus *FPGA-client* étant un client du système SAT, il est relié à un unique processus serveur. Il a pour rôle de collecter les requêtes provenant du FPGA et de réaliser le service demandé. Ces requêtes correspondent aux instructions d'acquisitions pour l'accès aux *chunks* ou à une instruction de libération. La communication entre les deux composants se fait par passage de messages. De manière commune à beaucoup de protocoles de communication, un message se constitue d'un en-tête suivi d'un paquet de données. L'en-tête

renseigne le type de requête émis et les métadonnées du *chunk*. Le paquet de données contient le bloc de données du *chunk*.

Le traitement réalisé par le processus *FPGA-client* consiste à demander l'allocation du *chunk*, à émettre la requête correspondant à l'instruction reçue et à envoyer un message d'acquittement au FPGA. Pour limiter le nombre de demandes d'allocation, le processus instancie un répertoire contenant les entrées du répertoire du module *FPGA-server*. Lorsque le processus *FPGA-client* reçoit un message provenant du FPGA, il décode l'en-tête pour adapter le traitement du message en fonction du *chunk* et du type de requête :

Instruction de lecture : cette instruction permet d'attendre que le *chunk* soit dans un état *Shared* et que les données de la mémoire du FPGA soient à jour. Deux cas de figure peuvent conduire à l'émission de cette requête :

— le *chunk* n'est pas présent dans le répertoire. Alors, le processus fait une demande d'allocation avec la fonction `SAT_LOOKUP`, puis émet une requête d'accès en lecture. Après la complétion de la requête, le processus *FPGA-client* envoie au FPGA le message d'acquittement qui contient la taille et le contenu du bloc de données du *chunk*.

— le *chunk* est présent dans le répertoire, mais son état n'est pas `Shared` ou son nombre de lecteur est nul. Alors, le processus émet uniquement une requête d'accès en lecture. Lorsque la requête est complétée, le processus envoie le message d'acquittement au FPGA. Si les données du *chunk* ne sont pas à jour dans la mémoire du FPGA, le message d'acquittement contient les données du *chunk*.

Instruction de lecture-écriture : cette instruction permet d'attente que le *chunk* soit dans un état `Exclusive` et que la mémoire du FPGA contiennent une version à jour des données. Comme pour l'instruction de lecture, deux cas de figure peuvent conduire à l'émission de cette requête :

— le *chunk* n'est pas présent dans le répertoire. Le processus réalise le même traitement que pour l'instruction de lecture à la différence du type de requête émis.

— le *chunk* est présent dans le répertoire. Alors, peu importe l'état du *chunk*, le processus émet une requête d'accès en lecture-écriture. Un message d'acquittement est envoyé au FPGA après que la requête ait été complétée. Comme pour l'instruction de lecture, en cas

de besoin, le message d'acquittement permet de mettre à jour les données de la mémoire du FPGA.

Instruction d'écriture : cette instruction permet d'attendre que le *chunk* soit dans un état `Exclusive`. Si le *chunk* n'est pas présent dans le répertoire, le processus fait une demande d'allocation avec la fonction `SAT_MALLOC` en indiquant la taille du bloc de données à allouer. Ensuite, dans tous les cas, le processus émet l'instruction d'acquisition en écriture. Lorsque la requête est satisfaite par le serveur, le processus envoie un message d'acquittement au FPGA.

Instruction de libération : le traitement est adapté en fonction de l'état du *chunk* :

Shared : le *chunk* était accédé en lecture uniquement. L'instruction de libération est émise par le module *FPGA-server* seulement lorsque le nombre de lecteur est nul. Dans ce cas, le processus *FPGA-client* émet l'instruction à son serveur pour que le *chunk* soit libéré.

Exclusive : le *chunk* était accédé en écriture ou en lecture-écriture. Le module *FPGA-server* émet la requête lorsque le transfert de données entre le noyau de calcul et la mémoire du FPGA est complété. Par défaut, une politique d'écriture immédiate (*write-through*) est adoptée. Dans ce cas, le message contient les données du *chunk*. Les données sont copiées dans l'espace mémoire du processus *FPGA-client* qui a été alloué pour le *chunk*. Après avoir effectué la copie, le processus émet l'instruction de libération pour que l'état du *chunk* passe de `Exclusive` à `Modified`.

Une instruction de libération ne nécessite pas d'envoyer de message d'acquittement au module *FPGA-server*.

Il en résulte un protocole de cohérence hiérarchique où la gestion des données est répartie entre le module *FPGA-server*, le processus *FPGA-client* et les serveurs de la MVP. L'objectif est de limiter le nombre de requêtes émises aux serveurs.

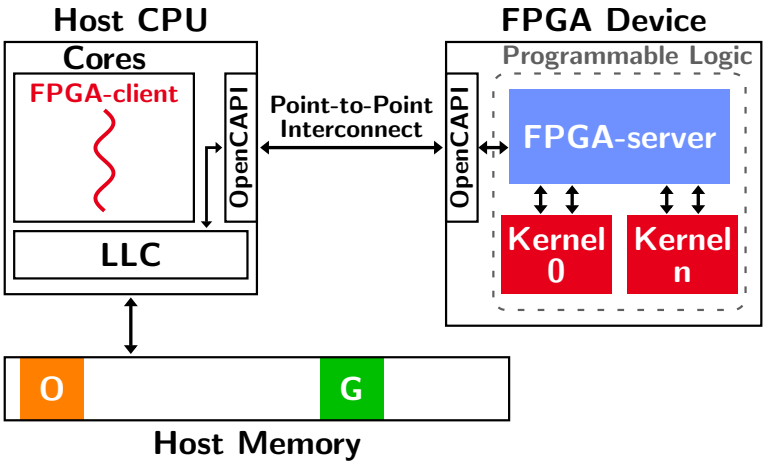


FIGURE 4.7 – Représentation de la topologie du système d'interface utilisant un bus OpenCAPI. La boîte LLC (Last Level Cache) représente le dernier niveau de cache du processeur.

4.4 Systèmes d'interconnexions entre le processeur hôte et l'accélérateur reconfigurable

Pour qu'un accélérateur reconfigurable puisse accéder aux données de la MVP, une interface doit être créée entre l'environnement matériel de l'accélérateur et l'environnement logiciel du système SAT. Cette interface est créée par le module *FPGA-server* qui est implémenté sur la logique programmable de l'accélérateur et par le processus *FPGA-client* qui est exécuté par le processeur hôte. Pour que l'interface soit opérationnelle, ces deux composants doivent communiquer par l'intermédiaire d'un système d'interconnexion qui relie le processeur hôte et l'accélérateur. Dans le cadre des travaux de cette thèse, trois solutions ont été étudiées comme système d'interconnexion : une basée sur la technologie OpenCAPI, une basée sur un bus PCI Express et une basée sur l'utilisation d'une puce FPGA embarquant un processeur.

La première solution est basée sur la technologie OpenCAPI. Cette étude a fait l'objet d'une publication à la conférence COMPAS 2019 [73]. OpenCAPI est un nouveau standard de bus soutenu par le consortium du même nom. Son objectif est de mettre en place une connexion point-à-point entre un processeur hôte et un accélérateur. OpenCAPI permet à un accélérateur d'utiliser l'espace d'adressage virtuel d'un processus pour accéder directement à la mémoire du processeur hôte. Pour ce faire, les requêtes d'accès à la mé-

moire sont traitées par le dernier niveau de cache du processeur qui utilise son unité de gestion mémoire pour traduire les adresses et ses mécanismes matériels pour maintenir la cohérence des données entre le processus et l'accélérateur. La figure 4.7 représente la topologie du système d'interface utilisant la technologie OpenCAPI. Avec ce système, le module *FPGA-server* transmet les requêtes d'accès à la MVP par l'intermédiaire du bus et du dernier niveau de cache. Lorsque le processus *FPGA-client* reçoit une requête, le traitement a pour effet d'allouer le bloc de données du *chunk* dans son espace mémoire. Ensuite, le processus envoie un message à l'accélérateur pour transmettre les métadonnées qui indiquent l'adresse à laquelle se trouve le bloc de données. Par la suite, l'accélérateur peut accéder directement aux données grâce à l'interface OpenCAPI. Cette solution présente l'avantage majeur de ne pas devoir répliquer les données entre la mémoire du processeur hôte et la mémoire du FPGA. Elle est envisageable parce que le bus permet à l'accélérateur d'accéder à la mémoire du processeur avec une bande passante élevée et une latence relativement faible. Le consortium OpenCAPI annonce avoir mesuré une bande passante mémoire de 22 Go/s et une latence de 378 ns entre un processeur IBM Power 9 et un FPGA Xilinx VU3P [96].

Cependant, cette solution a plusieurs limites. Premièrement, OpenCAPI n'est pas une technologie mature et son développement ne semble pas avoir beaucoup évolué depuis le début des travaux de cette thèse commencée en 2018. Deuxièmement, l'utilisation d'un bus OpenCAPI nécessite que l'accélérateur et le processeur intègre une interface dédiée. Actuellement, seuls les processeurs IBM Power 9 intègrent ce type d'interface ce qui rend l'accès à cette technologie limité. Enfin, avec cette technologie le dernier niveau de cache du processeur devient une ressource partagée entre les processus exécutés sur le processeur et l'accélérateur. Ce partage de la ressource peut provoquer des conflits conduisant à une chute des performances à la fois pour les processus et pour l'accélérateur.

Il existe d'autres technologies de système d'interconnexion proposant des services similaires à ceux de OpenCAPI, telles que QPI (*QuickPath Interconnect*) de Intel et le bus *HyperTransport* qui sont deux technologies matures ou Gen-Z et CCIX (*Cache Coherent Interconnect for Accelerators*). Ces deux derniers sont des nouveaux standards de bus proposés par des consortiums, mais qui ne sont pas encore matures ou qui sont intégrés dans des "composants sur étagère" (*Components-Off-the Shelf* (COTS)).

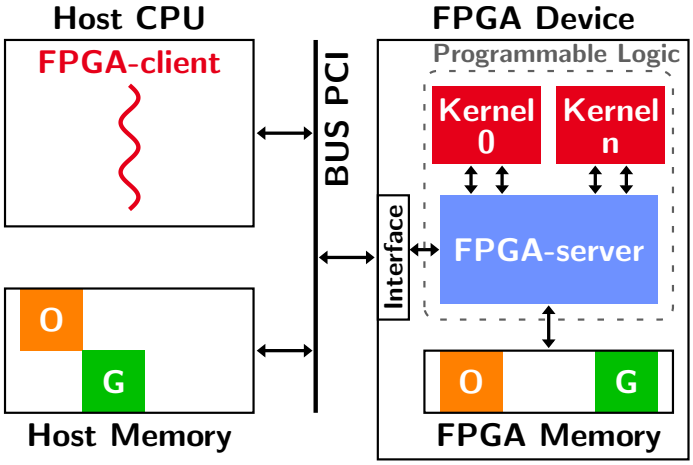


FIGURE 4.8 – Représentation de la topologie du système d'interface utilisant un bus PCI Express.

La deuxième solution est basée sur l'interconnexion de l'accélérateur et du processeur par un bus PCI Express. La figure 4.8 représente la topologie d'interface utilisant cette solution. Elle repose sur une approche logicielle et matérielle, où le FPGA instancie une IP DMA lui permettant d'accéder directement à la mémoire du processeur hôte et où le processeur utilise un pilote pour transférer des paquets de données au FPGA. Les IP DMA et les pilotes peuvent être proposés sous forme de produits propriétaires par les vendeurs de FPGA. De cette manière, des tampons mémoires sont alloués dans la mémoire du processeur hôte pour que le module *FPGA-server* communique ses messages en initiant des transferts DMA. Le processus *FPGA-client* traite les requêtes à la réception des messages, ce qui a pour effet d'allouer les blocs de données des *chunks* dans son espace mémoire. Ensuite, le processus utilise le pilote pour renvoyer le message d'acquittement au FPGA. L'avantage majeur de cette solution est que les bus PCI Express sont une technologie mature largement disponible et largement employée. En effet, la majorité des cartes FPGA utilisées comme accélérateurs intègre une interface PCI Express et la majorité des cartes mères utilisées dans les systèmes de calcul dispose d'un ou plusieurs emplacements pour connecter une interface PCI Express. Cependant, cette solution présente également deux limites majeures. Premièrement, elle nécessite de répliquer les données des *chunks* dans la mémoire du FPGA. Deuxièmement, l'utilisation d'un bus PCI Express pour relier un FPGA et un CPU implique une bande passante mémoire limitée et des latences élevées [26]. Cette limitation est la raison de l'émergence de plusieurs consortiums, cités

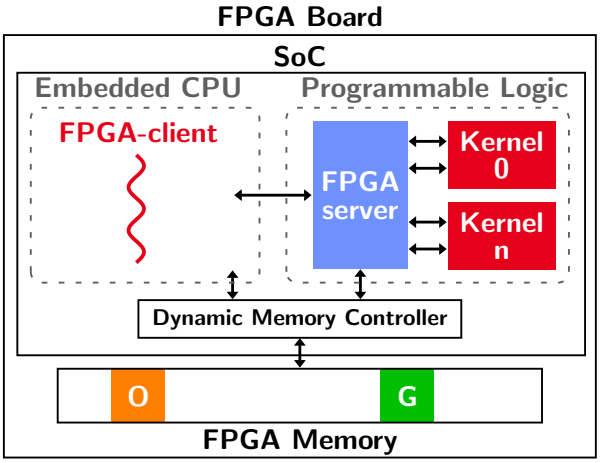


FIGURE 4.9 – Représentation de la topologie d'interface basée sur l'utilisation d'un système sur puce FPGA embarquant un CPU.

précédemment, dont l'objectif est de proposer un nouveau standard de bus d'interconnexion permettant à la fois d'augmenter la bande passante et les latences d'accès à la mémoire.

Tel qu'illustré sur la figure 4.9, la troisième solution est basée sur l'utilisation d'une carte FPGA qui intègre un système sur puce hétérogène embarquant de la logique programmable et un processeur. Ce type de carte est notamment proposé par Intel avec la gamme *Agilex* et Xilinx avec la gamme Zynq. Pour les plus récentes cartes de ces deux gammes, l'architecture du processeur est la même : un Arm Cortex-A53 à 4 cœurs. Pour cette solution, le module *FPGA-server* communique avec le processeur par l'intermédiaire d'un port dédié (AXI ou *Accelerator Coherency Port* (ACP)). La logique programmable et le processeur partagent le même module mémoire, dont la gestion des accès est centralisée par un unique contrôleur mémoire. Cependant, la logique programmable et le processeur peuvent utiliser des ports différents pour être interfacés au contrôleur mémoire. Le processus *FPGA-client* est exécuté sur le processeur embarqué. Lorsqu'il reçoit un message provenant du module *FPGA-server*, il traite la requête, ce qui a pour conséquence d'allouer le bloc de données du *chunk* dans la mémoire du FPGA. Ensuite, le processus envoie le message d'acquittement et indique l'emplacement mémoire des données. Le *FPGA-server* peut alors orchestrer le transfert de données entre la mémoire et les noyaux de calcul qui ont émis la requête.

Comme pour la solution utilisant la technologie OpenCAPI, cette solution présente l'avantage de ne pas répliquer les données. Cette solution doit per-

mettre également de réduire la consommation énergétique de l'ensemble du système pour deux raisons. Premièrement, ces cartes FPGA sont optimisées pour avoir une consommation énergétique faible. Le processeur embarqué dans ces cartes est de petite dimension et est beaucoup moins énergivore que les processeurs intégrés dans les serveurs de calcul HPC. Néanmoins, l'architecture des processeurs embarqués reste bien adaptée aux besoins du processus *FPGA-client*, dont le traitement est simple [33]. Deuxièmement, le processeur hôte et le module *FPGA-server* sont contenus dans la même puce, ce qui permet de limiter les mouvements de données hors puce. En effet, un réseau sur puce est utilisé pour relier la logique programmable et le processeur. Ce réseau permet de réaliser des communications moins énergivores et à plus faible latence qu'un bus d'interconnexion hors puce. Cependant, l'intégration physique d'un processeur sur ces systèmes sur puces hétérogènes se fait au détriment de l'intégration de la logique programmable. De ce fait, pour une même génération de FPGA, ces systèmes disposent d'une logique programmable de plus petite dimension qu'un système n'embarquant pas de processeur. Ainsi, cette solution présente comme limite majeure une réduction de la puissance de calcul permise par la logique programmable.

4.5 Inconvénients et propositions d'améliorations

Deux perspectives ont été identifiées pour faire évoluer les travaux présentés dans ce chapitre. Premièrement, en plus de construire une mémoire virtuellement partagée, le système SAT offre un support pour la programmation évènementielle [32]. Ce support repose sur un mécanisme appelé *Publish-Subscribe* où des processus clients peuvent souscrire à un *chunk* et d'autres processus peuvent notifier qu'ils ont modifié le *chunk*. Les processus souscripteurs peuvent appeler une fonction spéciale lorsqu'ils reçoivent une notification. Ce support de programmation peut être utile dans le cadre de l'utilisation d'un FPGA. En étant émetteur de notifications, le FPGA peut permettre à un processus de surveiller son activité. En étant souscripteur, le FPGA peut réduire sa consommation énergétique. Les FPGA ont la capacité d'utiliser plusieurs signaux d'horloges opérant à des fréquences différentes et pouvant être reconfigurées dynamiquement. De cette manière, un FPGA pourrait réduire la fréquence d'horloge d'un noyau de calcul dans l'attente d'une notification.

Cette réduction de la fréquence d'horloge doit permettre une réduction de la consommation énergétique.

La deuxième perspective concerne l'utilisation des mémoires à haute bande passante (HBM) intégrées sur les cartes FPGA de nouvelles générations. Les mémoires HBM intègrent par superposition plusieurs puces de mémoire vive dynamique (*Dynamic Random Access Memory* (DRAM)) dans un même module mémoire. Cette superposition offre plusieurs chemins de données permettant d'augmenter le débit des transferts mémoires et ainsi offre au FPGA une plus grande capacité de vitesse de traitement pour des applications utilisant un grand débit de données. L'utilisation d'une mémoire HBM nécessite d'instancier un contrôleur mémoire pour chaque chemin de données. Le modèle d'intégration d'accélérateurs reconfigurables dans une MVP que nous proposons considère des FPGA embarquant des modules mémoires de type *Double Data Rate* (DDR) possédant un seul chemin de données. La perspective d'évolution est donc de prendre en compte l'architecture de ces mémoires où les données sont distribuées dans plusieurs puces et dont l'accès est non uniforme. L'utilisation la plus évidente de ces mémoires serait qu'un *chunk* soit alloué dans une même puce accessible par un unique chemin de données et que plusieurs *chunks* puissent être accédés en parallèle en utilisant l'ensemble des chemins de données. Pour ce faire, le module *FPGA-server* devrait segmenter son répertoire de *chunks* et instancier un contrôleur mémoire pour chaque chemin de données.

Le modèle d'intégration d'accélérateurs reconfigurables dans une mémoire virtuellement partagée exposé dans ce chapitre présente deux inconvénients majeurs. Le premier est le coût de l'interface en termes d'utilisation des ressources du FPGA et de latence d'accès aux données distantes. L'implémentation du module *FPGA-server* requiert l'utilisation de ressources du *FPGA* pour instancier le contrôleur mémoire, le contrôleur pour l'interface du bus externe, le bus de données interne, le répertoire de *chunks*, les interfaces avec les noyaux de calcul et les automates qui contrôlent le fonctionnement du module. Ces ressources sont également nécessaires pour l'implémentation des noyaux de calcul. Ainsi, l'instanciation du module *FPGA-server* empiète sur la capacité d'intégration des noyaux de calcul. Cependant, l'utilisation d'un FPGA comme coprocesseur requiert forcément l'instanciation d'IP permettant d'échanger des données et de contrôler le FPGA. Pour ce type d'utilisation, il est commun d'instancier un contrôleur mémoire, un contrôleur pour l'interface

du bus externe, un bus de données interne et des automates pour contrôler le fonctionnement du FPGA. Ainsi, seule l'instanciation du répertoire de *chunks* apparaît comme un surcoût évident, notamment en termes d'utilisation d'éléments mémorisant pour contenir les métadonnées. Le parti pris de ces travaux est de considérer qu'une MVP apporte un confort de programmation qui se paye au prix d'un coût logiciel, pour l'exécution des processus serveurs, et matériel pour l'instanciation de l'interface de l'API. Une requête d'accès à un *chunk* qui n'est pas présent dans le répertoire du FPGA doit transiter du module *FPGA-server* jusqu'au processus serveur qui gère le *chunk* et la réponse à cette requête doit réaliser le trajet inverse. Il apparait évident que la complétion d'une requête d'accès à un *chunk* peut induire une grande latence. Ce temps long d'accès aux données est en opposition avec l'idée d'accélérer le traitement de ces mêmes données. Pour répondre à ce problème, les travaux de cette thèse ont permis de proposer un modèle de programmation présenté dans le chapitre 5.

Le deuxième inconvénient est lié à la politique d'écriture immédiate (*write-through*) adoptée par défaut. Lorsqu'un processus client souhaite accéder à un *chunk* qui a été modifié, le processus serveur orchestre un transfert à partir de l'espace mémoire d'un processus qui en possède une version à jour. Avec la solution d'interconnexion utilisant un bus PCI, les processus serveurs ne connaissent pas la localisation des données dans la mémoire du FPGA et n'ont pas la capacité d'accéder à cette mémoire. Pour cette raison, il est nécessaire que les données modifiées dans la mémoire du FPGA soient propagées à la mémoire du processus *FPGA-client*. Ces transferts ont un coût et peuvent être inutiles dans le cas où seulement un FPGA réutilise les données. Pour adopter une politique d'écriture différée (*write-back*), il faudrait modifier le modèle d'intégration et le système SAT de sorte à donner aux processus serveurs l'information de l'existence d'un coprocesseur. Cette modification pourrait être d'autoriser les processus serveurs d'interagir directement avec le module *FPGA-server* ou de modifier le protocole de cohérence pour que le processus *FPGA-client* devienne acteur de la mise à jour.

Chapitre 5

Modèle de programmation pour noyaux de calcul irréguliers sur accélérateur reconfigurable dans un système distribué hétérogène

Un modèle de programmation est une abstraction d'un système informatique qui permet d'exprimer à la fois les algorithmes et les structures de données qui forment les programmes. Un modèle de programmation permet de lier l'architecture matérielle d'un système informatique et les couches logicielles sous-jacentes aux applications. Généralement, la définition d'un modèle de programmation a pour objectif d'améliorer les performances des applications et d'en faciliter le développement. La bonne conception d'un modèle de programmation repose sur la capacité de mettre en adéquation les besoins applicatifs avec les caractéristiques de l'architecture matérielle, du système d'exploitation et du support d'exécution [102].

Tel qu'exposé dans la première partie de ce manuscrit, l'évolution des besoins applicatifs a conduit à la construction de systèmes de calcul massivement distribués et hétérogènes pour satisfaire une demande croissante de puissance de calcul. Dans le même temps, ces systèmes sont plus complexes à programmer, notamment en ce qui concerne la gestion des données. La complexité de la gestion des données provient, d'une part, de la répartition physique des mémoires et, d'autre part, de l'hétérogénéité de la hiérarchie mémoire induite par celle des ressources de calcul. Un système de calcul hétérogène intégrant

des coprocesseurs permet d'exécuter sur les coprocesseurs les portions d'un programme dont l'intensité arithmétique est élevée. Dans ce manuscrit, ces portions sont désignées comme des noyaux de calcul. Le développement d'un programme se fait au moyen d'un ensemble de langages de programmation et éventuellement d'un ensemble d'API. Les langages de programmation et les API reposent sur un modèle de programmation dont le rôle est d'abstraire le fonctionnement du système de calcul. Le modèle de programmation standard d'un système hétérogène repose sur le modèle maître-esclave. Un processeur, ayant la maîtrise du système, décharge une partie de sa charge de travail en portant les noyaux de calcul sur un coprocesseur. Avant de lancer l'exécution d'un noyau de calcul, le processeur orchestre le transfert des données à traiter vers la mémoire locale du coprocesseur. De ce fait, le coprocesseur n'a pas la maîtrise des données auxquelles il accède. Pour le portage de noyaux de calcul traitant des structures de données régulières, l'orchestration des transferts de données est relativement simple à entreprendre. Au contraire, pour le traitement de structures de données irrégulières, provoquant des accès mémoires aléatoires et dépendant des données, l'orchestration des transferts de données peut se révéler complexe.

Dans les disciplines des sciences et de l'ingénierie, l'utilisation de simulations numériques est devenue nécessaire pour résoudre des problèmes, par exemple : pour étudier la déformation d'un matériau soumis à une charge, pour simuler l'écoulement d'un fluide ou pour réaliser une simulation cosmologique. Les problèmes modélisés devenant de plus en plus larges et leur résolution devenant de plus en plus fine, le coût des applications, en termes de charge de calcul et d'empreinte mémoire, a explosé. En utilisant des algorithmes naïfs, ce coût rend la résolution des problèmes impossible, même en utilisant les systèmes de calcul les plus puissants. Heureusement, les progrès des méthodes de simulations numériques permettent de modéliser les problèmes en utilisant une précision variable dans le temps ou dans l'espace en fonction des besoins applicatifs. Cependant, cette adaptation conduit à modéliser les problèmes de manière irrégulière et rend le développement des applications de calcul plus complexe. Ces applications nécessitent le recours à des structures de données permettant de décrire l'irrégularité du problème modélisé, tels que des matrices creuses stockées avec des formats compressés, des maillages non structurés, des graphes ou des arbres. Le traitement de ces structures de données génère des motifs d'accès irréguliers à la mémoire, qui sont dépendants des données. Dans

ce manuscrit, nous utilisons le terme de noyau de calcul irrégulier pour désigner un noyau de calcul qui traite ce type de structure et par conséquent qui génère des accès mémoires aléatoires et dépendants des données.

Le chapitre 4 fait la présentation d'un intergiciel permettant d'intégrer des accélérateurs reconfigurables dans une MVP. Ce système donne la capacité aux noyaux de calcul d'être initiateurs de l'accès aux données qu'ils traitent. L'objectif est de faciliter la gestion des données réparties pour la programmation de noyaux de calcul irréguliers. La proposition de ce système invite à réfléchir à la manière dont un coprocesseur doit être programmé lorsqu'il peut initier l'accès à des données distantes. Dans le cadre des travaux de cette thèse, nous proposons d'effectuer cette réflexion et de la structurer en proposant un modèle de programmation.

Ce chapitre est organisé de la façon suivante : la section 5.1 expose le principe général du modèle de programmation proposé, la section 5.2 illustre l'utilisation du modèle de programmation pour des applications traitant des matrices creuses, la section 5.3 illustre l'utilisation du modèle de programmation pour traiter des maillages non structurés par méthode des éléments finis, enfin la section 5.4 conclut ce chapitre.

5.1 Principe général du modèle de programmation

Une mémoire partagée est un paradigme de programmation pratique pour développer des applications parallélisées qui accèdent de manière aléatoire aux données. Dans le cadre des travaux de cette thèse, pour faciliter la programmation de noyaux de calcul irrégulier sur des accélérateurs reconfigurables, nous proposons un modèle de programmation qui repose sur une mémoire virtuellement partagée logicielle permettant aux noyaux de calcul d'initier les accès aux données réparties, tel que présenté dans le chapitre 4. L'intérêt est, d'une part, d'abstraire l'architecture d'un système de calcul distribué en offrant aux développeurs une vision unifiée des mémoires physiquement réparties et, d'autre part, d'initier l'accès aux données à l'endroit où leurs motifs sont générés, c'est-à-dire par les noyaux de calcul. L'objectif de ce modèle de programmation est d'amener les bonnes données au bon endroit facilement.

Cependant, tel qu'exposé dans la section 4.5, l'utilisation d'une MVP lo-

gicielle peut induire une latence d'accès aux données élevée parce que la répartition des données est abstraite et l'accès à des données distantes induit un temps de transfert incompressible. Pour un noyau de calcul instancié sur un accélérateur reconfigurable, ces latences sont encore plus grandes à cause du modèle d'intégration présenté dans le chapitre 4 qui augmente la distance du trajet des données. Un accès lent aux données s'oppose à l'idée d'en accélérer le traitement. Pour résoudre ce problème, le modèle de programmation doit permettre de masquer les latences d'accès aux données. Le masquage des latences de transfert des données est une technique courante et nécessaire lors de l'utilisation d'un système distribué. Pour une exécution où le traitement est réparti sur plusieurs ressources, le principe du masquage des latences du transfert des données est de découper les données d'une application en n blocs et de superposer temporellement le transfert du bloc $i+1$ pendant le traitement du bloc i par une ressource. L'objectif est que, lorsque la ressource termine le traitement du bloc i, elle ait un accès immédiat aux données du bloc $i+1$. Autrement dit, le masquage des latences du transfert des données repose sur la superposition du flux de données transféré et du flux de données traité.

Avec le modèle de programmation proposé, l'accès aux données se fait par l'émission de requêtes d'accès à des *chunks*. Un *chunk* est l'unité atomique de mémoire de la MVP et correspond à un bloc de données d'une taille variable. Pour répondre à l'objectif du modèle de programmation, qui est d'amener les bonnes données au bon endroit, les structures de données doivent être partitionnées en blocs correspondant à la granularité d'accès aux données. Pour une structure de données régulières, ce partitionnement est aisé parce que les motifs d'accès aux données sont explicites. Pour une structure de données irrégulières, le partitionnement en *chunks* ne semble pas trivial, parce que l'accès aux données est aléatoire. De plus, le mécanisme d'accès aux données d'un CPU est différent de celui d'un FPGA. Un CPU exécute un flux d'instructions. Pour un noyau de calcul, le flux d'instructions est composé principalement d'instructions d'accès à la mémoire et d'instructions arithmétiques et logiques. Si on dissocie les instructions mémoires des instructions arithmétiques et logiques, on obtient un flux d'instructions qui génère un flux de données transféré entre les différents niveaux de la hiérarchie mémoire de l'ordinateur. Ces transferts se font à la granularité d'un registre pour un CPU, à la granularité d'une ligne pour le cache de données du processeur et à la granularité d'une page entre la mémoire principale de l'ordinateur et le système de stockage de données.

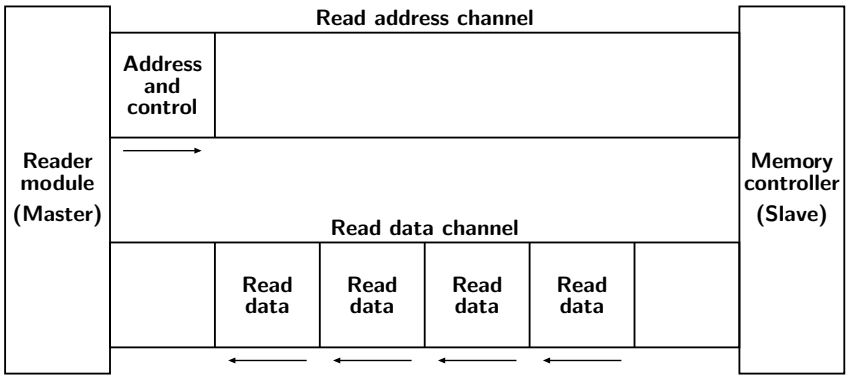


FIGURE 5.1 – Représentation d'une transaction de données en lecture réalisée en rafale avec un bus AXI entre un module initiateur et le contrôleur mémoire. La figure est reproduite à partir du guide de référence AXI [114].

L'architecture des caches de données des processeurs favorise les accès aléatoires en stockant plusieurs lignes de caches permettant au CPU d'avoir un accès rapide aux données. Un noyau de calcul instancié sur un FPGA accède et traite les données explicitement sous la forme de flux. L'architecture des FPGA favorise les accès séquentiels aux données sous la forme de blocs. Tel que représenté sur la figure 5.1, les accès aux données stockées dans le module mémoire d'un FPGA se fait à l'initiative d'un module instancié sur la logique programmable par l'intermédiaire d'un bus de données (typiquement un bus AXI). Le chemin de données du bus peut être large (pour un bus AXI de 32 à 1024 bits) et les accès aux données peuvent être faits en rafales, c'est-à-dire qu'une seule requête d'accès permet de transférer des données sur plusieurs cycles. Ce mécanisme de transfert peut être comparé à celui de la mise à jour d'une page dans la mémoire principale d'un ordinateur. Cependant, la logique programmable ne dispose pas nativement d'un cache de données tel que celui d'un processeur. Par conséquent, l'architecture d'un FPGA n'est pas favorable aux accès aléatoires aux données.

Pour expliquer comment les structures de données irrégulières doivent être partitionnées, il est nécessaire de détailler la manière dont les noyaux de calculs accèdent aux données sous forme de *chunks*. Pour un noyau de calcul instancié sur un FPGA, l'accès aux données d'un *chunk* se fait en deux étapes initié par le noyau de calcul, tel que présenté dans la section 4.2. La première étape consiste à demander l'accès au *chunk*, ce qui a pour effet d'allouer un espace dans la mémoire du FPGA servant à stocker les données du *chunk*. La deuxième étape consiste à demander le transfert des données entre le noyau

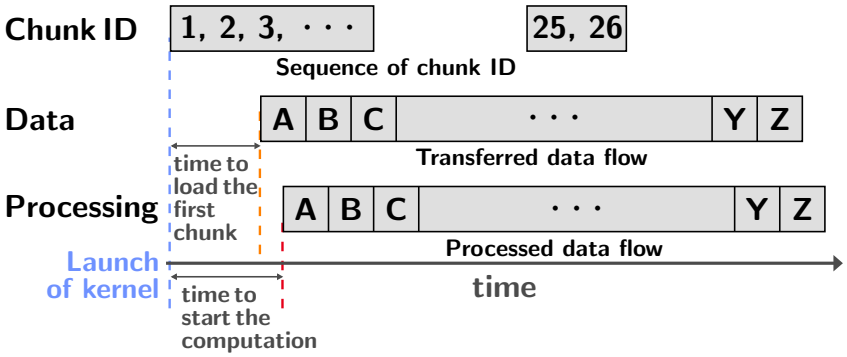


FIGURE 5.2 – Représentation de la superposition temporelle d'un flux de données traité et d'un flux de données transférées, dont l'accès est initié à partir d'une séquence d'identifiants de *chunks*.

de calcul et la mémoire du FPGA. La dissociation des deux étapes peut permettre de précharger les données de sorte à masquer les latences d'accès aux données. Tel que représenté sur la figure 5.2, en émettant les requêtes d'accès aux *chunks* par séquences, il est possible de transférer les données sous la forme d'un flux continu de données qui sera traité par le noyau de calcul. Si le préchargement est efficace, le flux de données transféré est en majorité superposé avec le flux de données traité. Seul le temps du chargement des premières données nécessaires pour initier le flux de traitement ne peut pas être masqué. Deux critères doivent être respectés pour que le préchargement soit efficace. Premièrement, l'accès aux *chunks* doit être exprimé en séquences suffisamment grandes pour masquer les latences de transfert des données. Deuxièmement, L'intensité arithmétique (la charge de calcul divisée par la quantité de données transférée) doit être suffisamment importante pour que la longueur temporelle du flux de données traité puisse recouvrir celle du flux de données transféré.

Du point de vue d'un noyau de calcul instancié sur un FPGA, l'accès à une structure de données se fait sous la forme d'un flux de données spécifique. Chacun de ces flux correspond à une interface avec le module *FPGA-server* permettant d'émettre les demandes d'accès et de transferts des *chunks* contenant les données de la structure. Chacun de ces flux est associé à une profondeur de préchargement, c'est-à-dire à un nombre de *chunks* pour lesquels une demande d'accès a été émise, mais dont le transfert des données entre la mémoire du FPGA et le noyau de calcul n'a pas été effectué. La limitation de la profondeur de préchargement d'un flux de données permet de contraindre la taille de l'espace mémoire utilisé pour stocker les données préchargées. L'ob-

jectif de cette limitation est d'empêcher que l'accès à une structure de données ait pour conséquence de saturer la mémoire du FPGA.

Chaque flux de données utilisé pour accéder à une structure de données est associé à une file (*First-In, First-Out* (FIFO)) instanciée dans le noyau de calcul. Pour chaque flux, le transfert des données en lecture se fait entre la mémoire du FPGA et cette FIFO. De manière inverse, un transfert en écriture se fait entre une FIFO et la mémoire du FPGA. Pour que le transfert puisse être effectué, la FIFO doit disposer de suffisamment d'espace pour contenir l'ensemble des données d'un *chunk*. Par conséquent, la taille maximale d'un *chunk* ne doit pas excéder celle d'une FIFO. Dans les faits, sur un FPGA, une FIFO est implémentée avec une BRAM qui a une capacité mémoire de 18 k bits ou 36 k bits.

Pour revenir au partitionnement des structures de données en *chunks*, il doit permettre de masquer les latences d'accès aux données en répondant à plusieurs contraintes. La première est de limiter le nombre de partitions pour réduire le nombre de requêtes nécessaires pour parcourir une structure de données. Le traitement d'une requête d'accès à un *chunk* a un coût. Limiter le nombre de partitions permet de réduire le coût d'accès à la structure de données. La deuxième contrainte est la taille des blocs de données des *chunks* qui ne doit pas excéder celle d'une FIFO sans être trop petite. Une petite taille de bloc donnerait un grand nombre de partitions et une mauvaise exploitation du bus de données interne du FPGA. La troisième contrainte est de permettre au noyau de calcul d'exprimer l'accès à la structure de données en une séquence de *chunks*, malgré l'irrégularité des structures. Pour répondre à ces contraintes, le partitionnement s'appuie sur deux techniques :

— **le réordonnancement des données pour en améliorer la localité spatiale.** L'amélioration de la localité spatiale des données permet d'augmenter l'utilisation des données contenues dans un bloc. Pour un CPU, cela signifie d'augmenter l'utilisation des données d'une ligne de cache ou d'une page mémoire. Pour la MVP, cela signifie d'augmenter l'utilisation par accès des données contenues dans un *chunk*. Cet objectif n'étant pas propre aux travaux de cette thèse, il existe de nombreuses techniques permettant d'améliorer la localité spatiale des données pour l'accès à différents types de structures irrégulières.

— **l'utilisation des métadonnées des *chunks* pour abstraire l'irrégularité de la structure de données.** Ces métadonnées renseignent

la taille du bloc de données allouées. Les structures de données irrégulières étant notamment utilisées pour modéliser des problèmes avec une précision variable, c'est-à-dire avec une quantité de données qui varie dans l'espace, l'obtention de la taille du bloc de données alloué dans un *chunk* peut aider à abstraire la structure de données. Les métadonnées permettent également de chaîner les *chunks*, tel que présenté dans la section 4.1. Une allocation chaînée peut permettre d'exprimer une relation entre deux *chunks* ou d'allouer un bloc de données dont la taille dépasse celle d'une FIFO. Dans ce cas, l'accès à un *chunk* permet de précharger l'ensemble des données de la chaîne de *chunks* et le transfert des données entre la mémoire du FPGA et le noyau de calcul peut être réalisé en plusieurs fois.

5.2 Application du modèle de programmation à l'algèbre linéaire creuse

De manière intuitive, la structure de données utilisée pour stocker une matrice A de dimensions $m \times n$ est un tableau bidimensionnel de mêmes dimensions. Chaque entrée du tableau correspond à la valeur d'un élément de la matrice. Cette structuration des données est efficace pour stocker des matrices denses, car les éléments sont accédés directement en utilisant leurs indices pour effectuer de l'arithmétique de pointeurs, ce qui permet de mémoriser uniquement les valeurs des éléments de la matrice. La densité d'une matrice est définie par son nombre d'éléments non nuls (*number of nonzero* (nnz)) divisé par son nombre total d'éléments ($m \times n$). Par opposition à une matrice dense dont la majorité des éléments sont non nuls, une matrice creuse a une faible densité. Utiliser un tableau bidimensionnel de dimensions $m \times n$ pour stocker une matrice creuse revient à stocker une majorité d'éléments inutiles. Pour réduire leur empreinte mémoire, les matrices creuses sont stockées avec des formats compressés permettant d'éliminer les éléments nuls. Hormis le cas où le motif d'une matrice permet de déterminer les indices des éléments non nuls de manière triviale (par exemple : une matrice où tous les éléments non nuls sont positionnés sur la diagonale de la matrice), la compression d'une matrice nécessite de stocker des informations permettant de définir les indices des éléments non nuls. Ainsi, un format de stockage compressé d'une matrice

creuse résulte en une structure de données plus complexe qu'un tableau bidimensionnel.

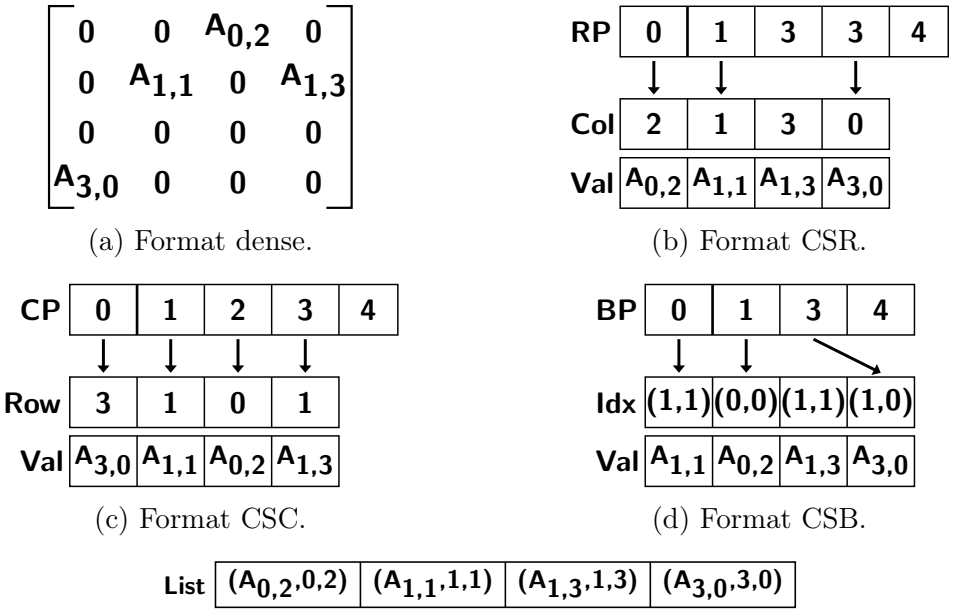
Les opérations d'algèbre linéaire creuse sont des opérations d'algèbre linéaire qui opèrent sur des matrices creuses ou des vecteurs creux, dont le format de stockage est compressé. Le parti pris des opérations d'algèbre linéaire creuse est de pouvoir tirer un avantage de la présence d'éléments nuls dans la matrice. Ainsi, l'utilisation d'un format de stockage compressé de matrices creuses et d'un opérateur d'algèbre linéaire creuse doit être économique à la fois en termes de stockage et de charge de travail. Les solveurs directs utilisant des opérateurs d'algèbre linéaire creuse permettent de résoudre de très gros problèmes qui ne peuvent pas être résolus par les solveurs utilisant des opérateurs d'algèbre linéaire dense.

5.2.1 Formats de stockage compressé de matrices creuses de l'état de l'art

Considérons une matrice A de dimensions $m \times n$ et composée de nnz éléments non nuls, telle que représentée sur la figure 5.3a. Le format de stockage naïf, appelé liste de coordonnées (COO), consiste à utiliser une liste de nnz tuples où chacun est composé de la valeur d'un élément non nul, de son indice de rangée et de son indice de colonne, tel qu'illustré sur la figure 5.3e. Le principal inconvénient de ce format est d'utiliser trois valeurs pour stocker chaque élément non nul de la matrice, résultant au stockage de $3nnz$ scalaires

Parmi les différents formats de compression existant, le format *Compressed Sparse Row* (CSR) est un des plus connus. Il est également appelé le format Yale parce qu'il a été décrit pour la première fois de manière explicite dans un rapport de recherche du département d'informatique de l'Université de Yale [43]. Tel que représenté sur la figure 5.3b, le format CSR stocke les éléments non nuls d'une matrice creuse dans l'ordre des rangées avec trois tableaux :

— Val, de dimension nnz, contient la valeur des éléments.
— Col, également de dimension nnz, contient les indices de colonnes.
— RP, de dimension $n+1$, contient des pointeurs permettant de délimiter les n rangées de la matrice. $RP[i]$ indique l'index du premier élément non nul de la rangée d'indice i dans les tableaux Val et Col. L'opération $RP[i+1] - RP[i]$ retourne le nombre d'éléments non nuls dans la



(a) Format dense.

(b) Format CSR.

(c) Format CSC.

(d) Format CSB.

(e) Format utilisant une liste de coordonnées (COO).

FIGURE 5.3 – Représentation des formats COO, CSR, CSC et CSB pour une matrice de dimensions 4×4 et composée de 6 éléments non nuls.

rangée i. La valeur de $RP[n]$ est égale à nnz.

De cette manière, le format CSR permet de stocker seulement $n + 2nnz$ scalaires. Ce format présente deux avantages par rapport au format COO : il a une empreinte mémoire plus faible et il permet d'accéder facilement aux données de la matrice par rangée. Le format *Compressed Sparse Column* (CSC) est une déclinaison du format CSR, où les éléments non nuls de la matrice sont stockés dans l'ordre des colonnes. Tel que représenté sur la figure 5.3c, les valeurs et les indices de rangées des éléments non nuls sont stockés dans l'ordre croissant des colonnes, respectivement dans les tableaux Val et Row. Le tableau CP, de dimension $m+1$, contient les pointeurs permettant de délimiter les m colonnes de la matrice. Le stockage au format CSR d'une matrice A correspond à sa transposée A^T stockée au format CSC.

Les formats CSR et CSC sont préférés au format COO, notamment parce que les opérations d'algèbre linéaire creuse traitent les matrices en accédant aux données à la granularité d'une rangée ou d'une colonne. Cependant, certaines opérations nécessitent de effectuer à la fois des accès par rangées et par colonnes à la même matrice. C'est le cas pour une opération qui traite une matrice A et sa transposée A^T. Pour ces opérations, l'utilisation du format CSR favorise l'accès aux rangées par rapport aux accès par colonnes et vice

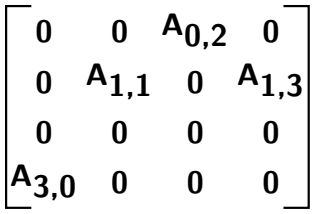
versa avec l'utilisation du format CSC. Pour répondre à ce problème, Buluç *et al.* ont proposé le format *Compressed Sparse Blocks* (CSB) [18] qui permet de compresser une matrice sous la forme de blocs carrés de dimensions $\beta \times \beta$. L'objectif de ce nouveau format est de ne pas favoriser l'accès aux rangées au détriment de l'accès aux colonnes, et inversement. Le principe de ce format est de partitionner une matrice A de dimension $n \times m$ en nm/β^2 blocs. Chaque bloc A_{ij} est une sous-matrice de A contenant les éléments dont l'indice de rangée est compris dans l'ensemble $[i\beta, (i+1)\beta-1]$ et dont l'indice de colonne est compris dans l'ensemble $[j\beta, (j+1)\beta-1]$. Le format CSB stocke un bloc en mémorisant chaque élément non nul de la sous-matrice avec un tuple en associant la valeur de l'élément, son indice de colonne et son indice de rangée dans la sous-matrice. En profitant de la décomposition de la matrice A en sous-matrices, le stockage des indices nécessite moins de bits. Si $\beta = \sqrt{n}$, chaque indice d'un élément dans une sous-matrice requiert la moitié des bits nécessaires au stockage du même indice dans la matrice A. Ainsi, le format CSB ne requiert pas plus de mémoire pour stocker les indices des éléments non nuls que les formats CSR ou CSC. Tel que représenté sur la figure 5.3d, le format CSB utilise trois tableaux pour stocker une matrice. Les tableaux Val et Idx mémorisent, respectivement, la valeur des éléments non nuls et leur paire d'indice dans l'ordre des blocs. Le tableau BP indique la position du premier élément de chaque bloc dans les tableaux Val et Idx. Sur la figure 5.3d, les blocs et les éléments sont stockés dans l'ordre croissant des rangées. Cependant, différentes stratégies peuvent être adoptées pour ordonner les blocs et les éléments. Par exemple, Buluç *et al.* [18] arguent l'utilisation d'une courbe de Morton [80] pour ordonner les données afin d'améliorer la localité spatiale des accès.

Les formats CSR, CSC et CSB tels que présentés précédemment ne sont pas bien adaptés pour le traitement par un FPGA sous forme de flux. Pour expliquer le problème, prenons l'exemple d'une matrice A stockée avec un format CSR dans la mémoire du FPGA et qui est parcourue à l'initiative d'un noyau de calcul. Pour accéder à une rangée, le noyau de calcul doit, dans un premier temps, accéder au tableau RP pour définir la position en mémoire des éléments de la rangée et déterminer le nombre d'éléments non nuls contenus dans la rangée. Ensuite, le noyau doit initier un premier accès au tableau Col, puis un second au tableau Val. Les données sont transférées sous la forme de flux. Le premier problème est que le noyau doit disposer de l'indice

de colonne et de la valeur d'un élément pour le traiter. Ainsi, avec ce schéma d'accès, le noyau doit stocker le flux d'indices de colonnes reçu en premier, puis attendre la réception du flux de valeurs pour initier le traitement. Le deuxième problème est que si le noyau veut accéder à une rangée aléatoire, seules deux données du tableau RP sont utiles. Si le bus de données reliant la mémoire et le noyau de calcul est large, par exemple si sa largeur est de 512 bits, les accès au tableau RP seront inefficaces conduisant à une mauvaise utilisation de la bande passante. Ces formats ne sont également pas bien adaptés à l'utilisation de la MVP. Si chaque tableau (RP, Col et Val) est alloué dans un unique *chunk*, l'accès à une rangée aléatoire nécessiterait de transférer toute la matrice vers la mémoire du FPGA.

5.2.2 Contribution : compression de matrices creuses sous forme de *chunks*

Pour répondre à ces problèmes, nous avons adapté les formats de stockage compressé de matrices creuses à l'utilisation de la MVP et au traitement par flux. Pour ce faire, nous avons utilisé le principe général du modèle de programmation exposé dans la section 5.1. Ce principe consiste à partitionner les structures de données en *chunks* en fonction de la granularité des accès aux données pour pouvoir y accéder sous la forme d'un flux régulier de *chunks*. Pour les matrices creuses, la granularité des accès est facile à identifier. Elle est définie par la structure de données utilisées : une rangée pour le format CSR, une colonne pour le format CSC ou un bloc pour le format CSB. Tel que le représente la figure 5.4b, nous avons adapté le format CSR à l'utilisation des *chunks*. L'indice de colonne et la valeur de chaque élément sont colocalisés pour former des paires. L'ensemble des paires d'une rangée est stocké dans le même *chunk*. Nous utilisons les métadonnées des *chunks* pour indiquer le nombre d'éléments contenus dans une rangée. De cette manière, l'accès en lecture à une rangée peut se faire uniquement en utilisant son indice. Ce format présente trois avantages par rapport au format original : il permet de réduire le nombre d'accès nécessaires pour lire ou écrire une rangée, il facilite les accès aléatoires aux rangées car chacune est indépendante, il augmente la localité des données et permet d'accéder à une rangée sous la forme d'un seul flux. La décomposition d'une matrice creuse en *chunks* peut être facilement adaptée à différents formats de compression, tel qu'illustré sur la figure 5.4c pour le



(a) Format dense.

ID	count	
0	1	(2,$A_{0,2}$)
1	2	(1,$A_{1,1}$) (3,$A_{1,3}$)
3	1	(0,$A_{3,0}$)

(b) Format compressé par rangée sous forme de chunks.

ID	count	
0	1	(3,$A_{3,0}$)
1	1	(1,$A_{1,1}$)
2	1	(0,$A_{0,2}$)
3	1	(1,$A_{1,3}$)

(c) Format compressé par colonnes sous forme de chunks.

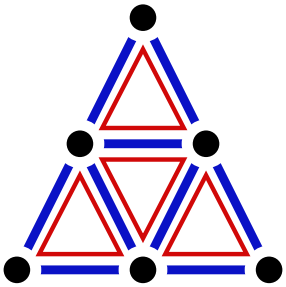
ID	count	
0	1	((1,1),$A_{1,1}$)
1	2	((0,0),$A_{0,2}$) ((1,1),$A_{1,3}$)
3	1	((1,0),$A_{3,0}$)

(d) Format compressé par blocs, ordonnés par rangées, sous forme de chunks.

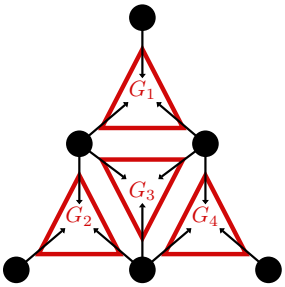
FIGURE 5.4 – Représentation des formats de stockage compressé de matrices creuses sous forme de chunks pour une matrice de dimensions 4×4 et composée de 6 éléments non nuls.

format CSC ou sur la figure 5.4d pour le format CSB.

Pour un noyau de calcul instancié sur un FPGA et en utilisant la décomposition d'une matrice creuse par rangées sous la forme de *chunks*, le parcours de la matrice consiste à demander l'accès à chaque rangée, ce qui a pour effet de charger les données des rangées dans la mémoire du FPGA, puis à demander le transfert des données entre la mémoire du FPGA et le noyau de calcul. La dissociation des requêtes de transfert et des requêtes d'accès aux *chunks* permet de précharger les données dans la mémoire du FPGA afin de masquer les latences d'accès. Pour que le préchargement soit efficace, les requêtes d'accès aux rangées doivent être émises par séquences. La décomposition de la matrice en *chunks* permet d'abstraire l'irrégularité de la matrice parce que l'accès à une rangée implique uniquement d'indiquer l'identifiant du *chunk* correspondant qui est aisé à déterminer en faisant de l'arithmétique avec l'indice de la rangée et une adresse de base (adresse dans l'espace d'adressage logique de la MVP du *chunk* de la première rangée de la matrice). De cette manière, la génération d'une séquence de *chunks* est facile à réaliser, peu importe l'ordre dans lequel les rangées sont accédées.



(a) Représentation du maillage. Les ronds noirs représentent les nœuds du maillage, les lignes bleues représentent les arêtes et les triangles rouges représentent les éléments du maillage.



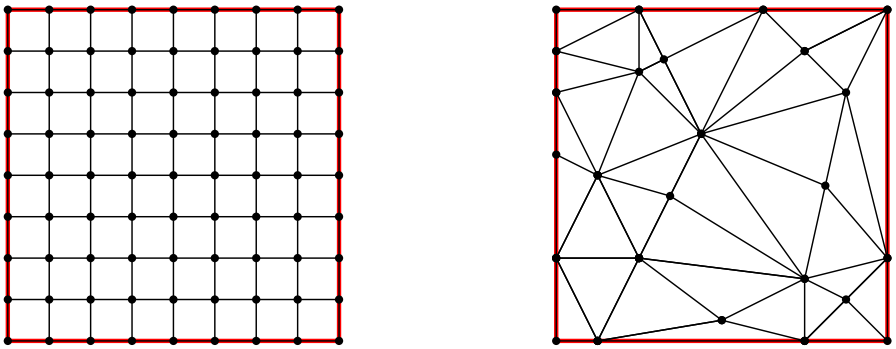
(b) Représentation d'une mise à jour de la valeur des triangles du maillage en fonction de la valeur des nœuds voisins.

FIGURE 5.5 – Représentation d'un maillage bidimensionnel triangulaire à 4 triangles et de son traitement.

5.3 Application du modèle de programmation pour le calcul par méthode des éléments finis

La méthode des éléments finis est largement utilisée dans les applications de simulations numériques pour résoudre des équations aux dérivées partielles. Ces applications opèrent sur des maillages qui permettent de modéliser un espace à N dimensions (généralement à 2 ou 3 dimensions) en le partitionnant en éléments. Tel qu'illustré sur la figure 5.5a, un maillage est composé de nœuds, d'arêtes et d'éléments. La charge de travail principal des applications opérant sur des maillages consiste à mettre à jour les données du maillage, c'est-à-dire les valeurs des nœuds, des arêtes ou des éléments, en appliquant sur plusieurs itérations une fonction aux valeurs du voisinage. Un exemple typique d'une telle fonction est illustré sur la figure 5.5b. La valeur de chaque élément, en l'occurrence des triangles, est mise à jour en appliquant une fonction sur la valeur des nœuds correspondant aux sommets des éléments.

On distingue deux classes de maillages qui se différencient par leur type de connectivité entre les éléments : les maillages structurés et les maillages non structurés. La connectivité d'un élément est la liste de ses voisins. Un maillage structuré a une connectivité fixe, par exemple une grille telle que représentée sur la figure 5.6a. Par opposition, un maillage non structuré a une connectivité



(a) Maillage structuré. (b) Maillage non structuré triangulaire.

FIGURE 5.6 – Comparaison d'un maillage structuré bidimensionnel et d'un maillage non structuré bidimensionnel

arbitraire, tel que représenté sur la figure 5.6b. Typiquement, un maillage non structuré est composé d'éléments triangulaires dans un espace bidimensionnel et d'éléments tétraédriques dans un espace tridimensionnel.

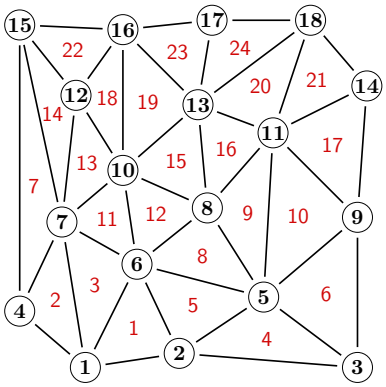
Le traitement d'un maillage structuré est plus simple à réaliser. La connectivité des éléments est implicite et le maillage peut être défini seulement par sa liste de nœuds. Par opposition, le traitement d'un maillage non structuré est plus complexe à réaliser. Il nécessite d'utiliser une structure de données pour décrire la topologie du maillage. Cependant, aujourd'hui les maillages non structurés sont très utilisés dans les applications de simulations numériques parce qu'ils facilitent la modélisation de structures à la géométrie irrégulière et permettent d'utiliser une résolution variable sur différents espaces du maillage. Le recours à une résolution variable permet d'utiliser moins de mémoire et de réduire la charge de travail en adaptant le nombre d'éléments dans certaines régions en fonction du besoin applicatif. Ainsi, comme pour les opérateurs d'algèbre linéaire creuse, les applications de méthode des éléments finis opérant sur des maillages non structurés permettent de résoudre des problèmes plus volumineux et plus complexes.

Motif d'accès aux données d'un maillage non structuré. Comme indiqué précédemment, les applications de méthode des éléments finis procèdent par mise à jour des données du maillage sur plusieurs itérations en appliquant une fonction aux valeurs du voisinage. L'utilisation des maillages non structurés nécessite le recours à des structures de données décrivant la topologie du maillage pour obtenir les valeurs du voisinage de chaque élément, nœud ou

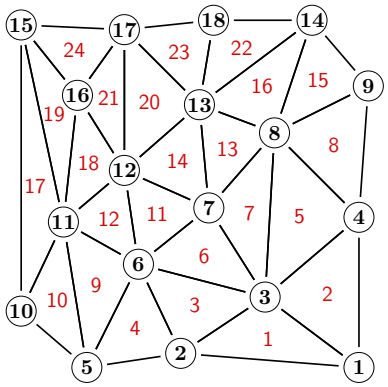
```
1  void f(size_t nTriangles, /*
2         Number of triangles in mesh */
3         size_t nVertices,  // Number of nodes in mesh
4         int elem_nodes[nTriangles][3], /*
5         Mesh's topology */
6         double elem_val[nTriangles], // Element values
7         double node_val[nVertices]) // Node values
8  {
9    for(int i = 0; i < nTriangles; ++i){
10       elem_val[i] = foo(node_val[elem_nodes[i][0]],
11                         node_val[elem_nodes[i][1]],
12                         node_val[elem_nodes[i][2]]);
```

Code 5.1 – Exemple de fonction appliquée à un maillage non structuré.

arête. Par conséquent, l'accès aux valeurs du maillage se fait par indirections, ce qui rend les applications *memory-bound* : les performances de l'application sont limitées par les performances du système mémoire. Cet effet s'explique, d'une part, avec le chargement des structures de données décrivant la topologie du maillage qui augmente le trafic mémoire et fait baisser l'intensité arithmétique et, d'autre part, par les accès indirects aux données qui sont aléatoires et dépendants des données. Pour illustrer le problème, prenons l'exemple de la fonction décrite par le code 5.1 qui réalise le traitement représenté sur la figure 5.5b. Dans cet exemple, pour chaque triangle, le traitement de trois valeurs nécessite le chargement de six données. Si on applique ce code pour traiter le maillage illustré sur la figure 5.7a, l'exécution va provoquer des accès au tableau *node_val* avec une mauvaise localité spatiale, parce que la différence d'indice des sommets de certains triangles est élevée (par exemple pour le triangle numéro 7, la différence d'indice entre le sommet 4 et 15 est de 11). La mauvaise localité des données d'un maillage a pour conséquence de faire chuter les performances d'exécution. Elle augmente les échanges de données entre les différents niveaux de la hiérarchie mémoire du système de calcul et par conséquent amplifie l'effet *memory-bound*. En réordonnançant les données en mémoire, il est possible d'augmenter la localité des accès aux données sans modifier la topologie du maillage. Le réordonnancement des données se fait en modifiant l'indexation des éléments, des nœuds et des arêtes du maillage. Pour l'illustrer, si on applique le code 5.1 sur le même maillage, mais en changeant l'indexation des nœuds tel qu'illustré sur la figure 5.7b, la localité des



(a) Maillage original. La différence d'indices maximale entre les sommets d'une arête est de 11.



(b) Maillage réordonnancé. La différence d'indices maximale entre les sommets d'une arête est de 6.

FIGURE 5.7 – Représentations d'un maillage bidimensionnel à 18 nœuds et 24 triangles.

accès aux données est meilleure parce que la différence maximale d'indice des sommets des triangles est réduite à 6.

5.3.1 Application du principe général du modèle de programmation au traitement de maillages non structurés

L'application du principe général du modèle de programmation, présenté dans la section 5.1, au traitement d'un maillage non structuré par méthode des éléments finis nécessite de partitionner les structures de données du maillage sous forme de *chunks*. Ce partitionnement doit permettre d'abstraire l'irrégularité de la structure de données afin de pourvoir parcourir les données sous la forme d'un flux continu de *chunks*. Comme exposé précédemment, les accès par indirections aux données du maillage rendent les accès mémoires aléatoires. Par conséquent, le partitionnement en *chunks* d'un maillage non structuré est moins trivial à réaliser que celui appliqué aux matrices creuses, parce que la granularité d'accès aux données sous forme de blocs n'est pas explicite. Cependant, nous pouvons observer qu'un maillage non structuré est composé :

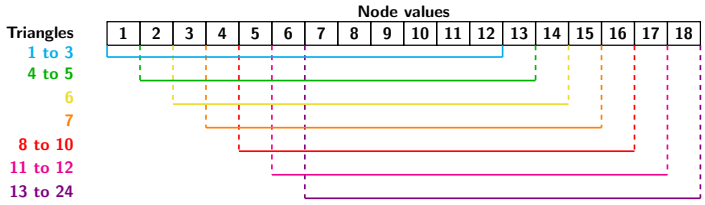
— d'éléments dont le nombre de sommets est fini et généralement identique pour l'ensemble du maillage (des maillages hybrides peuvent utiliser des éléments avec un nombre différent de sommets, par exemple en

combinant des triangles et des quadrilatères) ;
— d'arêtes connectées à deux nœuds et étant à l'intersection de deux éléments au maximum ;
— de sommets qui possèdent un nombre maximum de nœuds voisins, également appelé degré maximal et dénoté $\Delta(M)$ pour un maillage M.

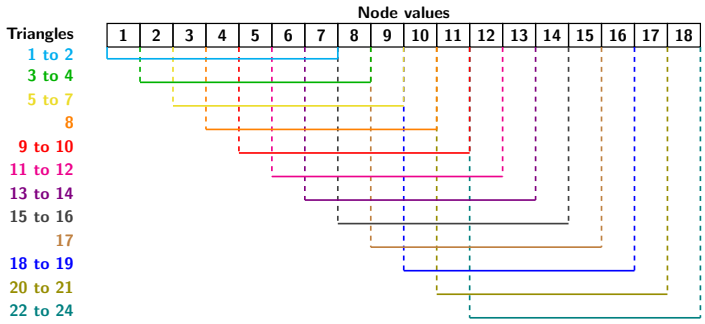
De cette manière, les données utilisées pour mettre à jour la valeur d'un élément, d'un nœud ou d'une arête, sont comprises dans un intervalle mémoire borné par le plus petit indice de voisin et le plus grand indice de voisin. En considérant que le traitement du maillage se fait en le parcourant dans un ordre séquentiel, par exemple dans l'ordre des indices des sommets, il est possible d'accéder aux données du maillage de manière continue et séquentielle à travers une fenêtre glissante. Cette observation est utilisée par Pablo Barrio et Carlos Carreras [12], pour parcourir les valeurs des sommets de maillages non structurés triangulaires. Une fenêtre glissante peut être définie comme un objet mémoire de taille fixe qui se déplace de manière séquentielle le long de la structure de données qui contient les valeurs nécessaires au traitement. Le glissement de la fenêtre est cadencé par l'évolution du traitement. En utilisant ce concept de fenêtre glissante, il est possible de partitionner les données du maillage en *chunks* de taille fixe et d'alimenter la fenêtre avec un flux continu de *chunks* dont les identifiants sont séquentiels. De ce point de vue, l'utilisation d'une fenêtre glissante répond aux besoins du modèle de programmation. Cependant, la taille de la fenêtre glissante doit permettre d'accéder à toutes les données nécessaires au traitement d'un élément, d'un nœud ou d'une arête selon la fonction appliquée. Si nous reprenons l'exemple du code 5.1 appliqué au maillage représenté sur la figure 5.7a, la taille de la fenêtre glissante doit permettre de contenir au minimum 12 valeurs de nœuds tel qu'illustré sur la figure 5.8a, soit presque toutes les données du maillage. En changeant l'indexation des sommets avec l'utilisation du maillage illustré sur la figure 5.7b, il est possible de réduire la taille de la fenêtre glissante, tel que représenté sur la figure 5.8b. Ainsi, pour rendre viable l'utilisation d'une fenêtre glissante, les données des maillages doivent être réordonnées.

5.3.2 Réordonnancement des données d'un maillage

Le réordonnancement des données d'un maillage se fait en utilisant un algorithme de tri. Généralement, le tri s'applique sur les nœuds du maillage,



(a) Évolution de la fenêtre glissante pour le maillage original. La taille minimale de la fenêtre glissante est de 12 valeurs.



(b) Évolution de la fenêtre glissante pour le maillage réordonnancé. La taille minimale de la fenêtre glissante est de 7 valeurs.

FIGURE 5.8 – Représentations de l'évolution d'une fenêtre glissante pour les maillages représentés sur la figure 5.7. Pour chaque maillage la taille de la fenêtre glissante est fixe et correspond à la différence d'indice maximale entre deux nœuds voisins.

puis les arêtes et les éléments sont indexés dans l'ordre croissant des indices de leurs sommets (les nœuds du maillage). Plusieurs algorithmes de tri visant des objectifs différents existent. Intuitivement, pour accéder aux données d'un maillage à travers une fenêtre glissante, l'objectif du tri doit être de réduire la largeur de bande de la matrice d'adjacence du maillage. La matrice d'adjacence A d'un maillage $M = (V, E)$, où V est l'ensemble des nœuds du maillage avec $|V| = n$ et où E est l'ensemble des arêtes, permet de représenter la connectivité de ses nœuds. Supposons que les nœuds sont numérotés arbitrairement $v_1, ..., v_n$. La matrice A est une matrice creuse symétrique booléenne de dimensions $n \times n$ définie telle que :

$$a_{ij} = \begin{cases} 1 & si \quad (v_i, v_j) \in E \\ 0 & sinon \end{cases}$$

La largeur de bande d'une matrice creuse A est la somme de sa largeur basse de bande k_b et de sa largeur haute de bande k_h. k_b est le plus petit entier,

supérieur où égal à 0, tel que tout élément a_{ij} où $j < i - k_b$ est nul. k_h est le plus petit entier, supérieur où égal à 0, tel que tout élément a_{ij} où $j > i + k_h$ est nul. La largeur de bande d'une matrice d'adjacence d'un maillage est égale à la différence d'indices maximale entre deux nœuds connectés.

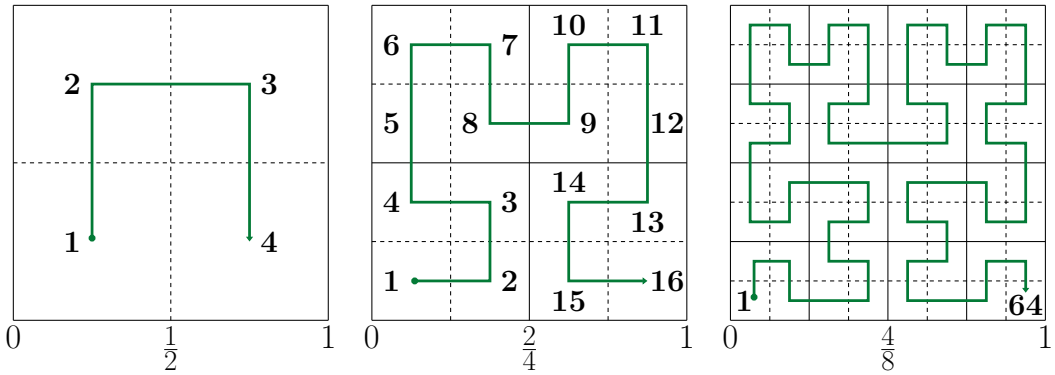
Algorithme 1 Algorithme de parcours en largeur pour le tri des sommets d'un maillage.

```
Input: nVertices                                   ▷ Number of mesh nodes
Output: order                                      ▷ Node index vector
  state[0 :nVertices-1] ← UNVISITED
  index ← 0
  queue                                            ▷ Empty queue
  seed ← select_seed()
  queue ← insert(seed)
  state[0] ← VISITED
  while queue ≠ empty do
    current ← extract(queue)                       ▷ current node being processed
    order[current] ← index
    index ← index + 1
    while not all neighboring nodes of current node are visited do
      neighbor ← select_neighbor(current)
      if state[neighbor] = UNVISITED then
        queue ← insert(neighbor)
        state(neighbor) ← VISITED
      end if
    end while
  end while
```

Méthodes de tri réduisant la largeur de la bande de la matrice d'adjacence d'un maillage. L'algorithme de parcours en largeur (*Breadth First Search* (BFS)) et l'algorithme de Cuthill-McKee inverse [77] (*Reverse Cuthill-McKee* (RCM)) sont fréquemment employés pour réduire la largeur de bande de la matrice d'adjacence d'un maillage. Tel que décrit par l'algorithme 1, le parcours en largeur consiste à visiter tous les nœuds d'un maillage en se déplaçant par voisinage à partir d'un premier nœud, appelé graine. Au cours de la traversée du maillage, les nouveaux nœuds visités sont insérés dans une file (*queue*) qui définit le nouvel ordre. La qualité du tri est déterminé en fonction de la largeur de bande de la matrice d'adjacence du maillage obtenue. L'ordre dans lequel les voisins d'un nœud sont visités et le choix de la graine sont deux facteurs déterminants pour la qualité du tri. Le tri réalisé avec l'algorithme

RCM consiste à appliquer l'algorithme de Cuthill-McKee [36] à la matrice d'adjacence du maillage et à inverser l'ordre obtenu. Cet algorithme procède à une permutation des éléments non nuls d'une matrice creuse symétrique de sorte à former une bande de largeur réduite. De cette manière, cette méthode de tri permet de réduire la largeur de bande de la matrice d'adjacence du maillage. Dans les faits, l'algorithme de Cuthill-McKee utilise un parcours en largeur. Un algorithme de parcours en largeur traverse un maillage de la même manière qu'une application de méthode des éléments finis : en se déplaçant de voisin en voisin. Ainsi, intuitivement, une méthode de tri BFS est la plus adaptée pour permettre l'accès aux données d'un maillage à travers une fenêtre glissante. Pablo Barrio et Carlos Carreras [12] utilisent un tri BFS pour trier les données de maillage non structurés triangulaires. Malheureusement, cet algorithme ne passe pas à l'échelle lorsque la taille d'un maillage augmente. La taille maximale qu'occupe la file au cours de l'exécution de l'algorithme correspond à la différence d'indices maximale entre deux nœuds connectés. Ainsi, plus un maillage a un nombre de nœuds grand et un nombre de voisins par nœud élevé, plus la largeur de la bande de la matrice d'adjacence augmente.

Méthode de tri utilisant une courbe de remplissage. Un maillage est constitué d'un ensemble de nœuds répartis dans un espace euclidien à n-dimensions, où chaque nœud est connecté à un sous-ensemble de nœuds géométriquement proches. Un algorithme de parcours en largeur utilise la connectivité des nœuds pour les ordonner. De manière opposée, une courbe de remplissage peut permettre de trier les nœuds en utilisant uniquement leurs coordonnées géométriques. L'étude des courbes de remplissage a commencé, avec les travaux de Giuseppe Peano [87] et David Hilbert [61], suite aux démonstrations de Georg Cantor, prouvant l'existence d'une correspondance bijective entre l'intervalle unité $[0, 1] \in \mathbb{R}$ et le carré unité $[0, 1]^2 \in \mathbb{R}^2$ [19]. Dans l'espace bidimensionnel, une courbe de remplissage est une courbe continue, dont le domaine de définition est l'intervalle unité et dont l'image contient l'entièreté du carré unité, c'est-à-dire une courbe recouvrant l'ensemble d'une surface. Pour l'illustrer, prenons l'exemple d'une des courbes de remplissage parmi les plus connue : la courbe de Hilbert [61] dans l'espace bidimensionnel. Telle que représentée sur la figure 5.9, une courbe de Hilbert peut être construite de manière itérative et par la discrétisation de son espace (le carré unité en 2-dimensions). À la première itération (figure 5.9a), l'espace est subdi-



(a) Première itération. (b) Deuxième itération. (c) Troisième itération.

FIGURE 5.9 – Représentation des trois premières itérations de la courbe de Hilbert dans l'espace 2D.

visé en 4 sous-espaces carrés et une courbe est tracée pour traverser les 4 carrés. À la deuxième itération (figure 5.9b), chaque sous-espace obtenu à l'itération précédente est subdivisé en 4 et le motif de la courbe tracée à l'itération précédente est reproduit en appliquant une mise à l'échelle et une rotation et/ou réflexion de sorte que le motif formé par l'ensemble des sous-espaces réunis soit continu. La procédure peut être reproduite de manière infinie, jusqu'à ce que la courbe remplisse tous les points de l'espace. La construction de la courbe de Hilbert et plus généralement d'une courbe de remplissage peut se faire de manière récursive à l'aide d'une grammaire [11, Chapitre 3] ou en utilisant une représentation arithmétique de la courbe [11, Chapitre 4]. Une courbe de remplissage peut également être tracée dans un espace tridimensionnel [11, Chapitre 8]. En profitant de leurs propriétés de localité [11, Chapitre 11], les courbes de remplissage permettent d'opérer un tri et un partitionnement efficace des données d'un maillage [11, Chapitre 10] [94].

En considérant que le tri réalisé avec un parcours en largeur ne permet pas de satisfaire les besoins du modèle de programmation lorsque la taille d'un maillage grandit, nous avons choisi d'utiliser les courbes de remplissage comme alternative pour trier et partitionner les données d'un maillage non structuré. Tel que représenté sur la figure 5.10, le tri et le partitionnement avec une courbe de remplissage, en l'occurrence une courbe de Hilbert, se fait en trois étapes :

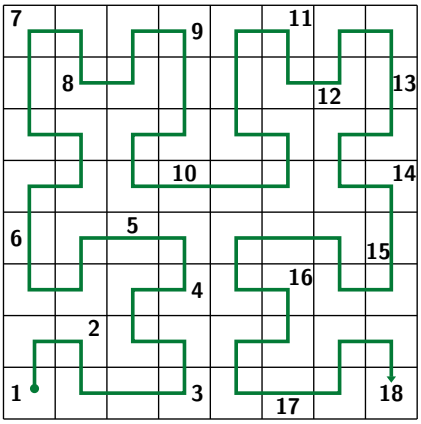
— la première étape (figure 5.10a) consiste à discrétiser l'espace du maillage sous la forme d'une grille de dimensions $2^n \times 2^n$, avec n un entier naturel

positif suffisamment grand pour que chaque case de la grille contienne au plus un nœud du maillage. Ensuite, une courbe correspondant à la n-ième itération est tracé dans l'espace du maillage et les nœuds sont indexés dans l'ordre où ils rencontrent la courbe.

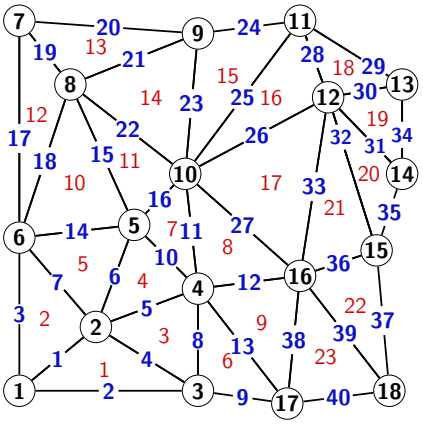
— la deuxième étape (figure 5.10b) est l'indexation des éléments et des arêtes du maillage qui est réalisée en fonction de l'ordre croissant de leurs sommets.

— la dernière étape (figure 5.10c) est le partitionnement des données du maillage. Il est réalisé en utilisant des *chunks* de taille fixe dans lesquels sont placés les données dans l'ordre de leur index.

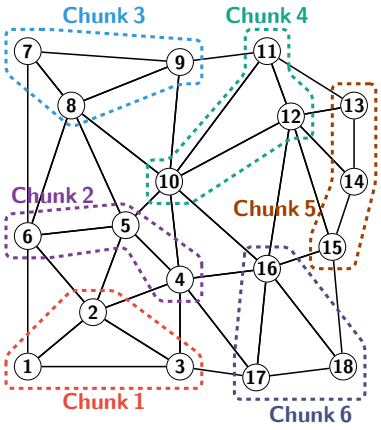
Le parti pris d'un tri réalisé avec une courbe de remplissage est d'accepter d'avoir une mauvaise localité spatiale pour une minorité des données afin d'obtenir une bonne localité pour la majorité. La mauvaise localité de certaines données est due au fait que certaines arêtes sont situées dans deux sous-espaces proches géométriquement, mais éloignés dans le sens de la courbe. Par exemple, sur la figure 5.10b, c'est le cas pour l'arête numéro 9 qui connecte le nœud 3, situé au début de la courbe, et le nœud 17, situé à la fin de la courbe. De la même manière, certains éléments peuvent avoir leurs trois sommets situés dans trois sous-espaces éloignés dans le sens de la courbe. C'est le cas par exemple, toujours sur la figure 5.10b, du triangle 8 qui a pour sommets : le nœud 4 situé sur le premier quart de la courbe, le nœud 10 situé sur le deuxième quart de la courbe et le nœud 16 situé sur le dernier quart de la courbe. Malgré cette contrainte, nous avons choisi d'utiliser une courbe de remplissage pour trier et partitionner les données afin de parcourir le maillage à l'aide d'une fenêtre glissante. Ce choix est motivé par l'observation suivante : le parcours du maillage lors de son traitement suit le tracé de la courbe utilisée pour le tri étant donné que le parcours est réalisé en fonction de l'ordre des sommets. Ainsi, la majorité des données du maillage peuvent être accédées à travers une fenêtre glissant le long de la courbe à l'allure du traitement. Le partitionnement en *chunks* étant réalisé en suivant le tracé de la courbe également, la fenêtre glissante peut être alimentée par un flux continu et séquentiel de *chunks*. Seules certaines données situées à l'intersection de plusieurs sous-espaces éloignés dans le sens de la courbe ne peuvent pas être accédées à travers la fenêtre glissante. L'accès à ces données nécessite de faire des requêtes d'accès spécifiques, c'est-à-dire hors du flux alimentant la fenêtre glissante. Cependant, ces données peuvent être aisément identifiées en suivant le tracé de la courbe.



(a) Première étape : discrétisation de l'espace géométrique du maillage et ordonnancement des nœuds en fonction du tracé de la courbe.



(b) Deuxième étape : ordonnancement des arêtes et des éléments en fonction de l'indexation des nœuds.



(c) Troisième étape : partitionnement en *chunks* des valeurs de nœuds en fonction de l'indexation des nœuds.

FIGURE 5.10 – Représentation des trois étapes du tri et du partitionnement d'un maillage non structuré bidimensionnel avec une courbe de Hilbert.

Ainsi, le modèle de programmation proposé consiste à trier et à partitionner en *chunks* de taille constante les données d'un maillage en utilisant une courbe de remplissage. Le noyau de calcul, instancié sur un FPGA, parcourt le maillage en suivant la courbe avec une fenêtre glissante, implémentée avec une FIFO adressable, qui est alimentée par un flux continu et séquentiel de *chunks*. Les données ne pouvant pas être accédées dans l'intervalle de la fenêtre glissante, le sont par l'intermédiaire de tampons mémoires qui sont alimentés par un second flux de *chunks* prédéfini en suivant le parcours de la courbe de remplissage. Le nombre de tampons à instancier est égal au nombre de

sommets accédés par la fonction de voisinage moins 1. Par exemple, pour la fonction du code 5.1, trois sommets sont accédés. Pour cet exemple, 2 tampons mémoires sont nécessaires car le sommet avec le plus petit indice du triangle en cours de traitement est toujours accessible via la fenêtre glissante (l'indice de ce sommet constitue l'adresse de base la fenêtre) et les deux autres sommets peuvent être situés dans deux sous-espaces éloignés dans le sens de la courbe, tel qu'exposé précédemment.

5.4 Conclusion

Tel que présenté dans l'introduction, un modèle de programmation sert à abstraire un système informatique pour exprimer les algorithmes et les structures de données d'un programme. Dans ce chapitre, un modèle de programmation dont l'objectif est de faciliter le portage de noyaux de calcul irréguliers sur des accélérateurs reconfigurables a été présenté. Il repose sur le paradigme de la mémoire partagée et sur la capacité donnée aux accélérateurs d'initier les accès aux données réparties. L'objectif est d'amener les bonnes données au bon endroit en exprimant l'accès aux données à l'endroit où les motifs sont générés.

Pour rendre ce modèle de programmation viable, les données doivent être préchargées pour masquer les latences d'accès. Pour y parvenir les données sont restructurées en les partitionnant en *chunks*. Cette restructuration a deux objectifs : améliorer la localité spatiale des données contenues dans un *chunk* et abstraire l'irrégularité de la structure de données. Le modèle de programmation a été illustré pour le traitement de deux structures de données largement employées dans les applications de calcul irrégulier : les matrices creuses et les maillages non structurés.

La section 5.2 a fait la démonstration du partitionnement de matrices creuses en adaptant les formats de stockages compressés les plus populaires. Cette démonstration a mis en avant que le partitionnement à la granularité des accès mémoire se fait aisément en exploitant le découpage explicite des matrices en rangées, colonnes ou blocs. La colocalisation des valeurs et des indices des éléments non nuls permet de former des paires afin d'augmenter la localité spatiale des données et de pouvoir accéder aux données d'une rangée, d'une colonne ou d'un bloc en un seul accès et en utilisant uniquement l'identi-

fiant du *chunk* (correspondant à l'indice de rangée, de colonne ou de bloc). Les métadonnées du *chunk* permettent d'indiquer la taille du bloc pour abstraire l'irrégularité de la structure de données. De cette manière, le parcours d'une matrice peut se faire par l'expression d'une séquence d'identifiants de *chunks* facilement générable.

La section 5.3 a illustré l'application du modèle de programmation pour le calcul sur des maillages non structurés par méthode des éléments finis. Il repose sur l'utilisation d'une courbe de remplissage de l'espace pour trier et partitionner en *chunks* de taille constante les données d'un maillage non structuré. Pour le traitement d'un noyau de calcul sur un FPGA, l'accès à la majorité des données du maillage se fait à travers une fenêtre glissant le long d'une courbe de remplissage utilisée pour trier les données. La fenêtre glissante est alimentée par un flux continu et séquentiel de *chunks*. Des tampons mémoires sont utilisés pour accéder à la minorité de données souffrant d'une mauvaise localité spatiale. Ces tampons sont alimentés par un flux continu de *chunks* prédéfini en suivant également le tracé de la courbe.

La définition des principes de ce modèle de programmation constitue un échelon pour rendre plus accessible la programmation de noyaux de calcul irréguliers sur des accélérateurs reconfigurables dans un système distribué et hétérogène à travers le prisme de la gestion des données. Les outils de synthèse de haut niveau constituent un deuxième échelon indispensable pour rendre l'utilisation d'accélérateurs reconfigurables plus accessible. La perspective majeure des travaux présentés dans ce chapitre est l'intégration dans un outil de synthèse de haut niveau des mécanismes d'accès aux *chunks* sous forme de flux.

Chapitre 6

Outil de simulation

Dans le chapitre 4, un modèle d'intégration d'accélérateurs reconfigurables dans une MVP a été présenté. Il repose sur une interface logicielle et matérielle qui doit être développée pour être fonctionnelle. Dans l'industrie informatique, le développement d'un produit est un ensemble de tâches (spécification, conception, implémentation, vérification, production et validation) qui doivent être réitérées pour améliorer ou corriger le produit en fonction des observations faites lors de leur réalisation. Si ces tâches sont exécutées séquentiellement, le développement peut être long et inefficace. Le manifeste pour le développement Agile de logiciels [14] a révolutionné la manière de développer des produits logiciels. Le principe de cette méthode est de développer rapidement des prototypes incomplets mais validant certaines fonctionnalités pour réduire le temps des premières itérations et aboutir au développement du produit complet plus rapidement. La méthode Agile a été adaptée au développement de composants matériels [72]. Pour ces derniers, cette méthode est notamment rendue possible par les outils de conception assistée par ordinateur pour l'électronique (*Electronic design automation* (EDA)) qui permettent de développer des prototypes avec un haut niveau d'abstraction. La première itération de la méthode Agile appliquée au développement matériel s'appuie sur l'utilisation d'outil de simulation permettant de valider des fonctionnalités à partir d'un prototype abstrait. Cette méthode est particulièrement efficace. Elle permet aux équipes travaillant sur les supports logiciels pour le produit matériel de commencer le développement en utilisant le prototype abstrait. Dans le cadre des travaux de cette thèse, nous avons décidé de nous ins-

pirer de cette méthode en développant un outil qui permet de modéliser le système à partir d'une description haut niveau et de simuler son exécution. Cette méthode nous a permis de développer et d'expérimenter le modèle de programmation présenté dans le chapitre 5.

Ce chapitre s'organise comme suit : la section 6.1 expose le contexte et les motivations qui ont conduit au développement de l'outil de simulation, la section 6.2 décrit le fonctionnement de l'outil, la section 6.3 liste les limitations de l'outil et ses perspectives d'évolutions et la section 6.4 conclut ce chapitre.

6.1 Préambule

6.1.1 Motivations

Avec le modèle d'intégration d'accélérateurs reconfigurables dans une MVP présenté dans le chapitre 4, l'accès aux données distantes par des noyaux de calcul instanciés sur un FPGA peut conduire à des latences élevées. Le modèle de programmation proposé dans le chapitre 5 a pour objectif de répondre à ce problème en masquant les latences d'accès aux données distantes. Pour évaluer la capacité du modèle de programmation à atteindre cet objectif, nous avons besoin d'un outil nous permettant de modéliser rapidement l'interface de l'API et des noyaux de calcul utilisant le modèle de programmation. Pour faciliter le travail d'exploration et pouvoir réitérer les expérimentations, nous avons également besoin d'une simulation rapide.

Étant donnée que la méthode Agile est largement employée dans les secteurs de la recherche et de l'industrie de la microélectronique, il existe de nombreux outils permettant de simuler l'exécution d'un logiciel sur un prototype matériel avec différents niveaux d'abstractions. Par exemple, certains outils EDA permettent de faire une simulation comportementale d'un prototype avec une modélisation *Register Transfer Level* (RTL) décrit avec un langage *Hardware Description Language* (HDL). Ces outils sont très utiles parce qu'ils permettent de concevoir un prototype avec précision pouvant être rapidement adapté à la production. Cependant, ils ne sont pas adaptés à nos besoins parce que l'effort de modélisation et le temps de simulation seraient trop importants par rapport à nos objectifs. Pour simuler l'activité du système que nous étudions, nous devons reproduire l'exécution logicielle de la MVP et l'activité matérielle des noyaux de calcul sur un FPGA. Réaliser une

modélisation RTL de l'ensemble de la plateforme matérielle nécessite de décrire les architectures d'un ou plusieurs processeurs, du système mémoire, du système d'interconnexion, d'un FPGA et des noyaux de calcul. Cette tâche représente un effort de modélisation trop important, notamment parce qu'une description avec un langage HDL peut être chronophage. D'autres langages permettent de modéliser et de simuler plus rapidement un système matériel tel que *SystemC*. Comme pour la modélisation RTL, ces langages nécessitent de décrire l'ensemble de la plateforme matérielle (du processeur au FPGA), ce qui est encore trop chronophage. Il existe des outils de prototypage qui permettent de modéliser et de simuler un système informatique en instanciant des composants sous la forme de modules préconçus, par exemple SESAM [109] ou SoCLib [1] qui reposent tous deux sur le langage *SystemC*. Cependant, ces outils ne répondent pas à nos attentes pour trois raisons :

— ils sont conçus pour le prototypage de multiprocesseurs sur un seul système sur puce (*Multiprocessor system on a chip* (MpSoC)) et ne proposent pas de module instanciant un FPGA. Développer ce module nécessiterait de prendre en main le code source d'un de ces outils et de le modifier, ce qui serait sujet aux erreurs et chronophage.
— le système SAT requiert les services du système d'exploitation et d'une bibliothèque MPI. Pour utiliser ces outils, il faudrait adapter le code du système d'exploitation et de la bibliothèque MPI à l'utilisation des modules préconçus, ce qui serait encore sujet aux erreurs et chronophage.
— la simulation de l'exécution du logiciel sur un système distribué et de l'activité du FPGA résulterait en un temps long.

6.1.2 Travaux connexes sur la simulation de plateforme CPU-FPGA

Il existe des outils de simulation, tels que *gem5-Aladin* [95], *PARADE* [29], *HeteroSim* [47] ou *PAAS* [76], permettant de simuler entièrement une plateforme hétérogène CPU-FPGA et de fournir des estimations de performances précises. Ces outils reposent sur l'intégration de simulateurs, tels que *gem5* [16], *Multi2Sim* [108] ou *Verilator* [98]. Pour répondre à nos attentes un outil de simulation doit satisfaire plusieurs besoins :

1. `http://www.soclib.fr`

A Permettre une modélisation haut niveau du module *FPGA-server* et des noyau de calcul, ainsi que de leurs interactions. Cette modélisation doit permettre de simuler en parallèle l'activité du module *FPGA-server* et des noyaux de calcul.

B Aboutir à un temps de simulation court afin de pouvoir mener une démarche d'exploration dans un temps raisonnable. L'exploration consiste à étudier différentes configurations de systèmes et l'impact de différents paramètres sur les performances de plusieurs noyaux de calcul. Ainsi, cette démarche conduit à produire beaucoup d'expérimentations.

C Modéliser des systèmes de calcul distribués où le CPU et le FPGA communiquent par un réseau d'interconnexion sur puce ou par un bus d'interconnexion hors puce. C'est-à-dire, que l'outil puisse modéliser un système où le CPU et le FPGA sont implémentés sur la même puce et un système où ils sont implémentés sur des puces différentes.

D Fournir une mémoire partagée entre le CPU et l'accélérateur permettant aux noyaux de calcul de faire des accès mémoire aléatoires.

E Autoriser l'emploi d'un support d'exécution permettant au module *FPGA-server* d'interagir avec la MVP logicielle.

Chacun des outils cités précédemment a des limitations et des contraintes différentes qui ne permettent pas de répondre complètement à nos attentes. Celles-ci sont synthétisées dans le tableau 6.1. Tous les outils sont pensés pour simuler sur l'accélérateur l'exécution de noyaux de calcul et ainsi ne sont pas adaptés à la modélisation du module *FPGA-server*. *HeteroSim*, *PAAS* et *PARADE* requièrent une description du matériel avec une langage HDL. Cette description peut être générée à partir d'une modélisation haut niveau grâce aux outils de synthèse haut niveau. Cependant, un outil de synthèse haut niveau n'est pas adapté à la modélisation du module *FPGA-server*. *gem5-Aladin* modélise l'accélérateur sous la forme d'un graphe de dépendances de données à partir de l'algorithme du noyau de calcul. Cette approche est efficace pour simuler des noyaux de calcul, mais ne permet pas de modéliser un module tel que le *FPGA-server*. Les outils *PAAS*, *PARADE* et *gem5-Aladin* sont limités à la simulation de systèmes sur puces hétérogènes et ainsi ne permettent pas de simuler un système distribué où le CPU et le FPGA sont implémentés sur deux puces différentes. Les outils *HeteroSim* et *gem5-Aladin* ne supportent pas la simulation d'un support d'exécution contrôlant dynamiquement les interactions

Outil \ Nos besoins	A	B	C	D	E
gem5-Aladin	✗	?	≈	✓	✗
HeteroSim	✗	✗	✓	✓	✗
PAAS	✗	✗	≈	✓	✓
PARADE	✗	✗	≈	✗	✗

Tableau 6.1 – Adéquation entre nos besoins et les outils de simulation de plateforme CPU-FPGA de l'état de l'art. Les colonnes A, B, C, D et E font références à nos attentes énumérées dans le premier paragraphe de cette sous-section. Le symbole ✗ désigne que l'outil ne permet pas de satisfaire au besoin. Le symbole ≈ indique que l'outil satisfait en partie un besoin. Le symbole ✓ signifie que l'outil permet de satisfaire au besoin. Le symbole **?** indique de nous n'avons pas l'information permettant de juger.

entre le CPU et le FPGA. L'outil *PARADE* intègre un support d'exécution qui contrôle l'ordonnancement et le placement des tâches de calcul. Ce support d'exécution ne nous semble pas nativement compatible avec l'utilisation de l'API SAT. De plus, cet outil permet seulement de simuler l'exécution de noyaux de calcul où toutes les données ont été chargées dans la mémoire locale des noyaux de calcul avant le lancement de la simulation.

Comme évoqué précédemment, la simulation d'une plateforme matérielle complète peut être longue. Une méthode pour accélérer la simulation consiste à associer une dimension temporelle à un évènement. Kirchsteiger et al. [67] ont présenté un mécanisme automatique d'annotation de délais pour remplacer l'emploi d'un simulateur d'instructions dans un simulateur *SystemC*. Papamichael *et al.* [86] ont proposé une approche utilisant l'estimation des latences de transmission de paquets de données pour remplacer la simulation d'un réseau sur puce. Avec cette approche les latences sont estimées à partir d'un modèle analytique. De ce fait, la précision de la simulation dépend de la qualité du modèle analytique. Créer un modèle analytique fiable pour notre cas d'usage semble complexe, car les latences d'accès aux données sont fortement variables en fonction de l'état de MVP, de l'activité des noyaux de calcul et de la topologie du système de calcul.

Enfin, il existe des méthodes de prototypages hybrides, comme celle proposée par Wicaksana *et al.* [112]. Pour accélérer la co-simulation d'un système matériel et logiciel, ces travaux utilisent une plateforme de simulation virtuelle pour exécuter la partie logicielle et un prototype physique instancié sur

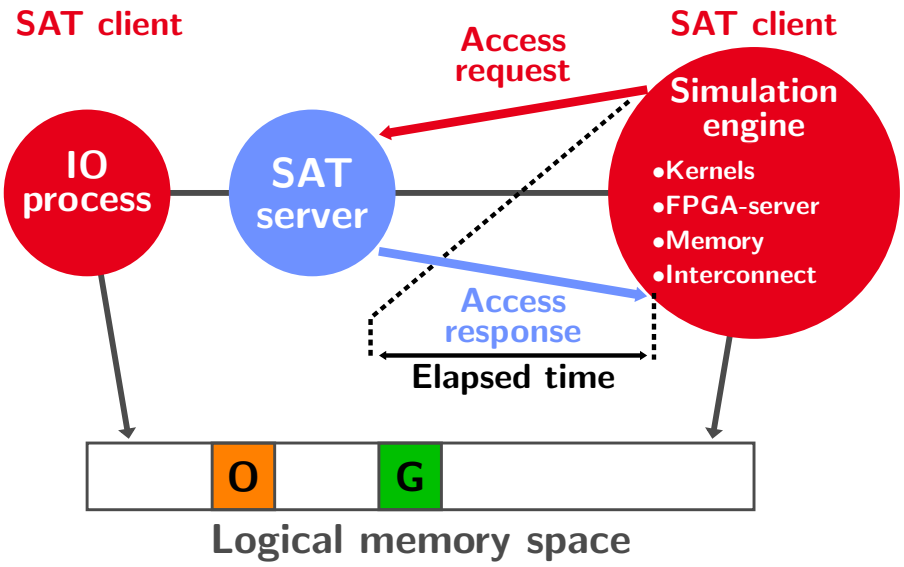


FIGURE 6.1 – Représentation de la topologie de l'outil de simulation.

un FPGA pour reproduire le comportement de la partie matérielle.

6.2 Description du fonctionnement de l'outil de simulation

Nous avons développé un outil répondant aux besoins spécifiés dans la sous-section 6.1.2. Son objectif principal est d'observer l'impact des latences d'accès aux données distantes sur les performances d'un noyau de calcul porté sur un FPGA. Pour ce faire, il emploie une méthode de simulation hybride où l'activité du FPGA est reproduite par un moteur de simulation et où la partie logicielle (les processus de la MVP) est réellement exécutée sur une architecture physique. Ainsi, tel que représenté sur la figure 6.1, l'outil utilise le système SAT comme support d'exécution. De manière analogue au modèle d'intégration d'accélérateurs reconfigurables dans une MVP proposé dans le chapitre 4, le moteur de simulation, reproduisant l'activité du FPGA, est représenté auprès de la MVP par l'intermédiaire d'un processus client. Ainsi, le moteur de simulation peut accéder aux services de la MVP, c'est-à-dire qu'il peut émettre des requêtes d'accès aux données partagées. Le temps de complétion de chaque requête est chronométré et les latences mesurées sont utilisées pour contraindre l'activité du moteur de simulation. La topologie de l'outil

de simulation est la même que celle de l'application étudiée. Typiquement, tel qu'illustré sur la figure 6.1, l'outil de simulation se compose d'un processus serveur, d'un premier processus client exécutant le moteur de simulation et d'un second processus client dont le rôle est de faire l'acquisition et l'allocation dans la MVP des données d'entrée et de restituer les données traitées par le FPGA. L'utilisation du système SAT comme support d'exécution permet de répartir les processus sur une architecture physiquement distribuée. Lors des simulations, il est possible d'exécuter chaque processus sur un nœud de calcul différent. L'intérêt de cette fonctionnalité est de pouvoir reproduire les latences d'accès aux données en fonction de différentes topologies de systèmes de calcul.

Le moteur de simulation est développé en langage *C++* pour profiter du paradigme de programmation orienté objet et du concept de classe qui est particulièrement bien adapté pour encapsuler un module matériel. De cette manière, le module *FPGA-server* est déclaré comme une classe dont les méthodes peuvent être invoquées par les noyaux de calcul afin de reproduire le comportement et les services attendus. Cette méthode de conception présente deux avantages. Le premier est que le mécanisme d'interaction entre un noyau de calcul et le module *FPGA-server* est fidèle à celui décrit dans le chapitre 4. En effet, l'émission d'une requête est conceptuellement proche de l'invocation d'une méthode. Le deuxième avantage est que la programmation orientée objet facilite la maintenance et la réutilisation du code. Ainsi, le développement des noyaux de calcul est rendu plus facile.

Le module *FPGA-server* est modélisé par une structure de données reproduisant le répertoire de *chunks* associée à des méthodes permettant de les mettre à jour, tel que décrit de la section 4.2, et d'un contrôleur mémoire. Une horloge globale orchestre l'activité des différents composants modélisés. Elle a pour rôle de synchroniser les interactions entre le module *FPGA-server* et les noyaux de calcul. Le répertoire de *chunks* est un tableau qui associe chacune de ses entrées à un emplacement de la mémoire du FPGA. Une entrée permet d'accueillir un *chunk*. Lorsqu'une entrée est attribuée, elle contient les métadonnées du *chunk* et une date qui correspond au cycle à partir duquel les données peuvent être accédées. Cette date est déterminée en fonction du cycle auquel la requête d'accès a été émise et la latence de complétion de la requête. La latence de complétion est définie en fonction du type de requête et de l'état du *chunk*. Si la mise à jour du répertoire nécessite de transférer la requête au

serveur de la MVP, le temps de complétion est celui mesuré entre l'émission de la requête et la réception de la réponse. Si la mise à jour du répertoire ne nécessite pas d'émettre une requête au serveur de la MVP, le temps de complétion est nul et les données sont immédiatement disponibles. Le nombre d'entrées du répertoire et la taille maximale d'un *chunk* définissent la taille de l'espace mémoire du FPGA modélisé. Au démarrage toutes les entrées du répertoires sont vides.

Le contrôleur mémoire agit comme un contrôleur de bus en attribuant l'accès à la mémoire au noyau de calcul qui le sollicite. Ce contrôleur accepte les transferts en rafales, c'est-à-dire qu'un transfert de données peut s'étaler sur plusieurs cycles. La quantité de données transférées à chaque cycle dépend de la largeur du bus de données interne modélisé. Le nombre de transferts en rafales par accès et la largeur du bus de données sont des paramètres définis lors de l'instanciation du contrôleur mémoire. L'accès à la mémoire est attribué à un noyau de calcul pour un flux de données spécifique correspondant au *chunk* auquel il souhaite accéder. Le contrôleur mémoire a pour rôle de s'assurer qu'un seul flux de données peut accéder à la mémoire par cycle. De cette manière, le contrôleur mémoire est vu comme une ressource partagée qui permet notamment de modéliser la bande passante mémoire. L'attribution du contrôleur mémoire prend fin lorsque le noyau relâche la ressource ou lorsque le nombre maximum de transferts en rafale autorisé a été atteint. Si toutes les données d'un *chunk* n'ont pas pu être transférées en un accès, le noyau de calcul doit refaire une demande d'accès pour compléter le transfert. Cet arbitrage permet de reproduire le protocole de transfert implémenté par un bus AXI et permet également de répartir dans le temps l'accès à la mémoire entre les différents flux de données utilisés par les noyaux de calcul.

Pour répondre aux besoins du modèle de programmation présenté dans le chapitre 5, un noyau de calcul (représenté sur la figure 6.2) est décrit comme une chaîne de traitement (*pipeline*), traversée par des flux de données, où les étages sont séparés par des files (FIFO). Chaque étage correspond à une action qui peut être d'interagir avec le module *FPGA-server* ou de réaliser un traitement sur les données. L'action réalisée par un étage doit pouvoir être complétée en un seul cycle d'horloge sur un FPGA. Les différents étages sont synchronisés par l'horloge globale. Les interactions avec le module *FPGA-server* correspondent à l'invocation d'une méthode pour demander le préchargement d'un *chunk* ou pour demander le transfert des données. Lorsqu'une demande d'accès

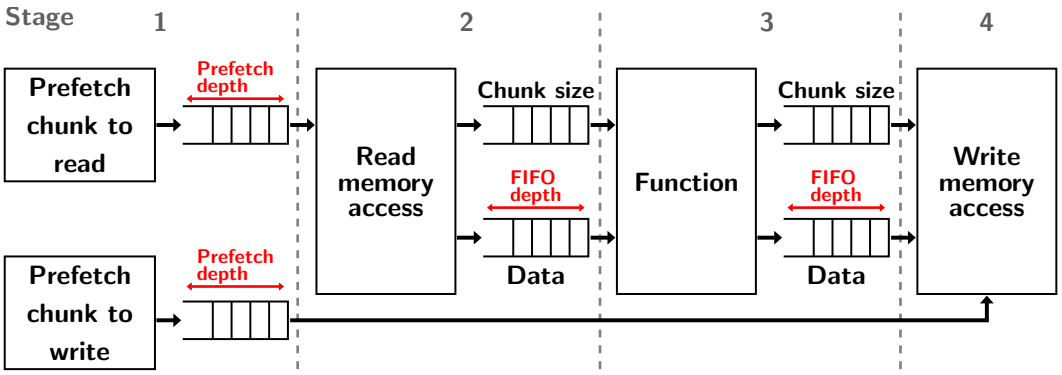


FIGURE 6.2 – Exemple d'architecture d'un noyau de calcul modélisé sous la forme d'une chaîne de traitement.

à un *chunk* est acceptée, l'identifiant du *chunk* est écrit dans la FIFO en sortie d'étage. La profondeur de cette FIFO définit le nombre de *chunks* préchargés par flux de données. La demande de transfert des données peut être effectuée seulement après l'acceptation de la requête d'accès au *chunk* et à partir du cycle auquel les données peuvent être accédées tel qu'indiqué dans le répertoire de *chunks*. Lors d'un transfert en lecture, les données sont écrites dans une FIFO en sortie d'étage. Pour qu'une requête de transfert en lecture soit émise, cette FIFO doit disposer de suffisamment d'espace pour contenir toutes les données du *chunk*. De manière similaire, pour une requête de transfert en écriture, les données sont transférées depuis une FIFO à l'entrée de l'étage. Pour que le transfert soit initié, la FIFO doit contenir toutes les données à transférer.

6.3 Spectre d'utilisation du simulateur et perspectives

L'atout principal de l'outil de simulation proposé dans les travaux de cette thèse est d'offrir la capacité de simuler rapidement l'activité d'un noyau de calcul à partir d'une modélisation haut niveau. Cependant, cet atout implique certaines limitations. Premièrement, une modélisation haut niveau est beaucoup moins précise qu'une modélisation RTL. Cet écart de précision implique certaines limitations. Notamment, la modélisation réalisée avec cet outil ne permet pas d'effectuer une 'analyse temporelle statique. Elle ne permet pas non plus d'évaluer l'utilisation des ressources du FPGA, ni d'évaluer la consom-

mation énergétique du FPGA. Sans analyse temporelle statique, il n'est pas possible de définir de manière fiable la fréquence à laquelle le noyau de calcul peut fonctionner. Cette limitation est particulièrement impactante pour la modélisation d'un noyau de calcul réalisant des traitements complexes sur les données conduisant à une fréquence de fonctionnement en dessous de la cadence maximale de l'horloge du FPGA. Le manque d'évaluation de l'utilisation des ressources du FPGA est également impactant. Il ne permet pas de quantifier le coût de l'implémentation du module *FPGA-server*, ni de déterminer la quantité de noyaux de calcul pouvant être instanciée. L'utilisation des FPGA en tant qu'accélérateurs matériels est principalement motivée par leur capacité à augmenter l'efficacité énergétique des applications. L'impossibilité d'évaluer la consommation énergétique du FPGA avec l'outil de simulation limite donc notre capacité à évaluer les performances du système à travers le prisme de l'énergie.

La modélisation du contrôleur mémoire est également sujette à des limitations. Son rôle est d'attribuer l'accès à la mémoire à un seul flux de données par cycle et de reproduire la limite de bande passante de la mémoire du FPGA modélisé. L'architecture interne du module mémoire n'est pas modélisée. De ce fait, les latences dues au rafraichissement des cellules mémoires qui ont été accédées ne sont pas reproduites. Cette limitation a pour effet de ne pas reproduire la baisse de performance des modules mémoires observée lors de leur usage réel.

L'outil de simulation utilise une horloge pour cadencer l'activité du FPGA modélisé. Cette horloge est générée par le moteur de simulation qui est exécuté par un processus client. Cette topologie présente l'avantage d'être fidèle au modèle d'intégration d'accélérateurs reconfigurables dans une MVP présenté dans la section 4.2, mais limite la capacité de simulation à un seul FPGA. Pour simuler plusieurs FPGA, il faudrait instancier plusieurs moteurs de simulation exécutés par différents processus client. De ce fait, les horloges des moteurs de simulation ne seraient pas synchronisées. Du point de vue de l'activité interne des FPGA, cela ne serait pas un problème. Mais du point de vue de la chronologie de l'émission des requêtes, ce modèle de simulation avec plusieurs FPGA ne serait pas fiable.

Les limitations de cet outil de simulation ouvrent la voie à plusieurs perspectives d'évolution. En tirant partie de la programmation orientée objet, il pourrait être possible d'évaluer automatiquement à gros grain l'usage des res-

sources du FPGA. Les documentations fournies avec les suites logicielles pour la conception sur FPGA donnent des informations sur l'utilisation des ressources pour l'instanciation de certains composants. Par exemple, une FIFO peut être assimilée à l'utilisation d'une BRAM dont la taille est définie par la profondeur de la FIFO. De manière similaire, les opérateurs arithmétiques utilisés par les noyaux de calcul peuvent être assimilés à l'utilisation d'un DSP.

Une deuxième perspective d'évolution de l'outil de simulation serait de permettre la simulation parallèle de plusieurs FPGA. Pour ce faire, il faudrait synchroniser les horloges des différents moteurs de simulation. Cette synchronisation peut être techniquement réalisable, mais elle risquerait de provoquer un grand ralentissement de la vitesse de simulation. Pour être efficace, il faudrait que la synchronisation des horloges soit réalisée seulement lorsque des requêtes d'accès aux données distantes sont émises.

6.4 Conclusion

Ce chapitre présente un outil de simulation développé pour répondre aux besoins de cette thèse et qui a fait l'objet d'une publication [74]. Il a pour objectif de permettre la modélisation rapide d'un prototype permettant de valider le comportement du modèle d'intégration d'accélérateurs reconfigurables dans une MVP présenté dans la section 4.2 et de permettre d'expérimenter le modèle de programmation exposé dans le chapitre 5. Cet outil s'appuie sur une modélisation haut niveau utilisant le paradigme de programmation orientée objet. Il emploie une méthode de simulation hybride où seule l'activité du FPGA est reproduite avec un moteur de simulation et où la partie logicielle du système étudié est réellement exécutée. Cette méthode présente plusieurs avantages :

- la modélisation des noyaux se fait à partir d'une description haut niveau qui est rapide à réaliser.
- les latences des requêtes d'accès aux données distantes sont produites avec précision, car elles sont mesurées à partir de l'exécution réelle du logiciel.
- l'exécution parallèle de la MVP permet de faire des allocations mémoire dynamiques.
- l'utilisation de l'architecture physique pour exécuter la partie logicielle

permet une simulation rapide, car elle ne nécessite pas de simuler la plateforme matérielle.

— l'utilisation du système SAT comme support d'exécution permet de répartir les processus de l'outil sur différents nœuds afin d'étudier les latences d'accès aux données distantes liées à différentes topologies de systèmes de calcul.

Cet outil a permis de mener des explorations pour évaluer les propositions faites dans cette thèse plus rapidement qu'en utilisant les outils de synthèse FPGA. Néanmoins, il présente plusieurs limitations. De ce fait, il doit être vu comme une solution pour mener des explorations dont le but est d'affiner la spécification du modèle d'intégration d'accélérateurs reconfigurables dans une MVP et de faciliter le développement du modèle de programmation en l'expérimentant.

Troisième partie

Validation du modèle de programmation

Introducrion

Une MVP logicielle peut induire des latences d'accès aux données élevées pour plusieurs raisons. Premièrement, un système de mémoires distribuées conduit à des effets d'accès mémoire non uniformes (NUMA). Le transfert des données entre la mémoire, qui en contient une version à jour, et l'unité de traitement, qui en a demandé l'accès, nécessite un temps incompressible dépendant des performances du système d'interconnexion et de la localisation des données. Deuxièmement, le protocole de cohérence et les services de localisation et d'orchestration des transferts de données sont réalisés par du logiciel qui nécessite du temps pour s'exécuter venant s'ajouter au temps de transfert des données. Le modèle d'intégration d'accélérateurs reconfigurables dans une MVP logicielle que nous proposons repose sur un système de proxy qui exacerbe les effets NUMA. Un temps d'accès long aux données est un frein à la capacité d'en accélérer le traitement. Le modèle de programmation que nous proposons a pour objectif de répondre à ce problème en masquant les latences.

Pour valider ce modèle de programmation, nous avons mené des expériences ayant pour objectif d'évaluer sa capacité à masquer efficacement les latences. Les expérimentations ont été menées à l'aide de l'outil de simulation introduit dans le chapitre 6. Deux cas d'études ont été utilisés : la multiplication de matrices creuses et un code de simulation de tsunami.

La multiplication de matrices creuses est une opération d'algèbre linéaire creuse fréquemment étudiée car elle génère des accès mémoires aléatoires et dépendants des données. Elle nous a permis d'expérimenter l'utilisation du format de compression de matrices creuses sous forme de *chunks* proposé dans le chapitre 5.

Le code de simulation de tsunami étudié est employé par le système germano-indonésien de détection et d'alerte rapide aux tsunamis (*German Indonesian Tsunami Early-Warning System (GITEWS)*). Ce code utilise comme struc-

ture de données de base un maillage non structuré bidimensionnel, ce qui nous a permis d'expérimenter les travaux sur la localité des données et la décomposition de maillages non structurés sous forme de *chunks* proposés dans le chapitre 5.

Cette partie du manuscrit s'organise de la manière suivante : le chapitre 7 présente la plateforme matérielle utilisée pour la simulation et la méthodologie employée pour mener les expérimentations, le chapitre 8 expose le cas d'étude portant sur la multiplication de matrices creuses, le chapitre 9 présente le cas d'étude portant sur le code simulation de tsunami et enfin une discussion est présentée en page 193.

Chapitre 7

Plateforme de simulation et méthodologie d'expérimentations

Pour expérimenter l'utilisation du modèle de programmation et pour en évaluer les performances, nous avons conçu et implémenté un outil de simulation et de modélisation, décrit dans le chapitre 6, qui a fait l'objet d'une présentation lors du *Workshop on Rapid System Prototyping* [74]. Cet outil emploie une méthode de simulation hybride qui consiste à générer les latences d'accès avec la MVP logicielle réellement exécutée sur une architecture physique et à reproduire le comportement des noyaux de calcul avec un moteur de simulation. Cette méthode hybride permet deux choses. Premièrement, de générer des latences fidèles à la réalité. Deuxièmement, d'exécuter facilement l'outil de simulation sur un système distribué afin d'étudier différents profils de latences. Les latences générées dépendent de la plateforme matérielle sur laquelle est exécuté le logiciel, parce que les processus de la MVP s'exécutent à la vitesse permise par les processeurs et que les latences de leurs communications sont dépendantes des performances du réseau d'interconnexion.

Afin d'améliorer la compréhension des données obtenues lors des expérimentations, la section 7.1 décrit les caractéristiques de la plateforme matérielle utilisée pour exécuter l'outil de simulation. La section 7.2 présente la méthodologie adoptée pour conduire les expérimentations. La section 7.3 expose une discussion sur les limitations de la méthodologie utilisée pour mener les expérimentations.

7.1 Caractérisation de la plateforme de simulation

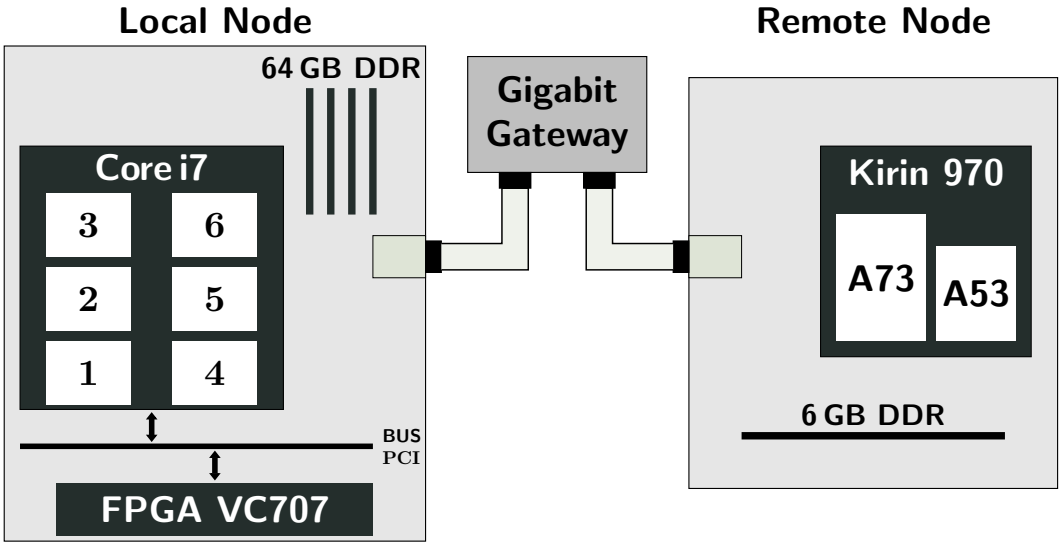


FIGURE 7.1 – Représentation de la plateforme matérielle utilisée pour exécuter l'outil de simulation. Le nœud local intègre un processeur Intel Core i7-6800K, un FPGA Xilinx VC707 connecté par un port PCI et 64 Go de mémoire vive. Le nœud distant est une carte de développement Hikey 970 intégrant un système sur puce Kirin 970 et 6 Go de mémoire vive. Les deux nœuds communiquent par un réseau Gigabit ethernet via une passerelle.

La plateforme matérielle que nous avons utilisée se compose de deux nœuds de calcul, un nœud dit local et un nœud dit distant, telle que représentée sur la figure 7.1. Chaque nœud correspond à une carte intégrant un processeur et de la mémoire vive. Les deux nœuds sont reliés par un réseau ethernet Gigabit via un commutateur réseau.

Le nœud local intègre un processeur Intel Core i7-6800K cadencé à 3.4 GHz, 64 Go de mémoire vive et une carte FPGA. Cette dernière est une carte d'évaluation Virtex-7 VC707 de Xilinx. Elle est physiquement reliée au nœud par un bus *Peripheral Component Interconnect* (PCI) Express de deuxième génération à 8 lignes.

Le nœud distant est une carte de développement Hikey 970 intégrant 6 Go de mémoire vive et un système sur puce Kirin 970, dont le processeur est une architecture ARM *big.LITTLE* composée d'un Cortex-A73 fonctionnant

jusqu'à 2.36 GHz et d'un Cortex-A53 fonctionnant jusqu'à 1.8 GHz. Cette architecture de processeur est proche de celles de certains processeurs embarqués dans des systèmes sur puces FPGA, tels que ceux de la gamme *Zynq* de Xilinx ou de la gamme *Agilex* de Intel. Le nœud distant est relié au réseau ethernet par un adaptateur USB 3.0.

Lors des expérimentations, nous avons utilisé le nœud local pour modéliser un système où le FPGA communique avec l'environnement logiciel de la MVP par l'intermédiaire d'un bus PCI. La sous-section 7.1.1 présente la caractérisation des latences de communication aller-retour entre le processeur du nœud local et le FPGA. Le nœud distant a été utilisé pour modéliser un système où le FPGA embarque un processeur qui exécute un processus de la MVP. La sous-section 7.1.2 caractérise les latences des requêtes de la MVP en fonction de la topologie du système.

7.1.1 Caractérisation des latences de communication par bus PCI entre le FPGA et le processeur

Nous avons utilisé la carte d'évaluation Virtex-7 VC707 dans des travaux exploratoires ayant pour objectif de faire communiquer un processus logiciel avec la logique programmable de la carte FPGA. Ces travaux nous ont permis de mesurer les latences d'une communication aller-retour entre le processeur et le FPGA. Pour ce faire, sur le FPGA, nous avons instancié l'IP matérielle DMA/Bridge Subsystem for PCI Express [117] fournie dans la suite logicielle Vivado qui permet de contrôler le port PCI et d'effectuer des accès direct à la mémoire (*Direct Memory Access* (DMA)). Sur le processeur, nous avons utilisé le pilote *XDMA* pour système Linux fournit par Xilinx. Ce pilote permet de programmer l'IP matérielle. L'expérimentation a consisté à réaliser un échange aller-retour de paquets de données dont les tailles varient de 8 octets à 8 kio.

Les résultats des mesures réalisées avec ce système sont présentées sur la figure 7.2. Chaque mesure a été reproduite 40 fois pour tenir compte de l'activité parallèle du système de la plateforme matérielle. Les résultats indiquent que la latence de communication moyenne est de l'ordre de la vingtaine de microsecondes. Entre 8 octets et 1 kio, la latence moyenne est d'environ 21 µs. Au-delà de 1 kio, la latence croit en fonction de la taille des paquets pour atteindre environ 24 µs.

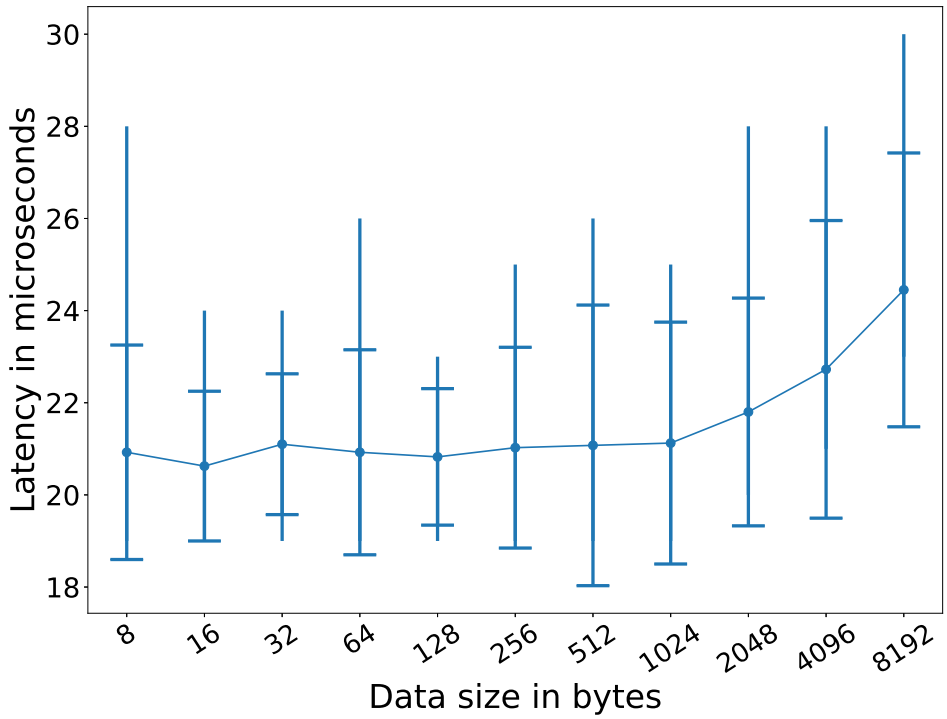


FIGURE 7.2 – Graphique des latences de communication aller-retour entre le FPGA et le processeur en fonction de la taille des paquets de données. Les points sur la ligne indiquent les valeurs moyennes des latences mesurées. Les extrémités des barres verticales indiquent les latences maximales et minimales mesurées. Les traits horizontaux sur chaque barre indiquent la variance des 40 mesures.

7.1.2 Caractérisation des latences des requêtes d'accès à la MVP logicielle

L'outil de simulation utilise la MVP logicielle comme support d'exécution. Celle-ci requiert une description de la topologie de l'application, c'est-à-dire une description du rôle de chaque processus et de leurs liens de connexion, tel que décrit dans la section 4.1. Pour les deux cas d'études, nous avons utilisé la même topologie se composant de deux processus clients reliés à un seul processus serveur. Un premier processus client est en charge des entrées-sorties. Il réalise l'acquisition et l'allocation dans la mémoire partagée pour les données d'entrée et il restitue les données de sortie. Un second processus client exécute le moteur de simulation qui est en charge de l'allocation des données de sortie. Le serveur gère les données partagées entre les deux processus clients.

Trois topologies de système, correspondant à des profils de latences différents, ont été utilisées pour mener les expérimentations : *No Latency*, *Local* et *Remote*. La topologie *No Latency* correspond au cas idéal, où l'accès aux données distantes ne requiert aucun temps. La topologie *Local* est utilisée pour modéliser un système où le FPGA est intégré au nœud par un bus PCI sur lequel sont localisées les données partagées et le serveur en charge de leur gestion. Dans ce cas, les trois processus de l'outil de simulation sont exécutés sur le nœud local. La topologie *Remote* correspond au cas où le FPGA est intégré sur un nœud distant de celui qui possède les données et qui exécute le processus serveur. Dans ce cas, le processus serveur et le processus d'entrées-sorties sont exécutés sur le nœud local et le moteur de simulation est exécuté sur le nœud distant. La différence de performance du nœud local et du nœud distant et la distribution des données permettent de faire apparaître nettement deux profils de latences.

La figure 7.3 représente les latences des requêtes de lecture et d'écriture mesurées sur les deux nœuds en faisant varier la taille des blocs de données de 8 octets à 8 kio. Pour réaliser ces mesures, nous avons utilisé la même topologie d'application que celle utilisée par l'outil de simulation : un premier processus client alloue les données, un deuxième processus client accède aux données et un processus serveur gère les données. Comme pour les mesures des latences de communication avec le FPGA, chaque mesure a été reproduite 40 fois. Les résultats montrent que les latences sont environ trois fois plus importantes pour le nœud distant. Nous pouvons également observer que les requêtes de lecture nécessitent environ deux fois plus de temps que les requêtes d'écriture pour être complétées. Cette différence s'explique par le protocole de cohérence qui ne requiert pas les mêmes actions pour les deux types de requêtes. Lors des simulations avec la topologie *Local*, les latences de communication du bus PCI sont ajoutées aux latences mesurées avec la MVP logicielle.

7.2 Méthodologie des expérimentations et calibration du modèle

Pour les expérimentations, nous avons utilisé les caractéristiques du FPGA VC707 [116] pour configurer l'outil de simulation. Ce choix a été motivé par son utilisation pour mesurer les latences des communications avec le bus PCI et par

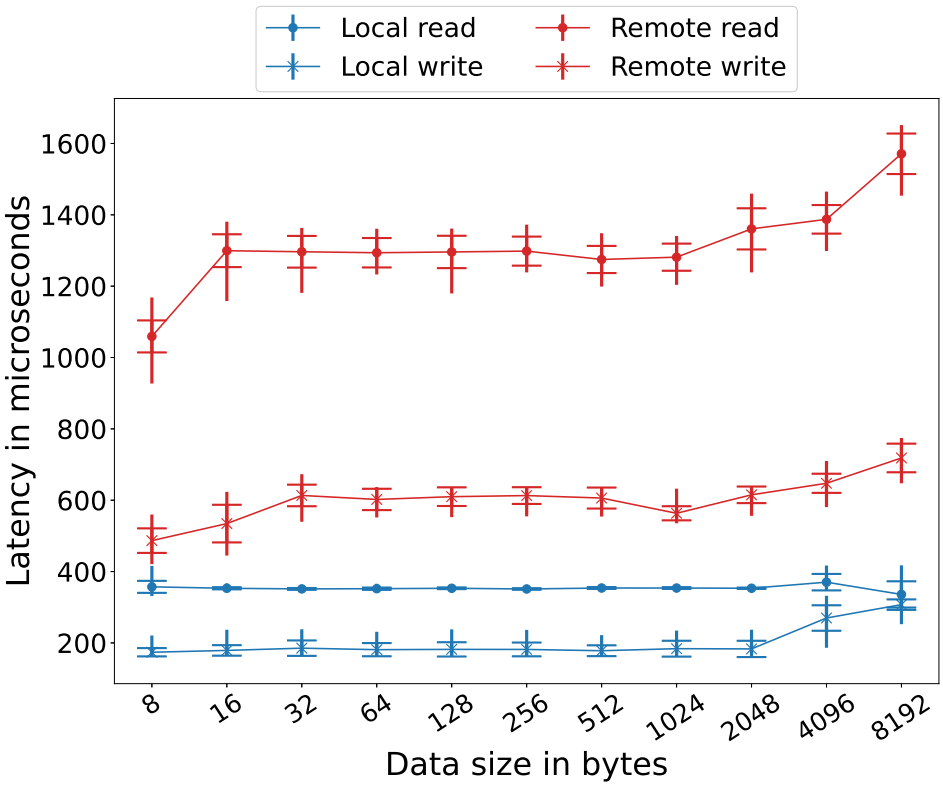


FIGURE 7.3 – Graphique des latences des requêtes de la MVP logicielle sur le nœud local et sur le nœud distant en fonction de la taille des paquets de données. Les points sur les lignes indiquent les valeurs moyennes des latences mesurées. Les extrémités des barres verticales indiquent les latences maximales et minimales mesurées. Les traits horizontaux sur chaque barre indiquent la variance des 40 mesures.

la connaissance de son architecture acquise lors de travaux exploratoires. Ainsi, le moteur de simulation utilise une horloge cadencée à 200 MHz. La largeur du bus de données interne, qui permet de relier le contrôleur mémoire et les noyaux de calcul, est définie par les caractéristiques du module mémoire du FPGA modélisé. Il s'agit d'un module mémoire dynamique synchrone à accès aléatoire (*Synchronous Dynamic Random Access Memory* (SDRAM)) DDR3 qui a une capacité de 1 Go, une largeur de chemin de données de 64 bits et qui peut réaliser jusqu'à 1.6 GT/s. Ainsi, en tenant compte de la fréquence de l'horloge, la largeur du bus de données interne a été fixée à 512 bits. De cette manière, la bande passante maximale du contrôleur mémoire est de 12.8 Go/s.

L'outil de simulation utilise les mesures des latences des requêtes de la MVP pour contraindre le comportement du moteur de simulation. Comme

le montre la figure 7.3, l'activité du système sur lequel est exécutée la MVP peut perturber les mesures des latences. Ces variations peuvent impacter de manière non uniforme la simulation. Par exemple, si les perturbations se produisent au lancement de l'exécution, l'ensemble de la simulation sera impacté. De plus, la simulation est un processus lent, avec cet outil 1 seconde de temps simulé nécessite entre 300 et 1000 secondes de temps de simulation. Le risque de perturbations des mesures pendant l'exécution est grand. De ce fait, les expérimentations portant sur les deux cas d'études ont été répétées 10 fois. Pour chaque expérimentation, les résultats présentés sont les valeurs médianes des 10 exécutions. Pour limiter les risques d'erreurs de mesures ou de manipulations de données, l'ensemble du processus d'expérimentation, du lancement de l'outil de simulation à la génération des figures, a été réalisé à l'aide de scripts.

7.3 Limitations de la méthodologie d'expérimentations

La méthodologie d'expérimentation a pour objectif majeur d'évaluer la capacité du modèle de programmation à masquer les latences d'accès aux données distances réalisées par un noyau de calcul exécuté sur un FPGA. Cette méthodologie repose sur l'utilisation d'un outil de simulation que nous avons conçu. De ce fait, la méthodologie d'expérimentation est sujette aux limitations de l'outil de simulation. Cet outil permet de modéliser rapidement un noyau de calcul à partir d'une description haut niveau et d'en évaluer les performances avec différentes topologies de systèmes. Néanmoins, l'utilisation de cet outil implique certaines limitations.

Premièrement, une modélisation haut niveau d'un noyau de calcul est moins précise qu'une modélisation RTL obtenue à l'aide d'une description avec un langage HDL.

Deuxièmement, la conception de l'outil de simulation s'est limitée à la modélisation d'un contrôleur pour un module mémoire SDRAM possédant un seul chemin de données. L'émergence des cartes FPGA intégrant des systèmes mémoire à haute bande passante (HBM) utilise un empilement de circuits mémoires. Les mémoires HBM offrent plusieurs chemins de données dans un même module (SDRAM). Ainsi, l'outil de simulation ne permet pas de modé-

liser des cartes FPGA intégrant une mémoire HBM. La modélisation de ces mémoires est une perspective d'évolution de l'outil de simulation.

Enfin, la simulation de l'activité de plusieurs FPGA n'est pas prise en charge par l'outil. Cette fonctionnalité nécessiterait de synchroniser les horloges des différents moteurs de simulation. La prise en charge de cette fonctionnalité est une deuxième perspective d'évolution de l'outil de simulation.

Chapitre 8

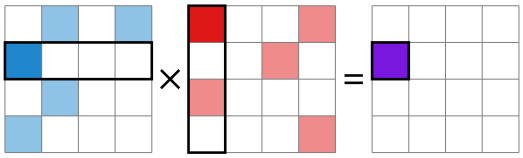
Cas d'étude 1 : Multiplication de matrices creuses

Le premier cas d'étude utilisé pour valider le modèle de programmation est la multiplication de matrices creuses, une opération fréquemment employée dans les applications HPC tels que les solveurs linéaires ou les applications d'analyse de graphes. Cette opération consiste à calculer $C = A \times B$, où les deux matrices sources A et B sont creuses et respectivement de dimensions $M \times N$ et $N \times P$. La matrice C est de dimensions $M \times P$ et a une densité dépendant du motif de A et B. La sporadicité des structures de données rend l'optimisation de ce noyau de calcul complexe et amène plusieurs problèmes. Le nombre d'éléments non nuls de la matrice C ne peut pas être connu sans avoir parcouru les deux matrices sources. Ceci oblige à adopter une stratégie d'allocation pouvant être dynamique, en ajustant l'espace mémoire au fil de l'exécution, ou pouvant reposer sur un prétraitement. L'intensité FLOP/accès mémoire est également variable et généralement faible, ce qui rend l'opération *memory-bound*. Enfin, la multiplication de matrices creuses génère des motifs d'accès mémoire aléatoires et dépendants des données.

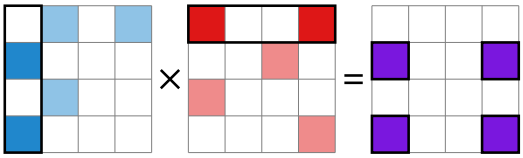
Le parti pris des travaux de cette thèse est que l'utilisation de la MVP peut améliorer la gestion des données en apportant un support dynamique d'allocation mémoire et en permettant d'accéder aléatoirement aux données distantes avec une granularité fine. L'intérêt de ce cas d'étude est d'évaluer la capacité du modèle de programmation à répondre aux besoins multiples de ce noyau de calcul et d'expérimenter l'utilisation du format de compression de matrices creuses sous forme de *chunks* dans un contexte où les motifs d'accès mémoires

ne sont pas facilement prévisibles. Le chapitre s'organise de la façon suivante : la section 8.1 décrit l'architecture du noyau de calcul modélisé, la section 8.2 présente les jeux de données utilisés pour mener les expérimentations et la section 8.3 expose et commente les résultats des expérimentations.

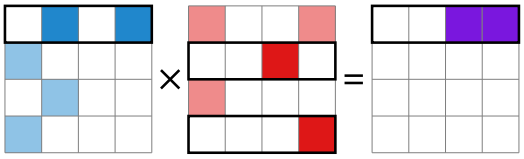
8.1 Modélisation du noyau de calcul



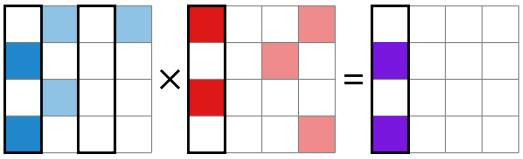
(a) Produit intérieur : Pour chaque élément de C, une rangée de A est multipliée à une colonne de B. Les indices de colonnes des éléments non nuls de la rangée de A doivent être comparés aux indices de rangées des éléments non nuls de la colonne de B.



(b) Produit extérieur : Chaque rangée de A est multipliée à la colonne de B de même indice. Les résultats partiels sont produits de manière désordonnée.



(c) Parcours par rangées : Chaque élément d'une rangée de A est multiplié par la rangée de B correspondant à son indice de colonne. Le traitement d'une rangée de A constitue le résultat d'une rangée de C.



(d) Parcours par colonnes : Chaque élément d'une colonne de B est multiplié par la colonne de A correspondant à son indice de rangée. Le traitement d'une colonne de B constitue le résultat d'une colonne de C.

FIGURE 8.1 – Représentation des quatre flots de données de la multiplication de matrices creuses.

En considérant que le parcours d'une matrice creuse peut se faire, de manière intuitive, soit dans l'ordre des rangées, soit dans l'ordre des colonnes, le flot de données de la multiplication de matrices creuses se décline en quatre versions : le produit intérieur, le produit extérieur, le parcours par rangées et le parcours par colonnes. La figure 8.1 représente ces quatre flots de données.

$$C[m,p] = \sum_{n=0}^{N} A[m,n] \times B[n,p] \tag{8.1}$$

Le produit intérieur, tel que décrit par l'équation 8.1, consiste à multiplier une rangée de A par une colonne de B pour produire un élément de C. Cette stratégie facilite la parallélisation des calculs, parce que chaque élément de C est produit de manière indépendante, mais est inefficace au regard de l'utilisation des données. En effet, le produit intérieur nécessite de faire correspondre les indices de colonnes des éléments de A avec les indices de rangées des éléments de B pour produire un résultat partiel. Plus les vecteurs de données sont creux, plus, d'un point de vue statistique, cette stratégie conduit à accéder à des données non utiles aux calculs. La multiplication de matrices creuses étant une opération *memory-bound*, un faible ratio d'utilisation des données conduit à faire chuter de manière mécanique les performances. Enfin, cette stratégie requiert l'utilisation de deux formats de stockage différents pour les matrices A et B, ce qui pourrait obliger à transformer leurs formats dans un contexte applicatif.

$$C[:,:] = \sum_{n=0}^{N} A[:,n] \times B[n,:] \tag{8.2}$$

Le produit extérieur, tel que formulé par l'équation 8.2, emploie la stratégie inverse du produit intérieur en accédant à la matrice A par colonnes et à la matrice B par rangées. Contrairement au produit intérieur, cette stratégie favorise la localité des accès aux matrices sources au détriment des accès à la matrice résultat. Du point de vue de l'utilisation des données, le produit extérieur est efficace, parce qu'il ne nécessite pas d'effectuer de comparaison d'indices. En considérant que les matrices ne possèdent pas de rangées vides, l'ensemble des données accédées sont utilisées. L'algorithme permet de réaliser l'opération en accédant séquentiellement une seule fois, à chaque rangée de A et, une seule fois, à chaque colonne de B. Cependant, les accès à la matrice C sont désordonnés, ce qui complexifie la parallélisation des calculs parce que l'accumulation des résultats partiels nécessite une synchronisation. Cette stratégie complexifie aussi la gestion du stockage des résultats partiels. De plus, comme pour le produit intérieur, elle requiert l'utilisation de deux formats de stockage différents pour les matrices A et B.

L'algorithme de Gustavson [59] décrit par l'algorithme 2, réalise le produit matriciel en parcourant les matrices par rangées. Les accès aux éléments de la matrice A sont séquentiels et dans l'ordre des rangées. chacun est multiplié par

Algorithme 2 Algorithme de Gustavson : multiplication de deux matrices creuses avec un parcours par rangées.

Input: $A \in \mathbb{R}^{M \times N}$, $B \in \mathbb{R}^{N \times P}$
Output: $C \in \mathbb{R}^{M \times P}$

```
for m = 0,...,M-1 do
    For Each A_{m,n} in row A_{m,*} do
        For Each B_{n,p} in row B_{n,*} do
            if C_{m,p} == 0 then
                C_{m,p} = A_{m,n} × B_{n,p}
            else
                C_{m,p} += A_{m,n} × B_{n,p}
            end if
        end for
    end for
end for
```

la rangée de B correspondant à son indice de colonne. Le traitement d'une rangée de A constitue l'ensemble des résultats de la rangée de C de même indice. Seuls les accès aux rangées de B sont aléatoires. Cet algorithme est facile à paralléliser puisque le traitement de chaque rangée est indépendant. L'algorithme de Gustavson permet d'utiliser le même format de stockage pour les trois matrices. Il peut être décliné avec un parcours par colonnes. GAMMA [121] et MatRaptor [103] sont deux accélérateurs matériels pour la multiplication de matrices creuses utilisant l'algorithme de Gustavson. Ces deux accélérateurs sont les plus performants de l'état de l'art, notamment lorsque les matrices sont très creuses.

Pour modéliser le noyau de calcul, nous avons choisi d'utiliser l'algorithme de Gustavson qui répond à plusieurs objectifs. Premièrement, un accès par rangées est la manière la plus adaptée de traiter l'opération avec le format de compression de matrices creuses que nous avons proposé dans le chapitre 5 et qui est représenté sur la figure 8.2. Ensuite, au vu de l'état de l'art, l'algorithme de Gustavson semble être le plus adapté à l'accélération matérielle du noyau de calcul. Enfin, il rend les accès à la matrice B fortement aléatoires et dépendants des données, ce qui nous est utile pour évaluer le modèle de programmation.

La figure 8.3 représente l'architecture du noyau de calcul modélisé. Il est implémenté sous la forme d'un pipeline de 8 niveaux d'étages séparés par des files (FIFO). Chaque étage peut s'exécuter à chaque cycle si la FIFO à son entrée n'est pas vide et que la FIFO à sa sortie n'est pas pleine. Plusieurs

(a) Format dense. (b) Format CSR sous forme de *chunks*.

FIGURE 8.2 – Représentation du format de compression par rangée d'une matrice dense en *chunks*.

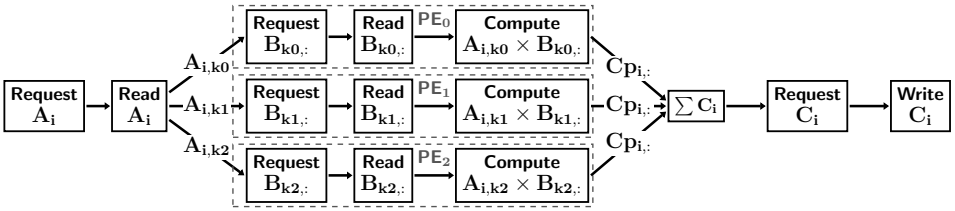


FIGURE 8.3 – Architecture du noyau de calcul de multiplication de matrices creuses. Chaque rectangle en pointillés représente une unité de traitement (*Processing Element* (PE)). Les flèches représentent des FIFO.

unités de traitement (*Processing Element* (PE)) peuvent être instanciées pour paralléliser le traitement au sein du noyau. Le premier étage émet les requêtes d'accès aux rangées de A. Lorsqu'une requête est acceptée, l'indice de *chunk* de la rangée est écrit dans la FIFO à la sortie de l'étage. La profondeur de cette FIFO définit la profondeur du préchargement. Plus elle est grande, plus le nombre de *chunks* préchargés est grand. Pour qu'une requête soit acceptée, la mémoire du FPGA doit disposer d'un emplacement pour stocker le *chunk*. Le deuxième étage lit les rangées de A préchargées et distribue les données aux différentes PE. Chacune est composée de trois étages et traite les rangées qui lui sont attribuées séquentiellement. Le premier émet les requêtes d'accès aux rangées de la matrice B en fonction des indices de colonnes des éléments de la matrice A qui lui ont été attribués. Comme pour le premier étage du noyau, les requêtes sont acceptées si la mémoire du FPGA dispose d'emplacements pour accueillir les données et les indices des *chunks* préchargés sont écrits dans une FIFO en sortie d'étage. La taille de celle-ci est la même que celle en sortie du premier étage du noyau. Le deuxième étage de chaque PE lit les données préchargées et le troisième étage calcule les résultats partiels. Les trois derniers étages du noyau de calcul servent à accumuler les résultats partiels et à écrire les rangées de la matrice C. Ainsi, le sixième niveau d'étage reçoit les

résultats partiels de toutes les PE et en fait l'accumulation. Lorsque tous les calculs d'une rangée de C sont réalisés, la taille et l'indice de la rangée sont transmis à l'avant-dernier étage qui émet la requête d'écriture. Étant donné que l'algorithme de Gustvason ne requiert pas de synchroniser les écritures du résultat, cette requête consiste à demander l'allocation d'un emplacement dans la mémoire du FPGA. La latence de la requête d'écriture est appliquée pour la libération de l'emplacement. Le dernier étage procède aux transferts des données entre le noyau de calcul et la mémoire du FPGA. Le noyau de calcul utilise un premier flux de données pour lire la matrice A, un deuxième pour écrire la matrice C et un flux par PE pour lire les rangées de la matrice B. Ainsi, en considérant npe le nombre d'unités de traitement instanciées, alors le noyau utilise $2 + npe$ flux de données en parallèle.

Le Département d'Informatique de l'Université Rutgers a proposé REAP [99], une approche pour traiter des multiplications de matrices creuses sur une plateforme CPU-FPGA présentant plusieurs points de ressemblance avec les travaux menés dans cette thèse. Premièrement, l'architecture du noyau de calcul de REAP repose sur l'algorithme de Gustavson. Deuxièmement, REAP utilise un format de compression des matrices creuses dérivé du format CSR. Comme dans nos travaux, l'indice de colonne et la valeur de chaque élément non nul sont colocalisés pour augmenter la localité des données et le nombre d'éléments non nuls contenus dans une rangée est mémorisé dans des métadonnées. Cependant, REAP repose sur la supervision du CPU pour distribuer les données au noyau de calcul. De cette manière, le FPGA reste esclave du système. De plus, REAP n'apporte pas de support pour la gestion de données distribuées. À l'inverse, notre approche permet à l'accélérateur d'initier l'accès aux données distribuées.

8.2 Jeux de matrices creuses utilisés pour les expérimentations

Pour mener les expérimentations, nous avons constitué deux jeux de matrices à partir de la collection de matrices creuses de l'Université de Floride [37]. Cette collection propose un large choix de matrices issues de cas applicatifs réels et dont les différentes caractéristiques varient. Elle est fréquemment utilisée pour évaluer les travaux portant sur l'optimisation d'algorithmes traitant

des matrices creuses car son accès est public, ce qui facilite la reproductibilité et la comparaison des expériences. La multiplication nécessite que le nombre de colonnes de la matrice A soit égal au nombre de rangées de la matrice B. Parmi la collection, le choix de matrices distinctes satisfaisant cette propriété est limité aux matrices de petites tailles. Comme dans l'ensemble des travaux utilisant cette collection et portant sur la parallélisation du noyau de calcul étudié, nous avons été contraint d'utiliser la même matrice pour les deux opérandes sources. Cependant, lors des expérimentations, les deux opérandes sources ont fait l'objet d'allocations mémoire séparées.

L'utilisation d'un outil de simulation et la répétition des mesures conduisent à un temps d'expérimentation long. Pour réduire ce temps, nous avons utilisé un jeu de matrices de petites dimensions. Pour corroborer les résultats obtenus avec ce premier jeu, nous avons utilisé un second ensemble de matrices dont les dimensions sont plus grandes. Le tableau 8.1 présente les caractéristiques de ces deux jeux. La multiplication de matrices creuses ayant une intensité arithmétique et des motifs d'accès mémoire dépendants des données, la sélection des matrices a été faite de sorte à pouvoir évaluer la capacité du modèle de programmation à s'adapter à l'irrégularité des structures de données. Plus une matrice est creuse, plus son intensité arithmétique est faible. Une intensité arithmétique faible conduit à un temps de traitement court. Ainsi, pour le modèle de programmation, une faible intensité arithmétique limite la capacité de superposition temporelle des flux de données transférés avec les flux de données traités ce qui limite la capacité à masquer les latences d'accès aux données distantes. Le motif de la matrice influe sur la localité des accès et la réutilisation des données du second opérande. Une matrice creuse dont ses éléments non nuls sont regroupés le long de la diagonale peut former une bande. Une matrice dont la largeur de bande est petite conduit à une forte localité temporelle des accès aux rangées de la matrice B. Plus le nombre d'éléments non nuls contenus dans une rangée est important, plus la rangée de B de même indice sera accédée. La dernière colonne du tableau 8.1 indique la vitesse de calcul maximale théorique pour chaque matrice. Cette vitesse est définie en considérant le temps de traitement comme étant le temps nécessaire pour réaliser tous les accès mémoire du noyau de calcul en fonction de la bande passante de la mémoire (12.8 Go/s).

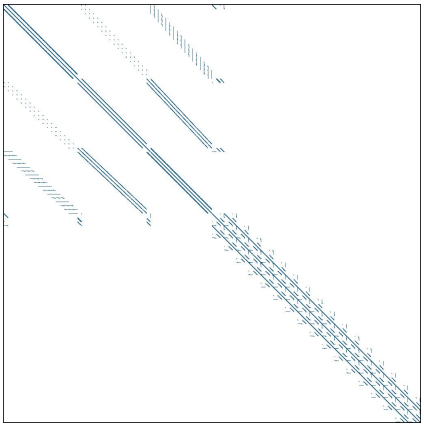
La figure 8.4 représente les motifs du jeu de petites matrices. La matrice `cop20k_A` se présente comme la plus complexe à accélérer avec le modèle de

Matrice	M	A(NNZ)	A(Densité)	GFLOP	Empreinte mémoire	Perf. de pointe en GFLOP/s
Petites matrices						
`F2`	71 505	5 294 285	0.10 %	0.99	383 Mo	2.93
`consph`	83 334	6 010 480	0.087 %	0.93	294 Mo	2.99
`s3dkt3m2`	90 449	3 753 461	0.046 %	0.31	134 Mo	2.93
`m_t1`	97 578	9 753 570	0.10 %	2.1	427 Mo	3.07
`cop20k_A`	121 192	2 624 331	0.018 %	0.16	182 Mo	2.52
Grandes matrices						
`SiO2`	155 331	5 719 417	0.047 %	28.5	1.02 Go	3.17
`F1`	343 791	13 590 452	0.023 %	5.3	2.23 Go	2.92
`ML_Laplace`	377 002	27 689 972	0.020 %	4.1	1.21 Go	3.02
`RM07R`	381 689	37 464 962	0.026 %	10.5	2.15 Go	3.07
`inline_1`	503 712	18 660 027	0.015 %	6.7	2.61 Go	2.94
`ldoor`	952 203	23 737 339	0.0051 %	4.8	1.90 Go	2.96

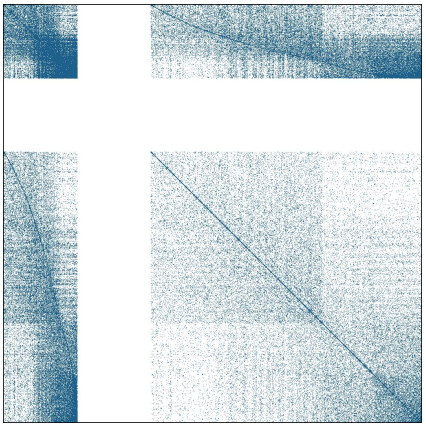
Tableau 8.1 – Caractéristiques des jeux de matrices carrées utilisés pour les expérimentations. M désigne le nombre de rangées des matrices. A(NNZ) et A(Densité) font référence au nombre d'éléments non nuls et à la densité de la matrice source. Gflop équivaut au nombre de milliards d'opérations flottantes, additions et multiplication cumulées, nécessaire pour multiplier les matrices. L'empreinte mémoire correspond à la taille nécessaire pour stocker les éléments non nuls de deux fois la matrice source et d'une fois la matrice résultat, en considérant des mots de 4 octets. Le calcul de l'empreinte mémoire exclut les indices et les nombres d'éléments non nuls des rangées.

programmation pour deux raisons. Premièrement, sa faible densité (0.018 %) offre la plus faible charge de travail (0.16 GFLOP). Deuxièmement, sa largeur de bande, qui est égale à la largeur de la matrice, permet une faible localité temporelle des accès à la matrice B. Au contraire, la matrice `m_t1`, dont la charge de travail est la plus élevée (2.1 GFLOP) parmi le jeu de petites matrices et dont la largeur de bande est petite, est plus propice au modèle de programmation.

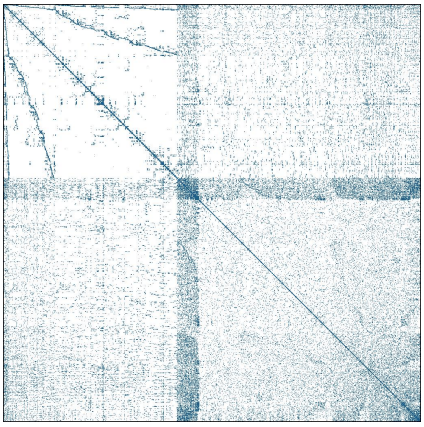
De la même manière, la figure 8.5 représente les motifs du jeu de grandes matrices. La matrice `ldoor` se présente comme la plus complexe à accélérer parmi ce jeu. Elle a une faible charge de travail (4.8 GFLOP) par rapport à sa dimension et une grande largeur de bande. De plus, le motif irrégulier de la matrice laisse apparaitre des zones à plus faible densité, conduisant à une répartition inégale de la charge de travail. Au contraire, la matrice `SiO2` semble être la plus favorable au modèle de programmation grâce à sa charge de travail plus élevée (28.5 GFLOP) et à son étroite largeur de bande.



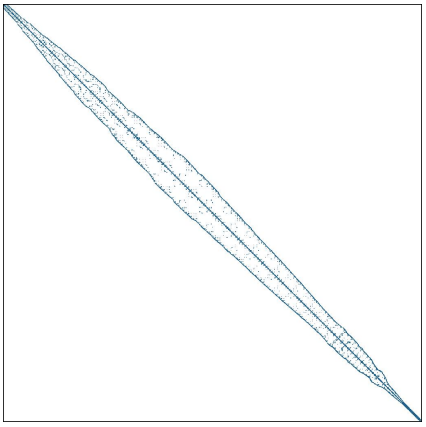
(a) Matrice `consph`.



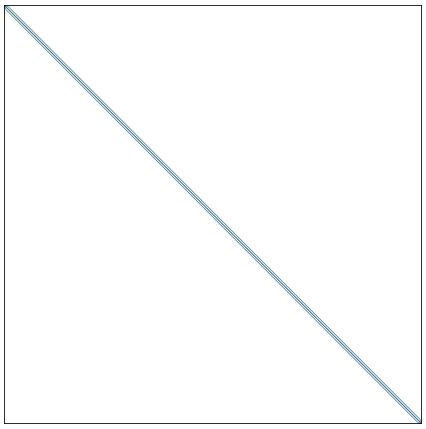
(b) Matrice `cop20k_A`.



(c) Matrice `F2`.

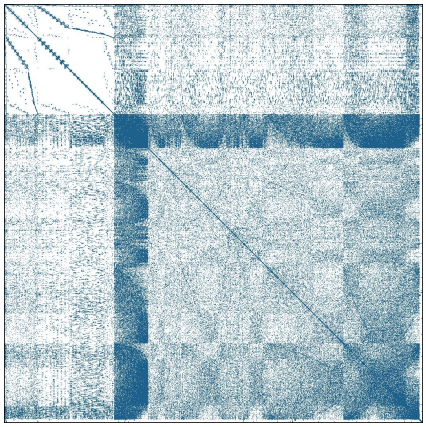


(d) Matrice `m_t1`.

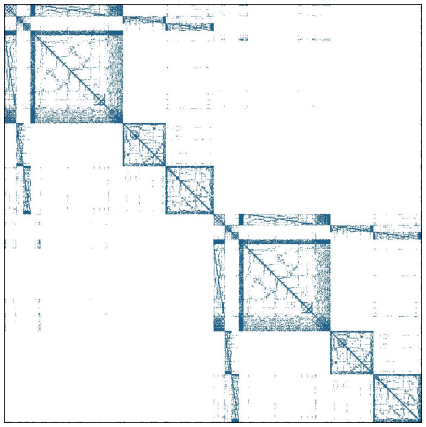


(e) Matrice `S3dkt3m2`.

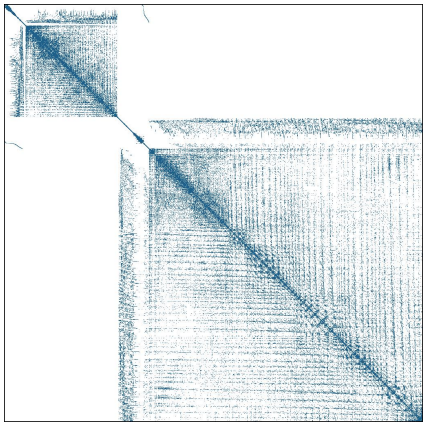
FIGURE 8.4 – Représentation en nuage de points du jeu de petites matrices.



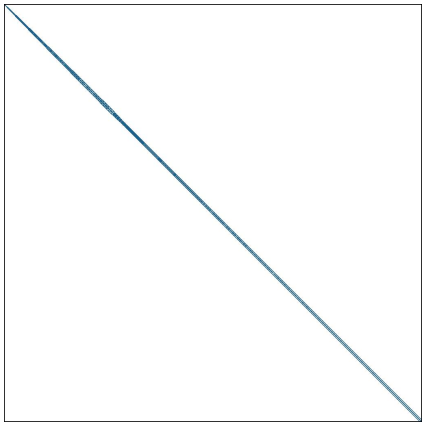
(a) Matrice `F1`.



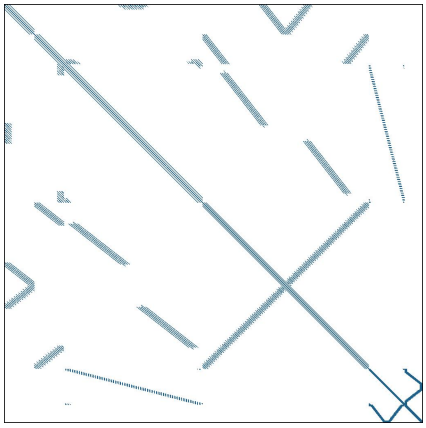
(b) Matrice `inline_1`.



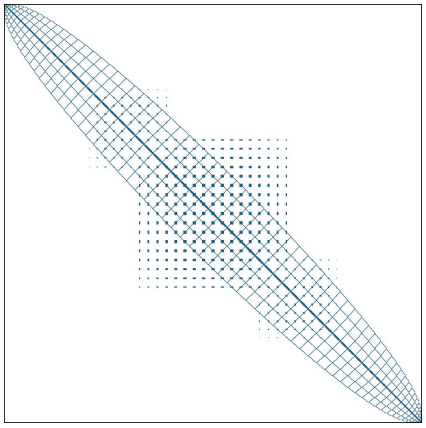
(c) Matrice `ldoor`.



(d) Matrice `ML_Laplace`.



(e) Matrice `RM07R`.



(f) Matrice `Si02`.

FIGURE 8.5 – Représentation en nuage de points du jeu de grandes matrices.

8.3 Résultats des expérimentations

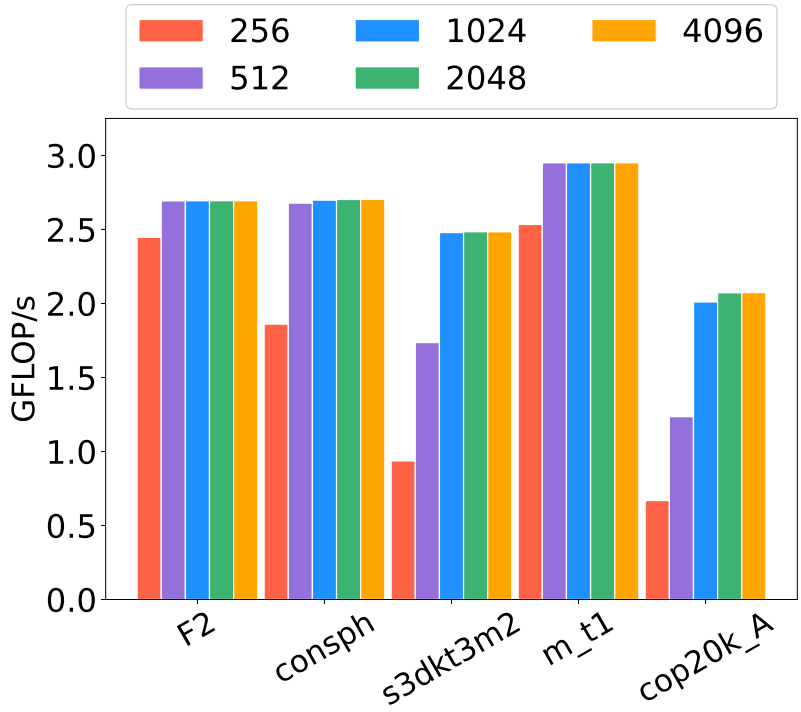
Les expérimentations menées avec l'outil de simulation se sont inscrites dans un processus exploratoire qui a pour objectif de déterminer les paramètres permettant d'obtenir les meilleurs performances du noyau de calcul afin d'évaluer la capacité du modèle de programmation à masquer les latences d'accès aux données distantes. Cette capacité repose sur le préchargement des données. Nous avons utilisé le scénario le plus défavorable où la mémoire de l'accélérateur est vide de toutes données au démarrage. La première expérimentation, présentée dans la sous-section 8.3.1, a permis de déterminer le nombre de *chunks* à précharger par flux de données pour exploiter au mieux le système. La bonne caractérisation de ce paramètre est primordiale parce qu'il définit la taille de l'espace mémoire de l'accélérateur nécessaire pour que le noyau de calcul soit dans un mode de fonctionnement nominal. Les données préchargées étant stockées dans la mémoire de l'accélérateur, si le nombre d'emplacements mémoire est trop petit par rapport au nombre de *chunks* préchargés, des effets de congestion mémoire risquent d'apparaître. De cette manière, une surévaluation du nombre de *chunks* à précharger conduirait à une surutilisation de la mémoire de l'accélérateur. La deuxième expérimentation, présentée dans la sous-section 8.3.2, a fait l'évaluation des performances du noyau de calcul en fonction du niveau de parallélisme, qui correspond au nombre d'unités de traitement instanciées dans le FPGA. Pour ce faire, nous avons utilisé les informations acquises lors de la première expérimentation pour déterminer le nombre de *chunks* à précharger par flux de données et la taille mémoire de l'accélérateur requise. L'objectif de la troisième expérimentation, présentée dans la sous-section 8.3.3, est d'évaluer les capacités du modèle de programmation à masquer les latences d'accès aux données distantes. Pour ce faire, nous avons comparé les performances du noyau de calcul en fonction des trois topologies de systèmes. Pour les trois premières expérimentations, nous avons utilisé le jeu de petites matrices afin de limiter le temps de simulation. Pour la dernière expérimentation, présentée dans la sous-section 8.3.4, nous avons utilisé le jeu de grandes matrices afin d'apporter une validation du comportement du modèle de programmation lorsque la taille des données augmente. Cette expérimentation consiste à évaluer les performances du noyau de calcul en fonction des trois topologies de systèmes.

8.3.1 Évaluation des performances du noyau de calcul en fonction de la profondeur de préchargement des données

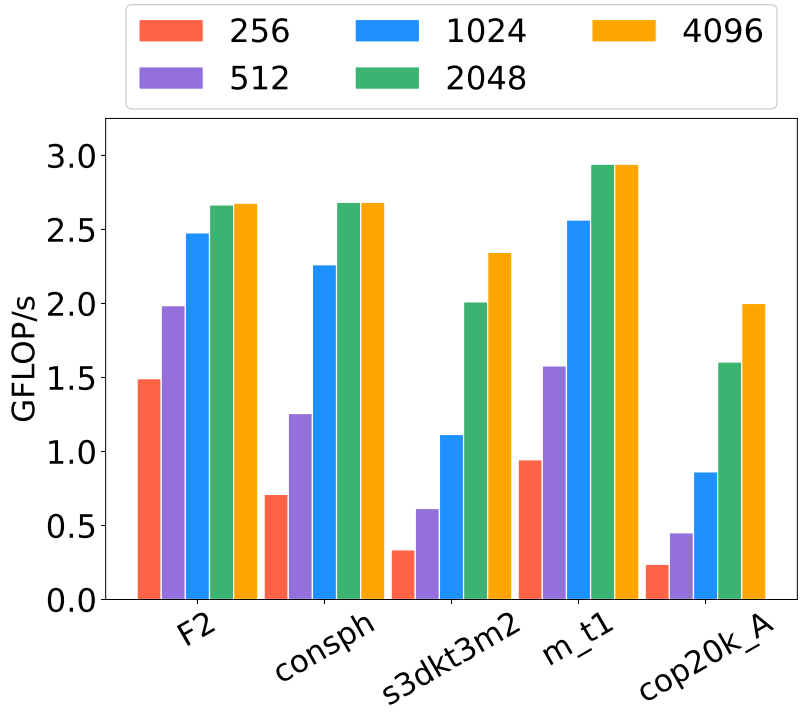
Le préchargement des données a pour objectif de masquer les latences d'accès aux données distantes qui, telles que présentées dans le chapitre 7, sont dépendantes de la topologie du système. Pour cette raison, la caractérisation du nombre de *chunks* à précharger par flux de données à été réalisée pour les deux topologies de systèmes induisant des latences d'accès aux données distantes (*Local* et *Remote*). Plus les requêtes d'accès sont émises en amont, moins le risque que le pipeline du noyau de calcul se bloque est élevé. La profondeur des FIFO se trouvant à la sortie des étages émettant les requêtes d'accès à la MVP définit le nombre de *chunks* préchargés pour chaque flux de données. Pour l'expérimentation, le noyau de calcul a été instancié avec 16 PE. La taille maximale d'un *chunk* a été fixée à 8 kio. Ainsi, un *chunk* peut contenir au plus 1024 éléments d'une rangée [1]. Pour ne pas perturber les mesures par des effets de congestion mémoire, la taille de la mémoire de l'accélérateur a été fixée à 512 Mio afin de pouvoir contenir entièrement les données.

La figure 8.6a représente les vitesses de calcul obtenues avec la topologie *Local* et la figure 8.6b celles obtenues pour la topologie *Remote*. Pour les deux topologies, l'augmentation de la profondeur de préchargement permet d'accélérer les calculs jusqu'à atteindre une limite. Cette limite caractérise le nombre de *chunks* à précharger par flux de données pour masquer les latences d'accès lorsque le noyau est dans un mode de fonctionnement nominal. De manière logique, les résultats montrent qu'il faut précharger plus de *chunks* pour la topologie *Remote* que pour la topologie *Local* puisque les latences d'accès sont plus élevées. Nous pouvons également observer que, plus la charge de travail pour une matrice est faible, plus le nombre de *chunks* à précharger pour atteindre la limite de performance est important. De cette manière, pour la topologie *Local*, les matrices les plus denses (`consph`, `F2` et `m_t1`) atteignent leur limite en préchargeant seulement 512 *chunks*, tandis que le masquage des latences pour les matrices les plus creuses (`cop20k_A` et `S3dkt3m2`) nécessite de précharger 1024 voire 2048 *chunks* par flux. Pour la topologie *Remote*, la

1. L'indice de colonne et la valeur d'un élément non nul sont chacun encodé avec 4 octets



(a) Vitesse de calcul en GFLOP/s avec la topologie *Local*.



(b) Vitesse de calcul en GFLOP/s avec la topologie *Remote*.

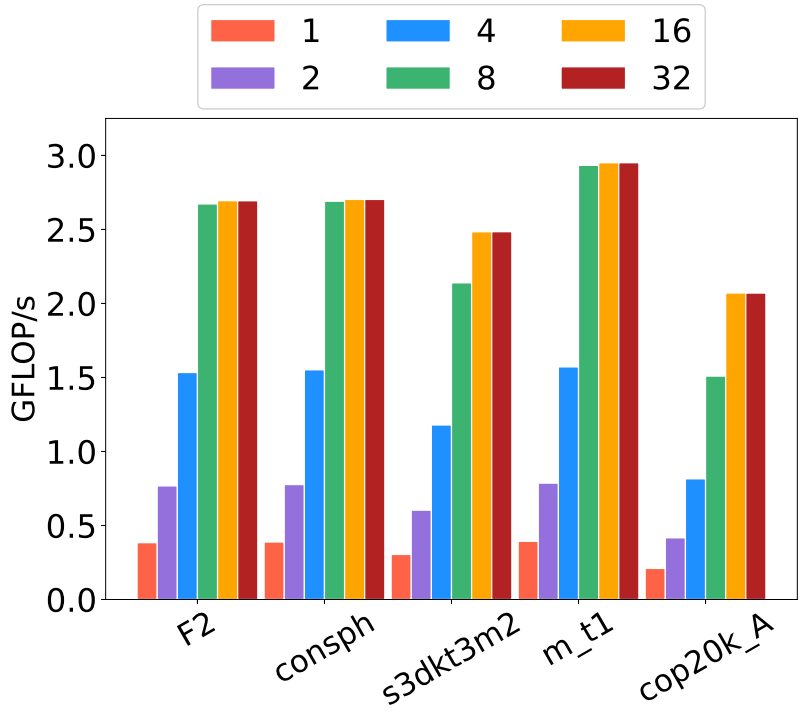
FIGURE 8.6 – Évaluation des performances du noyau de calcul en fonction de la profondeur de préchargement des données. La légende indique le nombre de *chunks* préchargés par flux de données.

limite de performance pour les matrices les plus denses est atteinte par le préchargement de 2048 *chunks*. Pour les deux matrices les plus creuses, la limite ne se dessine pas clairement sur la figure. Cependant, le préchargement de 4096 *chunks* permet d'atteindre les meilleures performances observées avec la topologie *Local*.

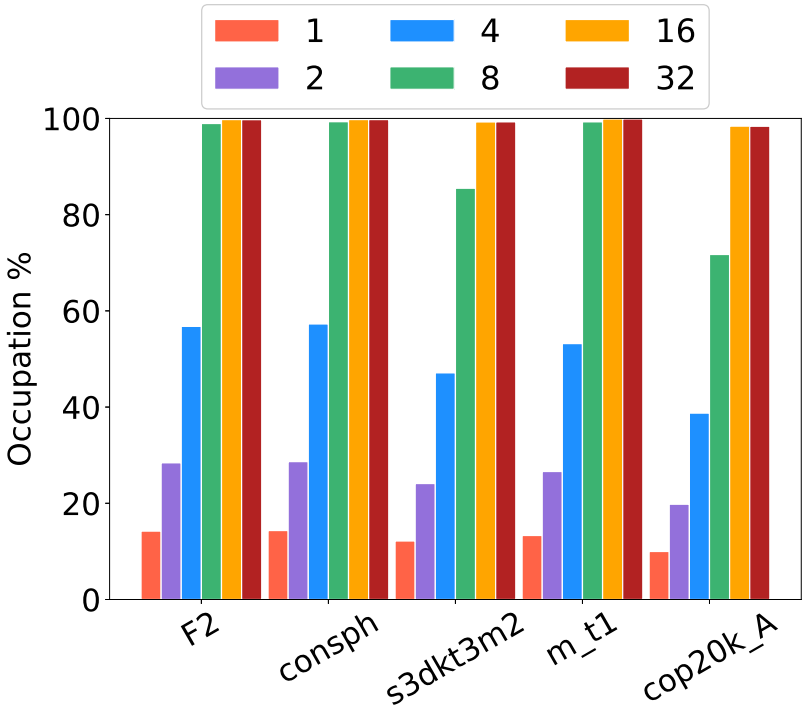
8.3.2 Évaluation des performances du noyau de calcul en fonction du niveau de parallélisme

Un accélérateur matériel permet de traiter les données avec un large parallélisme pour augmenter la vitesse de résolution d'un problème. Pour évaluer la capacité du noyau de calcul à accélérer le traitement des données, nous avons comparé les performances en faisant varier le nombre de PE pour la topologie *Local*. Chaque unité de traitement accède à un flux de données propre à son activité. Pour cette expérimentation, à partir des observations faites dans la sous-section 8.3.1, le nombre de *chunks* préchargés par flux de données a été fixé à 1024. La taille de l'espace mémoire de l'accélérateur a été fixée à 64 Mio de sorte à ce que les matrices ne puissent pas être entièrement contenues en mémoire. Ainsi, l'activité parallèle des PE a pour effet de provoquer l'éviction de données en mémoire. Pour cette raison, la taille de l'espace mémoire n'a pas été adaptée au nombre de PE instanciées afin de pouvoir observer si la diminution de l'espace mémoire par PE a un impact sur les performances. L'augmentation du nombre de PE permet de paralléliser les flux de données et doit conduire à une augmentation du débit de données traversant le noyau de calcul. L'augmentation du débit peut être limitée par les latences d'accès aux données ou par la bande passante de la mémoire. Un préchargement trop lent des données peut conduire à une famine, tandis qu'une bande passante trop faible peut former un goulot d'étranglement.

La figure 8.7a présente les vitesses de calcul obtenues lorsque le nombre de PE varie de 1 à 32. Nous pouvons observer que l'augmentation du parallélisme permet une accélération linéaire de la vitesse de calcul entre 1 PE et 4 PE. Lorsque le parallélisme continue d'augmenter, l'accélération décroit, puis devient nulle. Ainsi, pour les trois matrices les plus denses, l'augmentation du nombre de PE au-delà de 8 ne permet pas d'accélérer la vitesse de calcul. Pour les deux matrices les plus creuses, la limitation s'observe seulement à partir de



(a) Vitesse de calcul en GFLOP/s.



(b) Activité du contrôleur mémoire (taux d'occupation).

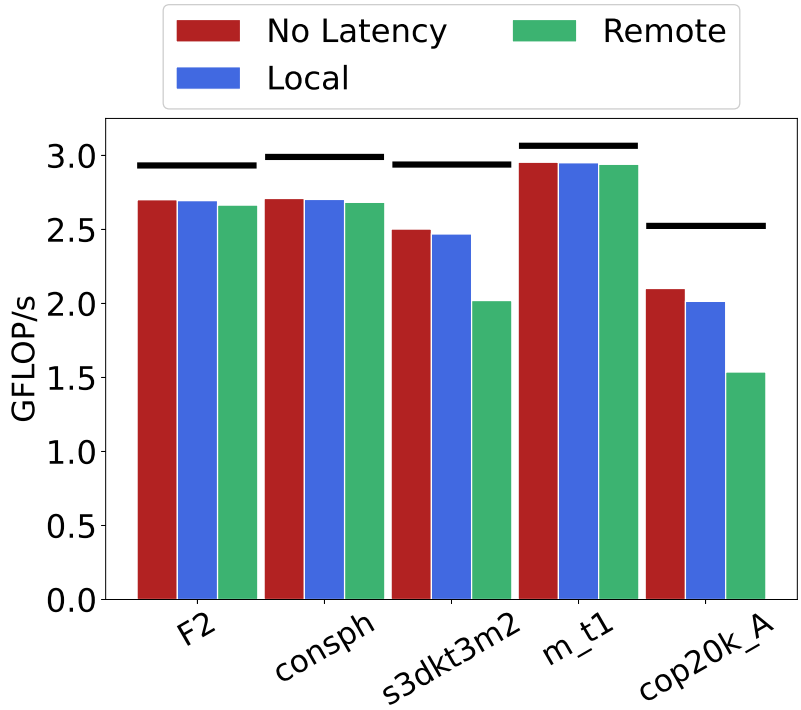
FIGURE 8.7 – Évaluation des performances du noyau de calcul en fonction du niveau de parallélisme avec la topologie *Local*. La légende indique le nombre d'unités de traitement instanciées.

16 PE. La figure 8.7b représente la part de temps où le contrôleur mémoire est actif. Plus son activité est importante, plus la bande passante de la mémoire est utilisée. Nous pouvons observer que la limitation des performances lorsque le parallélisme augmente s'explique par une saturation de la bande passante de la mémoire. Ce phénomène de saturation est inhérent aux applications dites *memory-bound*, comme la multiplication de matrices creuses parce que leur charge de travail est faible. De cette manière, le modèle de programmation permet de masquer suffisamment les latences pour que le noyau de calcul atteigne les performances maximales autorisées par la mémoire locale de l'accélérateur.

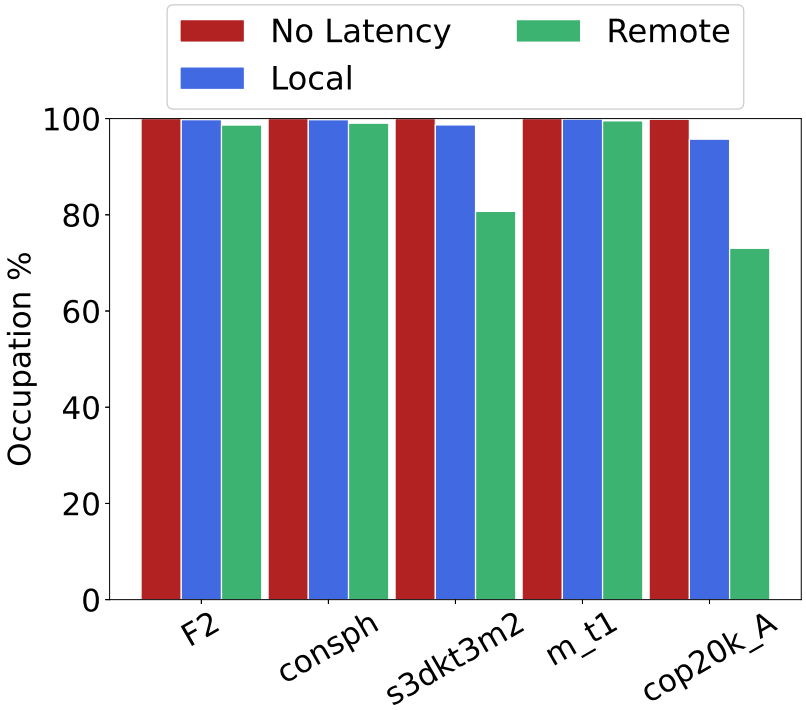
8.3.3 Évaluation des capacités du modèle de programmation à masquer les latences d'accès aux données distantes

Pour évaluer la capacité du modèle de programmation à masquer les latences d'accès aux données distantes, nous avons comparé les performances d'exécution avec les trois topologies de systèmes. Pour cette expérimentation, le noyau de calcul a été instancié avec 16 PE de manière à pouvoir atteindre le maximum des performances permises par la bande passante de la mémoire du FPGA. Comme pour l'évaluation précédente, la taille de l'espace mémoire de l'accélérateur a été fixée à 64 Mio. À partir des observations présentées dans la sous-section 8.3.1, le nombre de *chunks* préchargés par flux de données a été fixé à 1024 pour la topologie *Local* et à 2048 pour la topologie *Remote*.

La figure 8.8a représente les vitesses de calcul obtenues pour cette expérimentation. Nous pouvons observer que pour les trois matrices les plus denses, l'augmentation des latences induites par les topologies *Local* et *Remote*, n'a pas pour effet de dégrader les performances par rapport au cas idéal sans latence. Pour les deux matrices les plus creuses, nous pouvons observer un écart de performance lorsque le temps d'accès aux données augmente. Néanmoins, cet écart est relativement faible par rapport aux différences de latence entre les trois topologies. La figure 8.8b représente le taux d'occupation du contrôleur mémoire pour cette expérience. Telle que nous pouvons l'observer, la dégradation des performances est associée à une exploitation plus faible de la bande passante mémoire. Ainsi, la baisse des performances s'explique par une famine



(a) Vitesse de calcul en GFLOP/s. Les barres horizontales noires représentent la vitesse de pointe théorique pour chaque matrice.



(b) Activité du contrôleur mémoire (taux d'occupation).

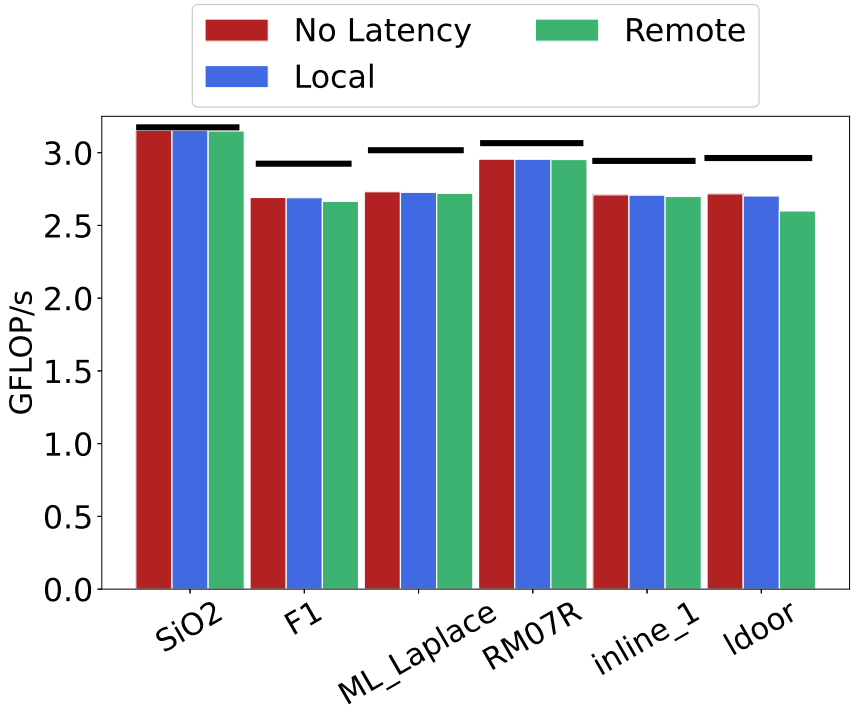
FIGURE 8.8 – Évaluation des performances du noyau de calcul en fonction de la topologie du système pour le jeu de petites matrices.

de données qui indique que le flux de données transférés ne peut pas être entièrement recouvert dans le temps par le flux de données traités. Lorsque les latences d'accès aux données augmentent, la longueur temporelle du flux de données transférés augmente également. Si la fréquence de fonctionnement du FPGA n'est pas modifiée, la longueur temporelle du flux de données traités ne change pas. De cette manière, la capacité du modèle de programmation à masquer les latences d'accès aux données est liée à la charge de travail du noyau de calcul. Les lignes horizontales noires sur la figure 8.8a indiquent les vitesses de calcul maximales théoriques pour chaque matrice. Nous pouvons observer que le noyau de calcul ne parvient pas a atteindre ces vitesses, même pour la topologie sans latence. Ceci s'explique par l'irrégularité des rangées des matrices. Les accès à la mémoire se font par paquets de données en fonction de la largeur du bus de données. Pour les expérimentations présentées, la largeur du bus est fixé à 512 bits, ce qui permet de lire ou écrire 8 éléments par cycle. De cette manière, l'irrégularité des données ne permet pas d'exploiter toute la largeur du bus à chaque accès.

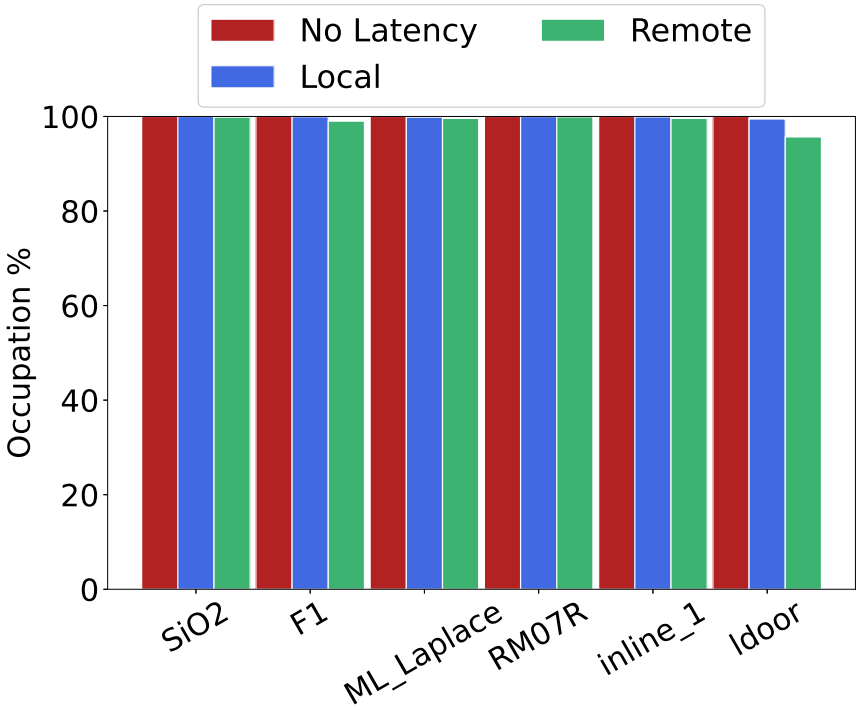
8.3.4 Validation du comportement du modèle de programmation avec le jeu de grandes matrices

La dernière expérimentation portant sur la multiplication de matrices creuses a pour objectif de confirmer les observations faites lors des premières expérimentations en utilisant le jeu de grandes matrices. L'objectif des expérimentations étant d'évaluer la capacité du modèle de programmation à masquer les latences d'accès, nous avons comparé les performances avec les trois topologies de systèmes. Le noyau de calcul a été configuré avec les mêmes paramètres que l'expérience similaire utilisant le jeu de petites matrices, à savoir 16 PE, un espace mémoire de 64 Mio, 1024 *chunks* préchargés par flux de données pour la topologie *Local* et 2048 pour la topologie *Remote*.

La figure 8.9 représente les résultats obtenus. Pour toutes les matrices, sauf la matrice `ldoor`, l'augmentation des latences d'accès aux données distantes ne montre pas d'impact significatif sur les vitesses de calcul. Une faible dégradation est observée pour la matrice `ldoor` avec la topologie *Remote*. Comme le montre la figure 8.9b, cette dégradation reflète une sous-exploitation de la bande passante mémoire, qui traduit une famine de données. Cette matrice est la plus creuse du jeu de matrices et par conséquent celle dont la charge



(a) Vitesse de calcul en GFLOP/s. Les barres horizontales noires représentent la vitesse de pointe théorique pour chaque matrice.



(b) Activité du contrôleur mémoire (taux d'occupation).

FIGURE 8.9 – Évaluation des performances du noyau de calcul en fonction de la topologie du système pour le jeu de grandes matrices.

de travail est la plus faible par rapport à sa taille. De plus, la perte de vitesse pourrait s'expliquer par le motif de la matrice qui résulte en une charge de travail irrégulière dans le temps. Malgré cette mesure singulière, les résultats obtenus pour cette expérimentation montrent que les performances du noyau de calcul ne sont pas directement liées à la sporadicité des matrices, puisque le jeu de grandes matrices est moins dense que celui de petites matrices. L'expérimentation montre que le modèle de programmation permet de s'adapter à l'irrégularité des structures de données en apportant le même niveau de performance lorsque la taille des données augmente et que la densité des matrices décroît. Comme pour l'expérimentation précédente, les lignes horizontales noires représentent les vitesses de calcul maximales théoriques pour chaque matrice. Nous pouvons encore observer que le noyau de calcul n'atteint pas ces vitesses maximales. Cependant l'écart est plus faible que pour le jeu de petites matrices. Pour la matrice `Si02`, la vitesse de calcul mesurée atteint la vitesse de calcul maximale théorique. Ces meilleures performances pour le jeu de grandes matrices s'expliquent par le plus grands nombre d'éléments par rangées permettant une meilleure occupation du bus de données.

8.3.5 Conclusion

Ce cas d'étude nous a permis d'expérimenter le modèle de programmation, proposé dans les travaux de cette thèse, et d'en évaluer la capacité à masquer les latences d'accès aux données distribuées pour une opération d'algèbre linéaire creuse très utilisée . Les expérimentations présentées dans ce chapitre ont mis en lumière un phénomène de famine de données qui se produit lorsque le parallélisme augmente. Ce phénomène s'explique par la formation d'un goulot d'étranglement dû à la bande passante de la mémoire du FPGA. Les expérimentations ont également permis d'observer que le modèle de programmation permet de masquer efficacement les latences d'accès aux données dès lors que la charge de travail est suffisamment importante pour superposer le flux de données transféré et le flux de données traité. Enfin, nous avons pu observer que l'irrégularité des rangées des matrices ne permet pas d'exploiter toute la bande passante de la mémoire du FPGA et donc d'atteindre la vitesse de calcul de pointe théorique. Cependant, plus les matrices sont volumineuses, plus les performances évaluées pour les trois topologies de systèmes se rapprochent de la vitesse de calcul de pointe théorique. La conclusion générale de ces ex-

périmentations est que le modèle de programmation répond particulièrement bien aux besoins du traitement de jeux de données volumineux.

Chapitre 9

Cas d'étude 2 : Simulation de tsunami

Le deuxième cas d'étude utilisé dans le cadre des travaux de cette thèse est un code de simulation numérique de tsunami nommé TsunAWI [1], qui a été développé par l'institut Alfred Wegener. Ce code résout des équations de Barré de Saint-Venant [2] par méthode des éléments finis. Il permet de simuler toutes les étapes du scénario d'un tsunami depuis le séisme jusqu'à l'inondation des terres, en passant par la propagation des vagues dans l'océan. TsunAWI est utilisé par le système germano-indonésien d'alerte rapide aux tsunamis (GITEWS) et est également utilisé sous contrainte temps-réel dans le projet européen LEXIS [55, 56].

Pour satisfaire les contraintes de temps et obtenir des données fiables, la précision et la rapidité sont les deux critères les plus importants de la simulation. Pour atteindre ces objectifs, la surface de l'océan et des terres inondables est modélisée par un maillage non structuré bidimensionnel, dont les éléments sont triangulaires. L'utilisation d'un maillage non structuré permet d'adapter la résolution de ses différentes régions en fonction des besoins de la simulation, dans l'objectif de réduire le nombre de sommets et donc de réduire le volume de données à traiter [111]. Dans le cadre des travaux du projet LEXIS, une zone d'intérêt est ciblée pour évaluer les dommages causés par un séisme et le tsunami qu'il déclenche. Ainsi, afin de modéliser avec précision l'inondation

1. Le code source TsunAWI est en libre accès sur le dépôt :`https://gitlab.awi.de/tsunawi/tsunawi`

2. `https://fr.wikipedia.org/wiki/%C3%89quations_de_Barr%C3%A9_de_Saint-Venant`

des terres, en particulier dans la zone d'intérêt, la résolution du maillage est plus élevée sur les zones côtières que sur les zones hauturières. La réduction de la résolution des zones hauturières permet de réduire la charge de travail et donc d'accélérer le traitement. Toujours dans l'objectif de réduire la charge de travail, le code a été optimisé, par une technique d'exécution conditionnelle, en exploitant la nature de l'évènement simulé. Les sommets du maillage qui ne sont pas immergés, c'est-à-dire qui ne sont pas atteints par une vague, sont traités différemment pour réduire l'intensité arithmétique.

TsunAWI est un code *memory-bound*, c'est-à-dire que sa vitesse de traitement est limitée par les performances du système mémoire du processeur sur lequel il est exécuté. Ainsi, son optimisation nécessite une concentration des efforts sur la gestion des données. La structure de données utilisée et les exécutions conditionnelles rendent cette gestion plus complexe. La parallélisation du code, à l'aide d'OpenMP, permet d'en améliorer les performances, notamment en exploitant toute la bande passante disponible entre le système mémoire et les unités de calcul. Cependant, la distribution des données induite par la parallélisation du traitement nécessite des échanges de données entre les sous-systèmes mémoires à chaque mise à jour du maillage pour maintenir la cohérence des données, ce qui augmente l'effet de saturation de la mémoire. Pour augmenter la parallélisation du code, au-delà des capacités d'un unique processeur, une version hybride MPI/OpenMP a été développée par l'institut Alfred Wegener. Sans mécanisme de cohérence matériel entre les processeurs, la cohérence des données est réalisée au moyen de communications MPI et d'un mécanisme d'échange en halo, tel qu'expliqué en page 47.

Le code TsunAWI a été choisi comme cas d'étude pour plusieurs raisons. Premièrement, il s'agit d'un code de production avec des jeux de données issus de problèmes réels. Ensuite, il permet d'expérimenter le modèle de programmation pour le traitement de maillages non structurés, qui a été présenté dans le chapitre 5. Enfin, le code pourrait tirer bénéfice de l'utilisation d'une mémoire virtuellement partagée afin de prendre le relais des mécanismes matériels de cohérence des données dans le cadre d'une exécution distribuée.

Le chapitre est organisé de la façon suivante : la section 9.1 décrit le noyau de calcul modélisé et son architecture, la section 9.2 présente le jeu de maillages utilisé et la section 9.3 expose les résultats des expérimentations.

9.1 Modélisation du noyau de calcul

Algorithme 3 Structure du code de simulation TsunAWI.

Input: Mesh topology, nodes coordinates, nodes bathymetry and topography & epicenter and magnitude of the earthquake
Output: sea surface height & wave velocity

```
Initialize mesh                          ▷ Allocation and filling of data structures.
for time = 0 ; time < T_end ; time += dt do   ▷ Loop for discretization. T_end is the simulated time and dt is the time step.
    Compute gradient                     ▷ Updating the sea surface gradient of mesh elements. Compute to wet elements and extrapolate to dry elements.
    Compute velocity at nodes            ▷ Updating nodes velocity.
    Compute velocity                     ▷ Updating edges velocity. Compute to wet edges and set to 0 at dry edges.
    Compute sea surface height           ▷ Updating the sea surface height and the water depth at each node. Compute to wet nodes and extrapolate to dry nodes.
end for
Extract results
```

Le code TsunAWI utilise une discrétisation temporelle pour résoudre des équations de Barré de Saint-Venant par méthode des éléments finis. L'algorithme 3 représente la structure du code, qui se compose d'une phase d'allocation et d'initialisation des données du maillage, d'une boucle de discrétisation temporelle, et d'une phase d'extraction des données traitées. La boucle, composée de quatre noyaux de calcul, représente la partie la plus chronophage lors de l'exécution du code. Son nombre d'itérations dépend du temps simulé, allant de 1 heure à 10 heures, et de la taille des pas de temps, allant de 0.1 seconde à 2 secondes. Ainsi le nombre d'itérations varie de 3600 à 180 000.

Dans le temps de la thèse, et plus particulièrement dans le temps attribué à l'étude de ce cas d'étude, il n'était pas possible de porter l'ensemble du code TsunAWI qui est composé de plus de 8000 lignes de code *Fortran* optimisées. Nous avons donc choisi de concentrer les efforts sur l'étude d'un seul noyau de calcul. Pour cibler le noyau que nous avons étudié, nous avons réalisé une analyse du code TsunAWI. Tel que le décrit le tableau 9.1, les noyaux ont une

Nom du noyau	ratio calculs/accès	Part du temps de traitement de la boucle
Compute gradient	≈ 0.5	4.4%
Compute velocity at nodes	≈ 0.62	2.3%
Compute velocity	≈ 1.4	82.7%
Compute SSH	≈ 0.77	10.15%

Tableau 9.1 – Analyse de l'activité des noyaux de calcul du code TsunAWI. Le ratio du nombre calculs réalisés par accès mémoire a été évalué par un comptage manuel. La répartition du temps de traitement par noyau est issu d'une suite d'exécutions sur un processeur AMD EPYC 7702P 64-Core parallélisés avec 128 threads OpenMP.

charge de travail et un ratio de calculs par accès mémoire inégal. Parmi les quatre noyaux, nous avons décidé d'implémenter le noyau *Compute gradient*. Ce choix a été motivé par son faible ratio de calculs par accès mémoire qui le rend plus éprouvant pour le modèle de programmation. En effet, plus le ratio est faible, plus la longueur temporelle du flux de données traité est courte par rapport au flux de données transféré. Par conséquent, plus le ratio est faible, plus il est difficile de masquer les latences. De cette manière, si le modèle de programmation se montre efficace pour ce noyau, il devrait également l'être pour les trois autres noyaux.

Le code 9.1 a été utilisé comme référence pour modéliser le noyau de calcul avec l'outil de simulation. Il s'agit d'une interprétation en langage C du code original écrit en Fortran. La figure 9.1 représente le traitement réalisé par le noyau sous la forme d'un maillage. Le noyau calcule le gradient de la hauteur de la surface de la mer de chaque triangle du maillage par interpolation barycentrique. Cette interpolation consiste à accumuler pour chaque triangle le produit de la hauteur de la mer et des coordonnées barycentriques aux trois sommets. Le tableau *elem2d* décrit la topologie du maillage, le tableau *ssh* contient la valeur de la hauteur de la mer de chaque nœud du maillage, le tableau *bafu* contient les coordonnées barycentriques des sommets de chaque triangle, enfin le résultat est contenu dans le tableau *grad*. Les coordonnées barycentriques étant propres à chaque triangle, seules les données du tableau *ssh*

```
void compute_gradient(const size_t nTriangles, /*
                      Number of triangles in mesh */
                      const uint32_t ** elem2d, /*
                      Mesh's topology */
                      const float * ssh, /* Nodes
                      sea surface heights */
                      const float ** bafu, /* Vertex
                      barycentric coordinates */
                      float * grad /* Element
                      computed gradients */)
{
    for(size_t i = 0; i < nTriangles; ++i)
        grad[i] =  bafu[i][0] * ssh[elem2d[i][0]]
                 + bafu[i][1] * ssh[elem2d[i][1]]
                 + bafu[i][2] * ssh[elem2d[i][2]];
}
```

Code 9.1 – Code de référence utilisé pour la modélisation du noyau de calcul du cas d'étude de simulation de tsunami et interprété à partir du code source TsunAWI.

sont partagées entre le traitement de plusieurs triangles. Les accès au tableau *ssh* sont dépendants de la topologie du maillage, de ce fait la réutilisation des données l'est aussi. Plus les nœuds ont un grand nombre de voisins, plus la réutilisation des données est importante.

Le noyau de calcul proposé dans le cadre des travaux de cette thèse et modélisé avec l'outil de simulation instancie plusieurs unités de traitement (PE) pour paralléliser les calculs. Chaque PE est indépendante et traite une partie différente du maillage. La figure 9.2 représente l'architecture d'une unité de traitement, qui est implémentée sous la forme d'un pipeline. Une unité de calcul, nommée *Multiply Accumulate*, procède aux trois multiplications et aux deux additions pour calculer la valeur du gradient d'un triangle. Cette unité est formée de trois étages. Les trois multiplications sont réalisées en un cycle en parallèle et les deux additions sont réalisées séquentiellement. Ainsi, le calcul d'un gradient nécessite trois cycles. L'architecture étant en pipeline, dans un mode de fonctionnement nominal, un triangle peut être traité à chaque cycle.

Chaque PE est constituée de 7 niveaux d'étages séparés par des FIFO. Le premier niveau d'étage émet les requêtes d'accès aux quatre flux de données d'entrée et au flux de données de sortie. Chaque flux fonctionne de manière

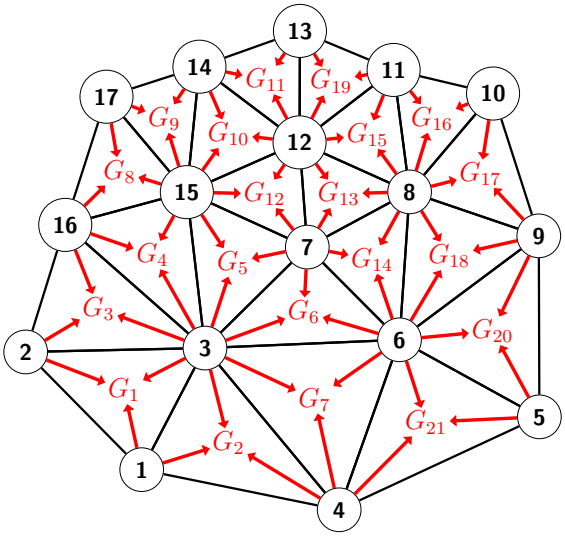


FIGURE 9.1 – Représentation sous la forme d'un maillage du traitement réalisé par le noyau compute gradient. La valeur du gradient de chaque triangle est mise à jour en fonction des coordonnées barycentriques de chacun de ses sommets et de la hauteur du niveau de la mer de chaque nœuds. Les flèches rouges représentent les accès aux données en fonction de la topologie du maillage.

indépendante. Pour chacun, une requête peut être émise si un emplacement dans la mémoire du FPGA est disponible et si la FIFO à sa sortie n'est pas pleine. La profondeur de cette FIFO définit le nombre de *chunks* préchargés par flux. Contrairement à la multiplication de matrices creuses, la taille des *chunks* constituant le résultat est fixe. Pour cette raison, le préchargement des *chunks* du tableau *grad* peut se faire en amont de la production des résultats. Le second niveau d'étages procède à la lecture des données pour les quatre flux d'entrée et à l'écriture des données pour le flux de sortie. Pour les flux d'entrée, la lecture peut être réalisée si le *chunk* est disponible dans la mémoire du FPGA et que la FIFO en sortie dispose de suffisamment d'espace pour contenir un *chunk*. Pour le flux de sortie, l'écriture est réalisée lorsque la FIFO à la sortie de l'unité de calcul contient suffisamment de données pour remplir un *chunk*. Le troisième niveau d'étage correspond à la fenêtre glissante (*slinding window*) et aux tampons mémoires 1 et 2 (*buffer* 1 et *buffer* 2). Deux tampons sont nécessaires, car les éléments du maillage possèdent trois sommets. Le sommet du triangle en cours de traitement avec le plus petit indice est toujours accessible par la fenêtre glissante, puisqu'il correspond à la première donnée de celle-ci. Les deux autres sommets du triangle pouvant être contenus dans deux *chunks* différents, deux tampons sont nécessaires pour les

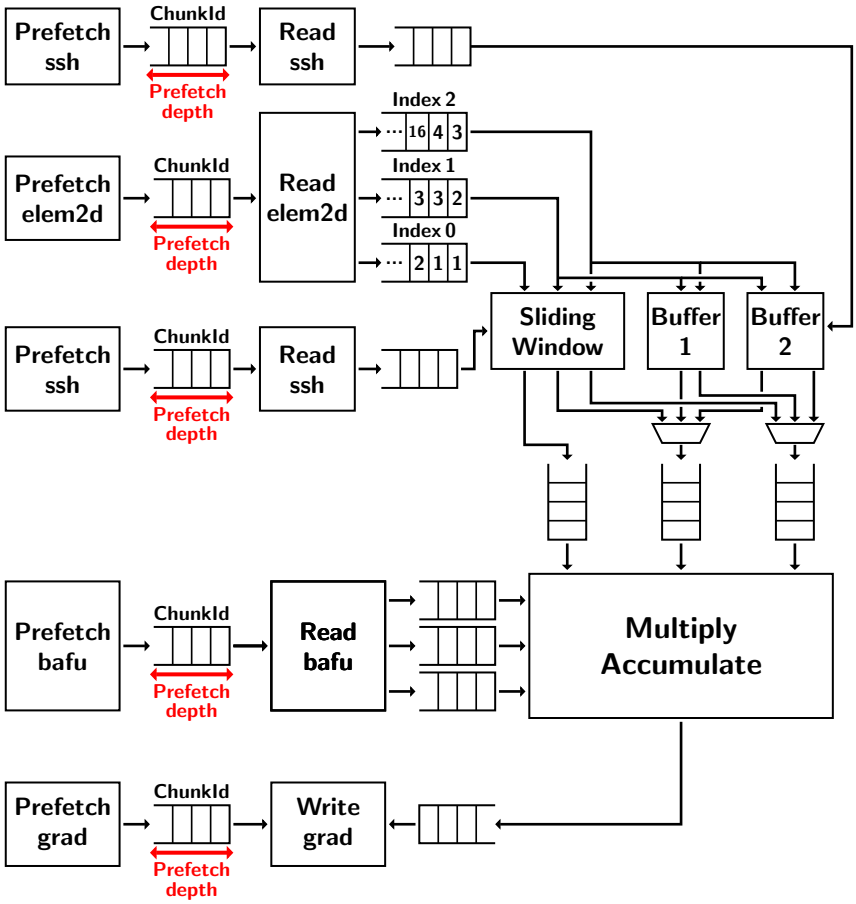


FIGURE 9.2 – Représentation de l'architecture proposée d'une unité de traitement du noyau de calcul compute gradient.

charger séparément. L'implémentation sur FPGA de ces éléments mémorisant correspondent à des BRAM, qui permettent de réaliser une écriture et une lecture en un seul cycle [115]. Ainsi, cet étage réalise deux actions par cycle : la mise à jour des données contenues dans ces trois éléments mémorisant et la mise à disposition de leur données. La fenêtre glissante et les tampons sont mis à jour par les flux de *chunks* contenant les valeurs du tableau *ssh* et sont adressés en lecture par les indices des sommets des triangles, c'est-à-dire la topologie du maillage contenue dans le tableau *elem2d*.

Pour le FPGA modélisé, la taille mémoire maximale d'une BRAM est de 36 kb, soit 4 kio. Ainsi, la taille de la fenêtre glissante a été fixée pour toutes les expérimentations à 1024 emplacements de 4 octets (taille d'un mot). De la même manière, la taille maximale d'un tampon est de 1024 emplacements. Un tampon doit pouvoir charger entièrement un *chunk*. Ainsi, la taille maximale

d'un *chunk* est contraint par celle des BRAM.

L'architecture du noyau de calcul décrite dans cette section a été modélisée avec l'outil de simulation présenté dans le chapitre 6. Environs 1500 lignes de code en langage *C++* ont été écrites pour décrire du noyau de calcul. La majorité du code est utilisée pour vérifier de la conformité de la simulation pendant l'exécution et pour surveiller l'activité du noyau de calcul afin d'en faire l'analyse après l'exécution. Le temps d'une simulation varie en fonction de la taille du problème à résoudre (*i.e.* la taille du maillage) et de la topologie du système modélisé (*i.e. No Latency*, *Local* ou *Remote*). Ainsi, pour le plus petit maillage utilisé lors des expérimentations avec la topologie *No Latency*[3], le temps de simulation d'une exécution du noyau de calcul est d'environ 5 s. Tandis que pour le plus gros maillage utilisé lors des expérimentations avec la topologie *Remote*[4], le temps de simulation d'une exécution du noyau de calcul est d'environ 410 s.

9.2 Jeu de maillages utilisées pour les expérimentations

Pour mener les expérimentations, nous avons utilisé un jeu de cinq maillages issus du projet de recherche LEXIS[5]. Ces maillages, dont les caractéristiques sont décrites dans le tableau 9.2, se différencient par leurs dimensions et par la région du globe qu'ils modélisent. Dans le cadre des travaux du projet LEXIS, ces cinq maillages sont utilisés pour évaluer les dommages causés par trois scénarios de séismes dans trois zones d'intérêts :

— Deux maillages (`Padang_C` et `Padang_F`), différenciés par leurs résolutions globales, modélisent l'océan Indien le long des côtes de l'île indonésienne de Sumatra. Ils sont utilisés pour simuler l'inondation de la ville de Padang qui a été identifiée comme vulnérable à cause de l'activité sismique de la fosse de Java [81].

3. La topologie *No Latency* est celle dont le temps de simulation est le plus court, car l'activité du moteur de simulation n'est pas contrainte par les latences d'accès aux données partagées.

4. La topologie *Remote* est celle dont le temps de simulation est le plus long, car l'activité du moteur de simulation est contrainte par les plus longues latences d'accès aux données partagées et que le processeur qui l'exécute est plus lent que celui utilisé pour les deux autres topologies de système.

5. `https://lexis-project.eu/web/`

— Deux autres maillages (`Coquimbo_C` et `Coquimbo_F`) permettent de modéliser l'océan Pacifique le long des côtes chiliennes avec deux granularités de résolutions. Ils sont utilisés pour simuler l'inondation de la ville de Coquimbo située au Chili et qui est menacée par l'activité sismique de la fosse du Pérou-Chili.

— Le maillage `Méditerranée` modélise l'ensemble de la mer Méditerranée. Il est utilisé pour reproduire le scénario du tsunami qui avait été provoqué par un séisme de magnitude 8.0 le 21 juillet 365 et qui fit des milliers de mort à Alexandrie [6].

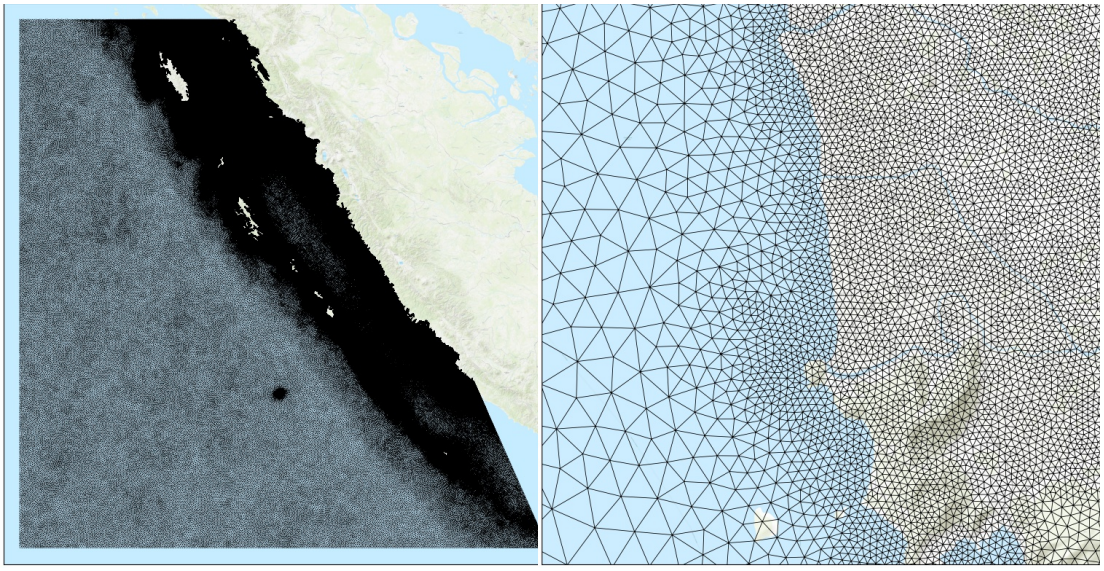
Nom	Résolution	$\|T(M)\|$	$\|V(M)\|$	Taille mémoire
`Padang_G`	82 m-15 km	460 119	231 586	14 Mo
`Padang_F`	7 m-5 km	2 470 345	1 242 653	74 Mo
`Coquibo_G`	72 m-12 km	3 396 755	1 709 506	102 Mo
`Coquimbo_F`	10 m-12 km	9 762 027	4 887 927	293 Mo
`Méditerranée`	31 m-10 km	9 917 645	4 999 404	298 Mo

Tableau 9.2 – Caractéristiques des maillages utilisés pour les expérimentations. La résolution minimale est la distance la plus courte entre deux nœuds voisins et la résolution maximale est la distance la plus longue entre deux nœuds voisins. Les noms de maillages suffixés par la lettre `F` correspondent aux maillages à granularité fine et ceux suffixés par la lettre `G` correspondent aux maillages à granularité grossière. $|T(M)|$ désigne le nombre de triangles du maillage. $|V(M)|$ désigne le nombre de sommets du maillage. La taille mémoire correspond à celle utilisée pour les 5 tableaux accédés par le noyau *compute_gradient*.

La figure 9.3 illustre la variation de résolution au sein du maillage `Padang_G`. Sur la figure 9.3a, nous pouvons observer que la densité de sommets est beaucoup plus importante sur les zones côtières et terrestres que sur les zones hauturières. Telle que représentée par la figure 9.3b, la surface de la zone d'intérêt est modélisée avec la résolution la plus élevée. La même stratégie de modélisation est employée pour tous les maillages.

Cinq opérations flottantes sont nécessaires pour calculer la valeur du gradient d'un triangle. Ainsi, le nombre de triangles formés par le maillage définit la charge de travail du noyau de calcul. De manière logique, la taille mémoire du jeu de données est proportionnelle à la dimension de chaque maillage. Le degré d'un sommet d'un maillage est le nombre d'arêtes le reliant aux autres

6. `https://www.history.com/this-day-in-history/tsunami-hits-alexandria-egypt`



(a) Topologie du maillage à résolution grossière (Pagand_G). La plus petite longueur d'arête est de l'ordre de la centaine de mètres. La plus grande longueur d'arête est de l'ordre de la quinzaine de kilomètres.

(b) Gros plan de la topologie du maillage à résolution fine (Padang_F) sur la ville de Padang dans l'île de Sumatra.

FIGURE 9.3 – Représentation de la topologie du maillage Padang suivant la zone de subduction de la fosse de Java le long de la côte de l'océan Indien de l'île indonésienne de Sumatra. La résolution du maillage est plus élevée sur les zones côtières et terrestres.

sommets. Plus le degré d'un sommet est grand, plus sa valeur de hauteur de mer est réutilisée. Pour chaque maillage M du jeu utilisé, le degré maximal est $\Delta(M) = 9$ et le degré minimal est $\delta(M) = 3$, sauf pour le maillage `Padang_F` qui a pour degré minimal $\delta(M) = 2$. Étant donnée que chaque triangle possède trois arêtes, le degré moyen de l'ensemble des sommets $V(M)$ est $\overline{deg(v)} = \frac{|T(M)|\times 3}{|V(M)|} \approx 6$.

Les cinq maillages ont été ordonnés avec une courbe de Hilbert. Cette technique d'ordonnancement des sommets permet d'améliorer la localité des données des maillages et d'optimiser l'utilisation de la fenêtre glissante et des tampons. Le tableau 9.3 présente les caractéristiques de localité des données des maillages pour l'utilisation de notre système avec différentes tailles de tampons et en considérant une taille de fenêtre glissante fixe de 1024 emplacements. Ces résultats montrent que la courbe de remplissage s'adapte bien à l'augmentation de la taille des maillages. En effet, le taux de triangles dont

Maillage	$T \in F$	Défaut de données en fonction de la taille des tampons				
		64	128	256	512	1024
`Padang_G`	95.8 %	0.30 %	0.22 %	0.17 %	0.13 %	0.098 %
`Padang_F`	95.7 %	0.33 %	0.23 %	0.17 %	0.13 %	0.096 %
`Coquibo_G`	95.7 %	0.29 %	0.21 %	0.16 %	0.12 %	0.092 %
`Coquimbo_F`	95.3 %	0.46 %	0.28 %	0.19 %	0.14 %	0.10 %
`Méditerranée`	95.5 %	0.30 %	0.21 %	0.16 %	0.12 %	0.094 %

Tableau 9.3 – Caractérisation de la localité des données des maillages ordonnées avec une courbe de Hilbert. $T \in F$ désigne le nombre de triangles dont les trois sommets sont contenus dans la fenêtre glissante. Le taux de défaut de données correspond au nombre de triangles nécessitant le chargement d'un nouveau tampon. Les tailles des tampons sont indiquées en nombre d'emplacements (4 octets).

les données des trois sommets peuvent être contenues ensemble dans la fenêtre glissante est presque identique pour les cinq maillages. Les résultats montrent également que l'ordre obtenu avec la courbe de remplissage permet une réutilisation des données contenues dans les tampons et que l'augmentation de leur taille diminue le taux de défaut de données. Chaque défaut de données nécessite le chargement d'un tampon avec un nouveau *chunk*. Ainsi, l'augmentation de la taille des tampons permet de réduire le nombre de requêtes d'accès à la MVP, mais augmente la quantité de données transférées entre la mémoire et le noyau de calcul pour chaque chargement. Ce qui nécessite de bien dimensionner les *chunks* pour obtenir la taille minimisant le nombre de requêtes à la MVP et optimisant l'utilisation de la bande passante mémoire pour obtenir les meilleures performances.

La figure 9.4 représente sous la forme d'une carte de couleur logarithmique la localité des données obtenue grâce à l'ordonnancement des sommets avec la courbe de remplissage pour la maillage `Padang_F`. Cette figure permet de visualiser facilement la décomposition de l'espace géométrique du maillage réalisée par la courbe de remplissage. Les nœuds souffrant d'une mauvaise localité, c'est-à-dire les nœuds ayant une différence d'indice élevée avec leurs voisins, forment des lignes oranges ou jaunes sur la carte de couleurs. Cette figure permet de comprendre facilement la bonne réutilisation des données chargées dans les tampons. Un *chunk* peut contenir plusieurs nœuds formant une portion de ligne jaune ou orange. Le partitionnement en *chunks* du maillage est réalisé en suivant le tracé de la courbe de remplissage. Le traitement du maillage est réalisé en suivant le même tracé. Ainsi, le partitionnement et le traitement

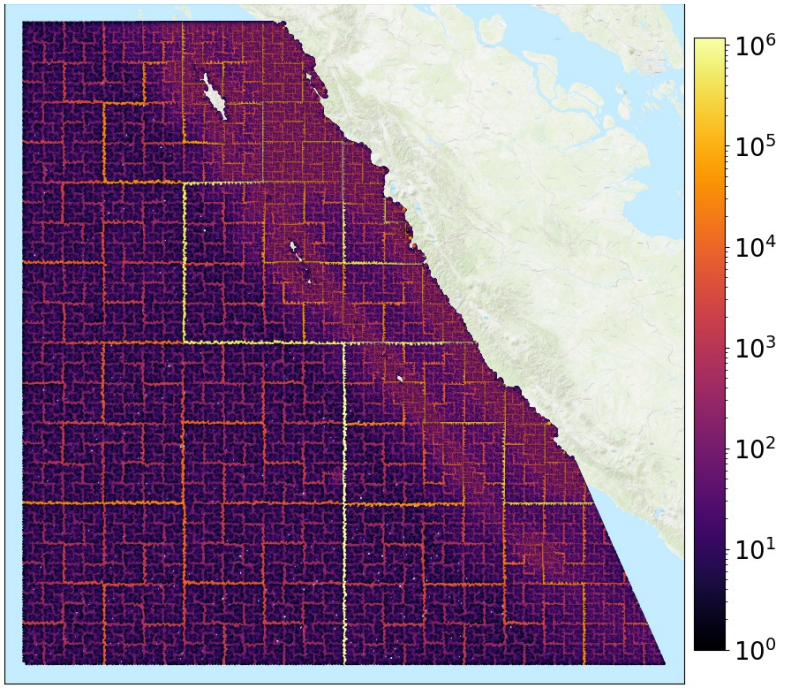


FIGURE 9.4 – Représentation, sous forme de carte de couleur à échelle logarithmique, de la différence maximale entre l'indice de chaque nœud et celui de ses voisins pour le maillage Padang_F. Les lignes formées par les points jaunes illustrent les intersections des espaces géométriques de la courbe les plus éloignés.

repose sur la même logique de décomposition spatiale.

9.3 Résultats des expérimentations

Comme pour le cas d'étude portant sur la multiplication de matrices creuses, les expérimentations décrites dans cette section se sont inscrites dans un processus exploratoire dont l'objectif a été de déterminer les paramètres permettant d'obtenir les meilleures performances du noyau de calcul afin d'en évaluer la capacité à masquer les latences d'accès aux données distantes. Pour le traitement d'un maillage non structuré, cette capacité repose sur le partitionnement en *chunks* de la structure de données et sur le préchargement de ces *chunks*. Comme pour le premier cas d'étude, nous avons considéré le scénario le plus défavorable où la mémoire du FPGA est vide. Ce scénario est celui de la première itération de la boucle de discrétisation du code TsunAWI. La première expérimentation, présentée dans la sous-section 9.3.1, a pour objectif

d'évaluer les performances du noyau pour les trois topologies de systèmes (*No Latency*, *Local* et *Remote*) en fonction de la taille des *chunks*. La deuxième expérimentation, présentée dans la sous-section 9.3.2, a permis de déterminer le nombre de *chunks* à précharger par flux de données pour exploiter au mieux le système. Les résultats de ces deux premières expérimentations permettent de caractériser le volume de données à précharger, et ainsi de définir la taille de l'espace mémoire du FPGA nécessaire au bon fonctionnement du noyau de calcul. La troisième expérimentation, présentée dans la sous-section 9.3.3, a pour objectif d'évaluer les performances du noyau de calcul en fonction du niveau de parallélisme, c'est-à-dire du nombre de PE instanciées. La troisième expérimentation, présentée dans le sous-section 9.3.4, permet d'évaluer la capacité du modèle de programmation à masquer les latences d'accès aux données distantes en fonction des trois topologies de systèmes et en utilisant les conclusions des trois premières expérimentations. La quatrième sous-section 9.3.5 décrit deux expérimentations qui ont pour objectif d'évaluer l'efficacité de l'ordonnancement des données des maillages sur les performances du noyau de calcul et sur celles du code TsunAWI dans son ensemble.

9.3.1 Évaluation des performances du noyau de calcul en fonction de la taille des chunks

Parcourir un maillage non structuré avec la MVP nécessite de partitionner en *chunks* ses données. Pour ce cas d'étude, tous les *chunks* utilisés pour stocker les données d'un maillage ont une taille fixe. Ainsi, cette taille définit le nombre de partitions utilisées pour décomposer la structure de données. Dans le même temps, la taille des *chunks* définit le dimensionnement des tampons mémoires implémentés dans les PE pour lire les données du tableau *ssh*. L'objectif de cette expérimentation est d'évaluer l'impact de la taille des *chunks* sur les performances du noyau de calcul en fonction des trois topologies de systèmes. Pour les trois topologies, nous avons utilisé un noyau de calcul qui instancie 8 PE et qui précharge 2048 *chunks* par flux de données. Pour ne pas perturber les mesures par des effets de congestion mémoire, la taille de la mémoire du FPGA a été fixée à 512 Mio de sorte à pouvoir contenir le plus gros jeux de données entièrement.

La figure 9.5 illustre les vitesses de calcul obtenues pour la topologie sans latence. Nous pouvons facilement identifier que, pour cette topologie, la taille

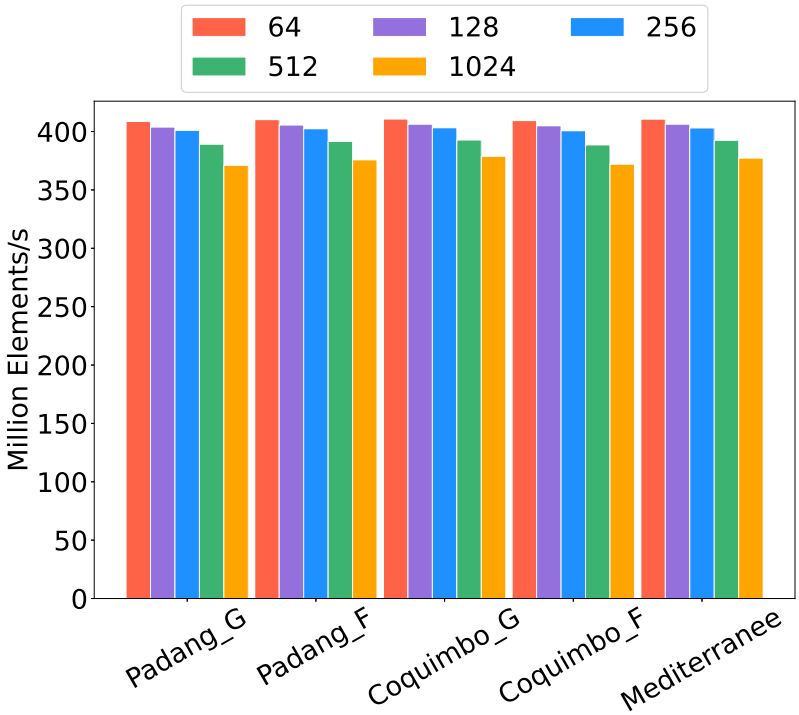


FIGURE 9.5 – Vitesse de calcul en MElements/s en fonction de la taille de *chunks* pour la topologie *No Latency* où l'accès aux données partagées est immédiat. La légende indique les tailles de chunks en nombre de mots de 4 octets.

optimale de *chunk* est de 64 mots, soit la plus petite. Lorsque les tailles augmentent la vitesse décroit. Cet effet est lié au chargement des tampons. Plus les tampons sont volumineux, plus le renouvellement de leurs données est lent et génère un trafic mémoire important. Telle qu'évoquée dans la sous-section 9.2, l'augmentation de la taille des tampons permet de réduire leur nombre de chargements. Cependant, seul un sous ensemble de données chargées dans le tampon est utilisé pour satisfaire les accès qui ne peuvent pas l'être à travers la fenêtre glissante. Autrement dit, toutes les données chargées dans un tampon ne sont pas utilisées. Ainsi, l'augmentation de leur taille conduit également à une réduction de l'utilisation des données par octet transféré. De cette manière, la plus petite taille de *chunk* offre la meilleure utilisation de la bande passante mémoire pour la topologie sans latence.

Les figures 9.6 et 9.7 représentent, respectivement, les vitesses de calcul pour les topologies *Local* et *Remote*. Contrairement à la topologie sans latence, ces résultats montrent qu'une taille de *chunk* plus grande apporte de

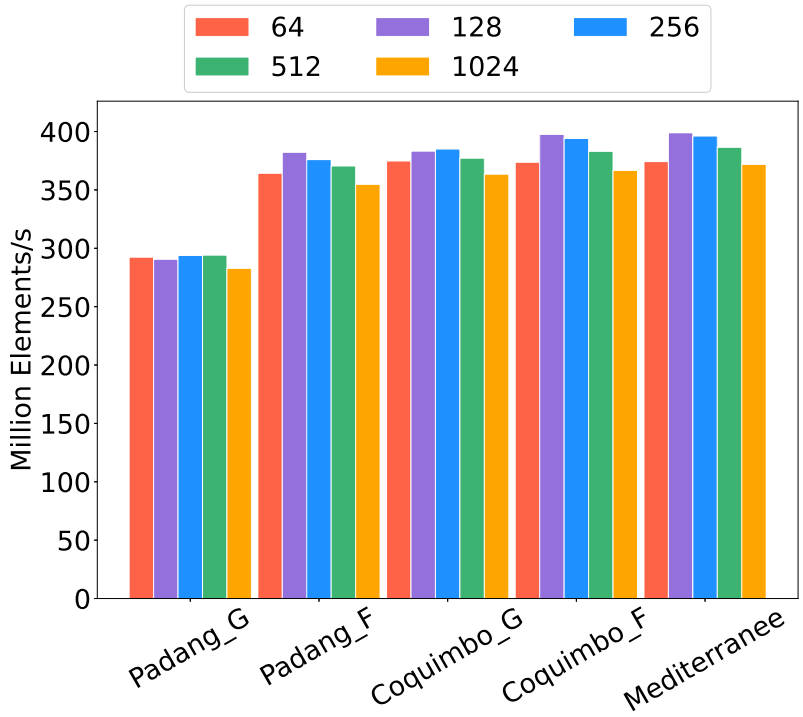


FIGURE 9.6 – Vitesse de calcul en MElements/s en fonction de la taille de *chunks* pour la topologie *Local* où le FPGA est connecté au nœud sur lequel sont situées les données partagées. La légende indique les tailles de chunks en nombre de mots de 4 octets.

meilleures performances. Pour la topologie *Local*, la taille optimale varie selon les maillages et est comprise entre 128 mots et 256 mots. En considérant la moyenne des performances pour les cinq maillages, des *chunks* de 128 mots offrent les meilleures performances. Pour la topologie *Remote*, les meilleures performances sont obtenues lorsque les *chunks* ont une taille de 512 mots. Ainsi, nous pouvons observer que les performances sont corrélées à la taille des *chunks* et aux latences d'accès aux données distantes. Cet effet s'explique par la logique utilisée pour masquer les latences, qui repose sur la superposition des flux de données transférés et des flux de données traitées. Plus les latences d'accès aux données distantes sont élevées, plus le volume de données à précharger doit être important pour masquer efficacement les latences. Pour cette expérimentation, la quantité de *chunks* préchargés par flux de données est la même pour toutes les configurations. Ainsi, l'augmentation de la taille des *chunks* permet d'augmenter le volume de données préchargées. Lorsque ce volume devient suffisamment grand pour masquer efficacement les latences,

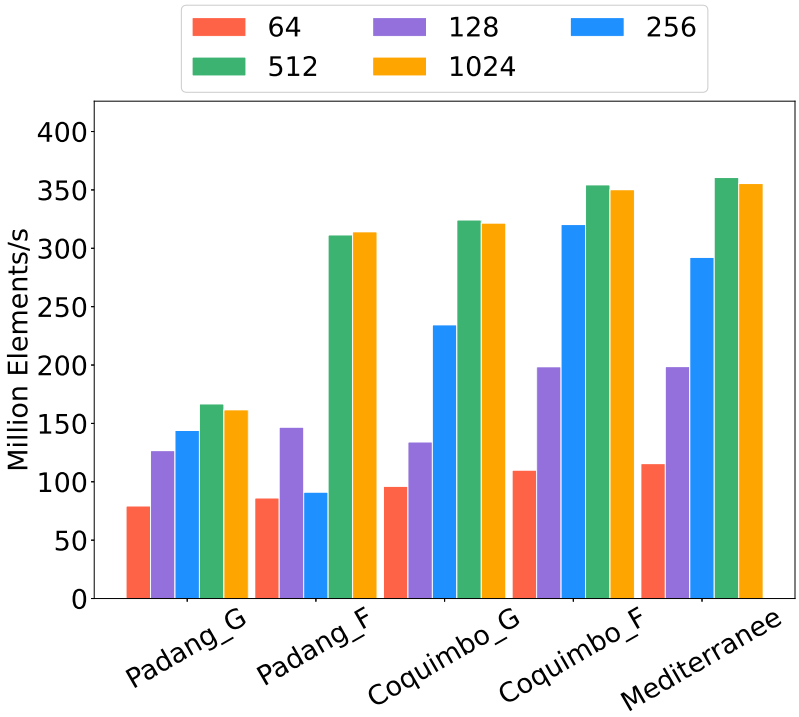


FIGURE 9.7 – Vitesse de calcul en MElements/s en fonction de la taille de *chunks* pour la topologie *Remote* où le FPGA est connecté à un nœud distant de celui sur lequel sont situées les données partagées. La légende indique les tailles de chunks en nombre de mots de 4 octets.

nous pouvons observer que, comme pour la topologie *No Latency*, l'augmentation de la taille des *chunks* a pour effet de ralentir la vitesse de traitement.

Conclusions

Les tailles optimales de *chunk* sont : 64 mots pour la topologie *No Latency*, 128 mots pour la topologie *Local* et 512 mots pour la topologie *Remote*.

9.3.2 Évaluation des performances du noyau de calcul en fonction de la profondeur de préchargement des données

Le modèle de programmation repose sur le préchargement des *chunks* pour masquer les latences d'accès aux données distantes. Tel que décrit dans la sous-section 8.3.1, la caractérisation du nombre de *chunks* à précharger par flux de données est importante pour permettre un masquage des latences efficace et

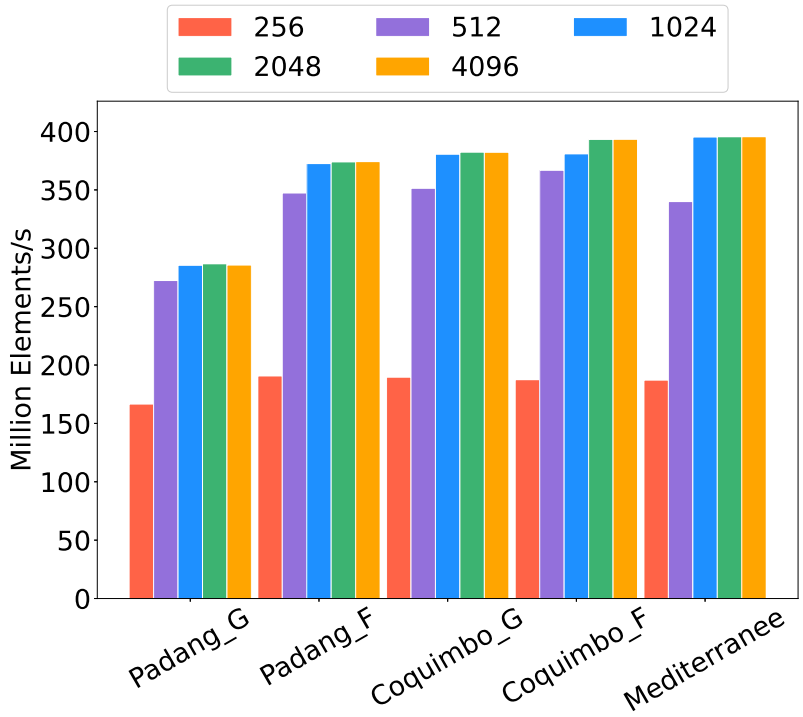


FIGURE 9.8 – Vitesse de calcul en MElements/s en fonction de la profondeur de préchargement des données pour la topologie *Local*. La légende indique le nombre de *chunks* préchargés par flux de données.

une utilisation de l'espace mémoire de l'accélérateur raisonnée. Comme pour le premier cas d'étude, la caractérisation a été réalisée pour les deux topologies de systèmes qui induisent des latences d'accès aux données distantes. Pour cette expérimentation, comme pour la précédente, nous avons instancié le noyau de calcul avec 8 PE et nous avons fixé la taille de la mémoire de l'accélérateur à 512 Mio. Pour chaque topologie, nous avons utilisé les conclusions de la sous-section 9.3.1 pour définir la taille des *chunks*.

La figure 9.8 illustre les vitesses de calcul obtenues pour la topologie *Local*. Ces résultats montrent que l'augmentation du nombre de *chunks* préchargés par flux de données permet d'améliorer les performances jusqu'au nombre de 1024. Au-delà, les performances ne sont pas améliorées ce qui indique que le préchargement de 1024 *chunks* par flux de données est suffisant pour exploiter au mieux le noyau de calcul avec la topologie *Local*.

La figure 9.9 représente les vitesses de traitement mesurées pour la topologie *Remote*. Nous pouvons observer que, pour les quatre maillages les plus volumineux, les performances maximales sont atteintes à partir de 2048 *chunks*

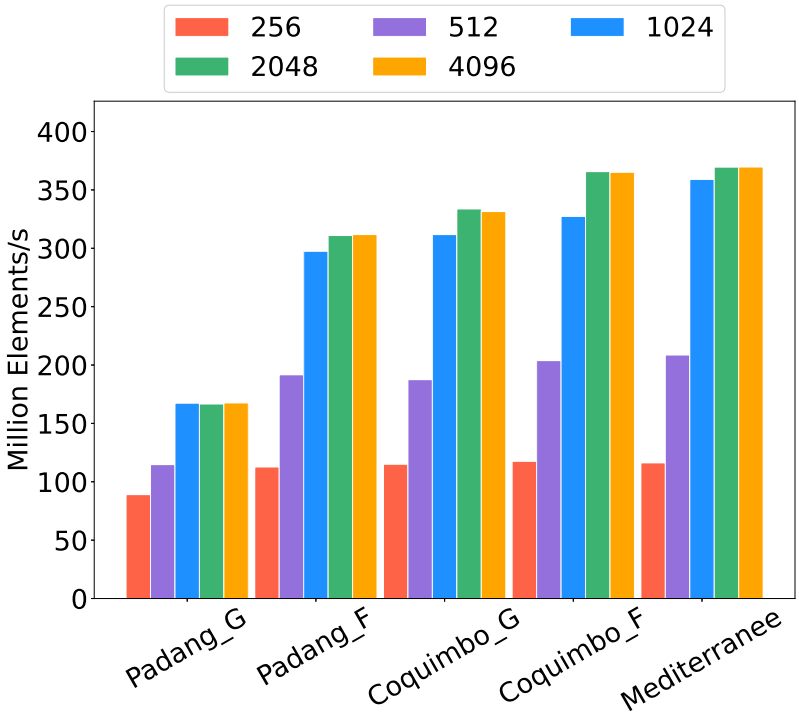


FIGURE 9.9 – Vitesse de calcul en MElements/s en fonction de la profondeur de préchargement des données pour la topologie *Remote*. La légende indique le nombre de *chunks* préchargés par flux de données.

préchargés par flux de données. Ainsi, l'augmentation des latences nécessite de précharger plus de *chunks* pour obtenir un masquage des latences efficace, malgré leur taille plus importante. Pour le plus petit maillage, la quantité de données à transférer étant moins importante, l'effet du préchargement des données est moins visible.

Conclusions

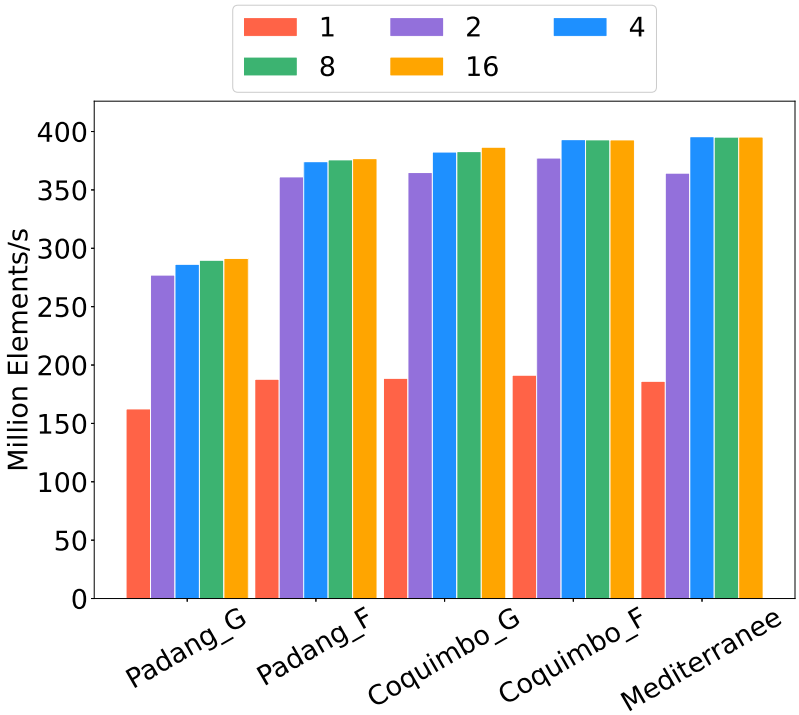
Le nombre optimal de *chunks* à précharger par flux de données est de 1024 pour la topologie *Local* et 2048 pour la topologie *Remote*. En considérant les conclusions de l'expérience précédente, l'espace dans la mémoire du FPGA requit pour le préchargement des données de chaque flux de données est de 512 kio pour la topologie *Local* et 4 Mio pour la topologie *Remote*.

9.3.3 Évaluation des performances du noyau de calcul en fonction du niveau de parallélisme

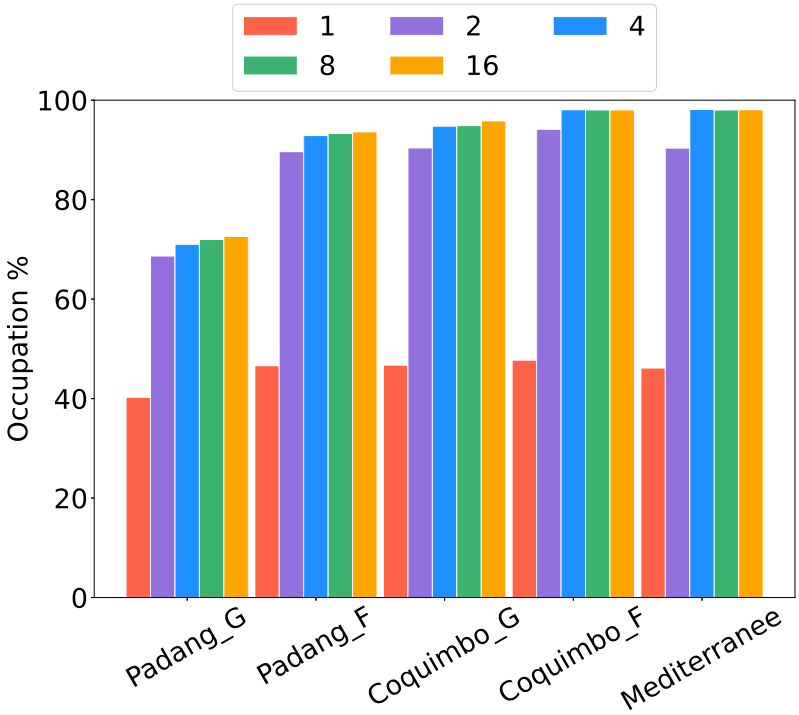
Pour évaluer la capacité du noyau de calcul à traiter le maillage en parallèle, nous avons comparé les performances du noyau lorsque le nombre de PE instanciées varie entre 1 et 16 pour la topologie *Local*. En utilisant les conclusions des deux expérimentations précédentes, nous avons fixé la taille des *chunks* à 128 mots et le nombre de *chunks* préchargés par flux de données à 1024. La taille de la mémoire de l'accélérateur a été fixée 32 Mio de manière à ce que la version du noyau instanciée avec 16 PE ait suffisamment d'espace mémoire pour précharger les données.

La figure 9.10a représente les vitesses de calcul obtenues pour cette expérimentation. Nous pouvons observer que l'augmentation du parallélisme, en passant de 1 PE à 2 PE, permet d'accélérer le traitement. Au-delà, l'augmentation du parallélisme résulte en une faible efficacité. Comme pour le cas d'étude de multiplication de matrices creuses, un phénomène de limitation des performances est visible. Cette limitation indique que les PE ne sont pas suffisamment alimentées en données pour être exploitées en permanence, c'est-à-dire qu'un phénomène de famine de données se produit à l'échelle du noyau de calcul.

La figure 9.10b illustre la part de temps où le contrôleur mémoire est actif pendant l'exécution du noyau. Celle-ci apporte en partie une explication au phénomène de famine observé. En effet, pour les maillages les plus volumineux, la limitation de la vitesse de calcul s'associe à une saturation du contrôleur mémoire, c'est-à-dire à une saturation de la bande passante entre la mémoire de l'accélérateur et le noyau de calcul. Cependant, lorsque la taille des maillages diminue le taux d'occupation du contrôleur mémoire décroît. De cette manière, pour les petits maillages, les résultats de l'expérimentation ne permettent pas d'attribuer la cause de la perte d'efficacité à un goulot d'étranglement formé par la bande passante mémoire. Ainsi, pour ces maillages, la latence d'accès aux données apparaît comme la seule explication à la limitation de la vitesse de traitement.



(a) Vitesse de calcul en GFLOP/s.



(b) Activité du contrôleur mémoire (taux d'occupation).

FIGURE 9.10 – Évaluation des performances du noyau de calcul en fonction du niveau de parallélisme avec la topologie *Local*. La légende indique le nombre d'unités de traitement instanciées.

Conclusions

L'augmentation du parallélisme fait apparaître un phénomène de famine de données dont la cause peut être attribuée, uniquement pour les maillages les plus volumineux, à un goulot d'étranglement formé par la bande passante mémoire. Pour les plus petits maillages, la latence d'accès aux données semble être le facteur de limitation des performances.

9.3.4 Évaluation des capacités du modèle de programmation à masquer les latences d'accès aux données distantes

Comme pour le premier cas d'étude, nous avons évalué la capacité du modèle de programmation à masquer les latences d'accès aux données distantes en comparant les performances des trois topologies de système. Pour ce faire, nous avons instancié un noyau de calcul avec 8 PE. L'expérience précédente à montré que cette configuration de noyau permet d'atteindre les performances maximales permises par la bande passante mémoire de l'accélérateur. Nous avons utilisé la taille de *chunk* la plus adaptée à chaque topologie de système, telle que caractérisée dans la sous-section 9.3.1. En utilisant les conclusions de la sous-section 9.3.2, nous avons fixé le nombre de *chunks* préchargés par flux de données à 1024 pour la topologie *Local* et à 2048 pour la topologie *Remote*.

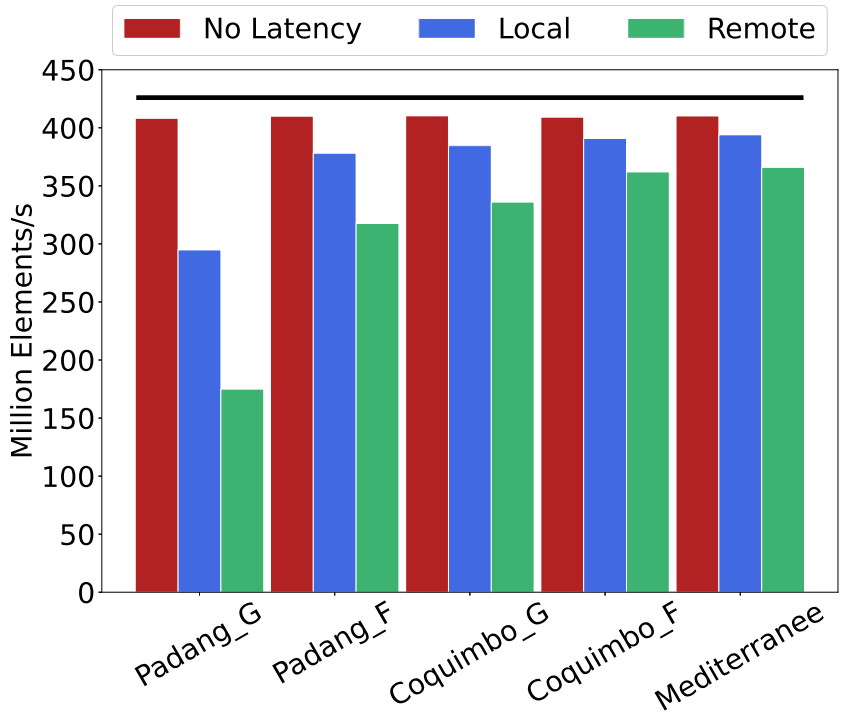
La figure 9.11a illustre les vitesses de calcul obtenues pour cette expérimentation. Nous pouvons observer que l'augmentation des latences d'accès aux données distantes a pour effet de ralentir la vitesse de calcul. Plus le maillage est petit, plus le ralentissement est important. Ainsi, pour le plus gros maillage (`Méditerranée`), le ralentissement de la vitesse de calcul entre les topologies *No Latency* et *Local* est de 4% et celui entre les topologies *No Latency* et *Remote* est de 11%. Pour le plus petit maillage (`Padang_G`), le ralentissement pour la topologie *Local* est de 28% et celui pour la topologie *Remote* est de 57%. La figure 9.11b, qui représente le taux d'occupation du contrôleur mémoire, apporte une explication à cet effet de ralentissement de la vitesse de calcul. Logiquement, pour la topologie *No Latency*, le contrôleur mémoire est en permanence occupé, ce qui signifie que la bande passante de la mémoire de l'accélérateur est saturée. Nous pouvons observer que l'utilisation de la bande passante mémoire décroît lorsque les latences augmentent

et que cette diminution s'amplifie lorsque la taille du maillage diminue. Les barres grises représentées sur la figure indiquent la part de temps de l'exécution nécessaire au chargement du premier *chunk*. Tant que le noyau de calcul n'a pas reçu de données, aucun calcul ne peut être entrepris par le noyau. De manière logique, nous pouvons observer que plus le maillage est petit et plus les latences d'accès aux données distantes sont grandes, plus la part de temps nécessaire au chargement du premier *chunks* est importante. En effet, pour chaque topologie de système, le temps de chargement d'un *chunk* est à peu près constant. Cependant, plus un maillage est petit, plus sa charge de calcul est réduite et donc plus son temps de traitement est court.

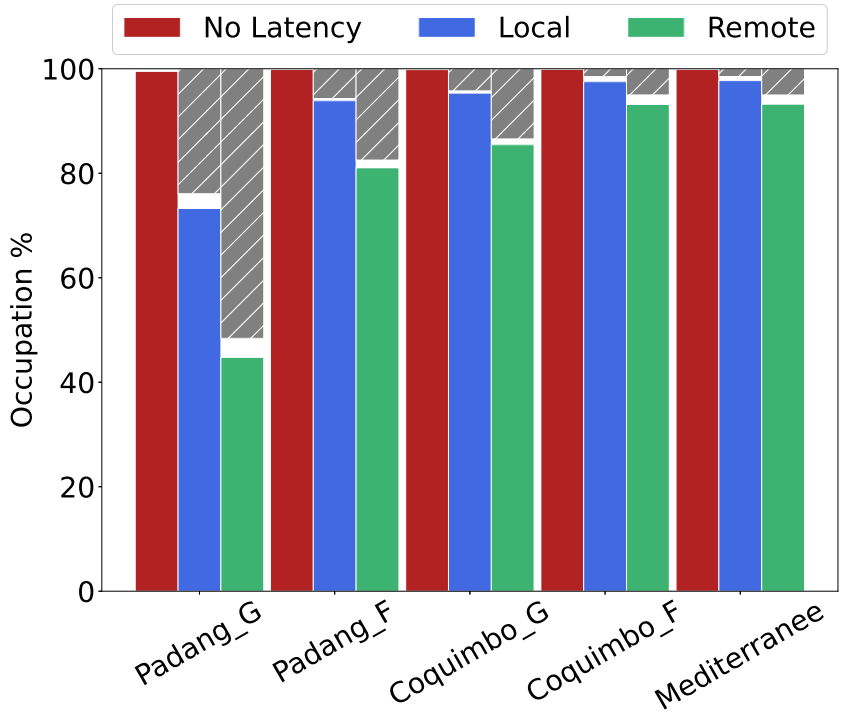
La ligne horizontale noire, présente sur la figure 9.11a, illustre la vitesse de calcul de pointe théorique. Cette vitesse a été déterminée en considérant le temps de traitement comme étant le temps nécessaire au transfert de l'ensemble du jeu de données pour chaque maillage en fonction de la bande passante de la mémoire de l'accélérateur. Tous les maillages ayant approximativement le même ratio de sommets par élément, la vitesse de calcul de pointe théorique est environ la même pour les cinq maillages, soit 426 MElements/s. Nous pouvons observer que la topologie *No Latency* n'atteint pas cette vitesse de pointe théorique. La différence de performance s'explique par les transferts mémoire requis pour le chargement des tampons. Ces transferts utilisent une partie de la bande passante mémoire disponible, ce qui a pour effet de réduire la vitesse de calcul. Néanmoins, nous pouvons observer que l'écart de performance est relativement faible, moins de 4%.

Conclusions

Cette expérience permet de tirer plusieurs conclusions. La première est que seul le temps de chargement des premiers *chunks* impacte significativement la vitesse de calcul, parce qu'au lancement aucun calcul n'est réalisé. La deuxième est que plus la longueur temporelle du flux de données traité est courte, plus il est difficile de masquer les latences d'accès aux données. La troisième est que l'architecture du noyau de calcul proposée permet une exploitation efficace des ressources de l'accélérateur.



(a) Vitesse de calcul en MElements/s.



(b) Activité du contrôleur mémoire (taux d'occupation). Les barres grises représentent le temps de chargement du premier *chunk*.

FIGURE 9.11 – Évaluation des performances du noyau de calcul en fonction de la topologie du système.

9.3.5 Évaluation de l'impact de l'ordonnancement des maillages sur les performances

Pour le code TsunAWI, comme pour d'autres applications manipulant des maillages non structurés, l'ordre dans lequel les sommets du maillage sont indexés a un impact significatif sur les performances. L'ordonnancement d'un maillage est un problème de recherche opérationnelle qui représente un vaste sujet en soi. Dans cette sous-section, deux expérimentations sont présentées, dont l'objectif est de comparer les performances permises par quatre méthodes de tri :

1. un tri utilisant une courbe de Hilbert (*SFC*), correspondant à l'ordre utilisé pour les expérimentations précédentes et qui est également utilisé pour les travaux du projet LEXIS.
2. un tri utilisant un algorithme de parcours en largeur (BFS), tel que présenté dans la section 5.3. Le tri réalisé consiste à visiter tous les nœuds d'un maillage en se déplaçant par voisinage à partir d'un premier nœud, appelé graine. Pour implémenter l'algorithme nous avons utilisé les travaux de Pablo Barrio et Carlos Carreras [12]. Ces travaux montrent qu'en choisissant la graine en fonction du centre de masse du maillage, il est possible d'obtenir une qualité de tri proche de celle obtenue avec une méthode par force brute (c'est-à-dire en testant tous les nœuds du maillage comme graine). Pour choisir l'ordre dans lequel les voisins d'un nœud sont visités, nous avons choisi d'utiliser l'ordre croissant de leurs distances, contrairement aux travaux de Pablo Barrio et Carlos Carreras qui utilisent le sens des aiguilles d'une montre. Pour les maillages utilisés, ce choix apporte une meilleure qualité de tri.
3. un tri utilisant l'algorithme de Cuthill-McKee inverse (RCM), tel que présenté dans la section 5.3. Pour implémenter ce tri nous avons utilisé la bibliothèque *C++ Boost Graph Library (BGL)*.
4. et un tri aléatoire (*Random*).

Ces quatre méthodes de tri se distinguent par plusieurs critères. Le tri réalisé par la courbe de remplissage repose uniquement sur les coordonnées géographiques des sommets du maillage. L'algorithme RCM utilise uniquement la topologie du maillage pour procéder au tri. Alors que le tri réalisé avec l'algorithme BFS utilise à la fois la topologie du maillage et les coordonnées

géographiques des sommets du maillage. Les algorithmes RCM et BFS ont le même objectif, qui est de réduire la largeur de bande de la matrice d'adjacence du maillage. Tandis que le tri réalisé avec la courbe de remplissage à pour objectif d'améliorer la localité des données en acceptant que la bande de la matrice d'adjacence du maillage soit très large. La méthode de tri aléatoire consiste uniquement à désordonner le maillage pour se comparer au pire cas.

La première expérimentation compare les performances du noyau de calcul en fonction des quatre méthodes de tri. L'objectif est d'observer l'effet de l'ordonnancement des données sur l'utilisation de la bande passante mémoire. Pour cela, nous avons utilisé la topologie *No Latency*, de manière à ne pas perturber les mesures par les effets de la latence d'accès aux données distantes. Nous avons fixé la taille des *chunks* à 64 emplacements (de 4 octets) et nous avons instancié 8 PE pour s'assurer d'exploiter au mieux la bande passante de la mémoire de l'accélérateur.

La deuxième expérimentation a pour objectif de comparer les performances de l'exécution du code TsunAWI sur un processeur massivement parallèle. Ce dernier est un processeur AMD EPYC 7702P qui intègre 64 cœurs, 128 threads et 256 Mio de mémoire cache L3. La mémoire cache et les cœurs du processeur sont répartis sur 8 circuits intégrés dans la même puce. De cette manière, cette architecture forme 8 domaines NUMA. L'ordonnancement des données peut impacter le nombre de défauts de cache provoqués par l'exécution du code et détermine le placement des données dans les différents domaines NUMA. Ainsi, si la méthode de tri apporte une bonne localité aux données, le nombre de défauts de cache sera réduit. Si la méthode de tri augmente la largeur de la bande passante de la matrice d'adjacence du maillage, la latence de certains accès peut être augmentée à cause des effets NUMA. L'exécution du code a été réalisée avec les 128 *threads* du processeur (*hyper-threading*).

Pour les deux expérimentations, le critère de performance est la vitesse de calcul. Pour chaque tri et chaque maillage, le résultat présenté est la vitesse de calcul normée par celle obtenue avec le tri utilisant la courbe de remplissage.

La figure 9.12 représente les résultats obtenus pour la première expérimentation. Nous pouvons observer que le tri avec la courbe de Hilbert, permet d'obtenir les meilleures performances pour tous les maillages. Le tri aléatoire conduit à une vitesse de calcul 10 fois inférieure, ce qui met en lumière l'importance du tri des données pour exploiter au mieux le noyau de calcul. Il est également intéressant d'observer que pour chaque méthode de tri, la valeur

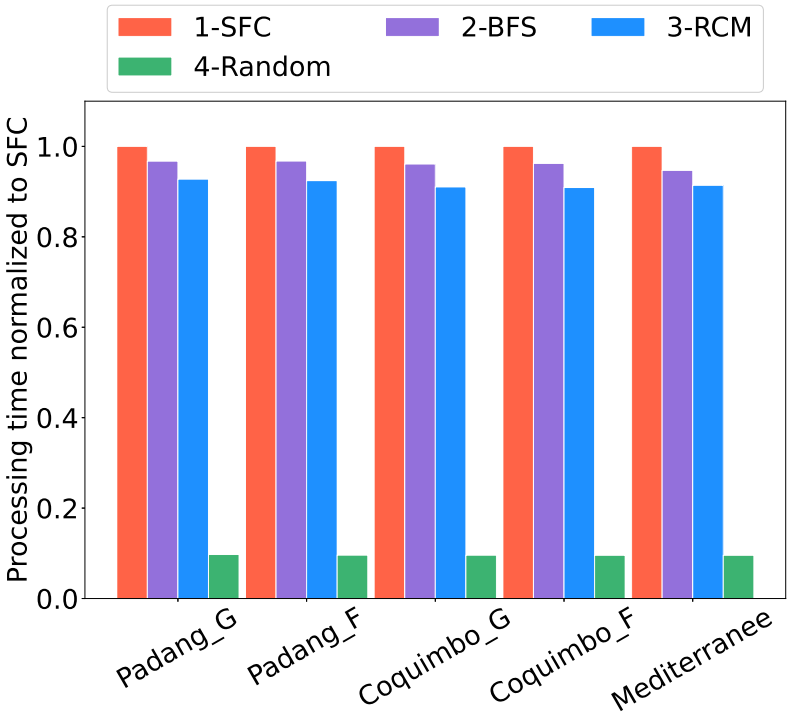


FIGURE 9.12 – Première expérimentation : évaluation du temps d'exécution du noyau de calcul *compute gradient* sur FPGA avec l'outil de simulation en fonction de l'ordre des données du maillage. Les temps d'exécution sont normalisés. Pour chaque maillage, la valeur de normalisation est le temps d'exécution obtenu pour le tri SFC.

normée de la vitesse de calcul est presque la même pour tous les maillages, malgré leurs différences de tailles. Cette observation indique que le modèle de programmation peut s'adapter à différentes méthodes de tri et à l'augmentation de la taille des jeux de données.

La figure 9.13 illustre les résultats obtenus pour la deuxième expérimentation. Le grand volume de mémoire cache offert par le processeur utilisé permet de contenir entièrement le jeu de données pour le maillage `Padang_G`. Pour les quatre autres maillages, le taille des jeux de données dépassent celles de la mémoire cache. Ainsi, les résultats pour le maillage *Padang_G* doivent être considérés différemment, étant donné que les accès mémoire sont moins couteux.

Comme pour la première expérimentation, le tri opéré avec la courbe de remplissage permet d'obtenir les meilleures performances parmi les quatre méthodes de tri. Cette concordance est importante car, en vue d'une exécution répartie entre des CPU et des FPGA, il est important que la méthode de tri

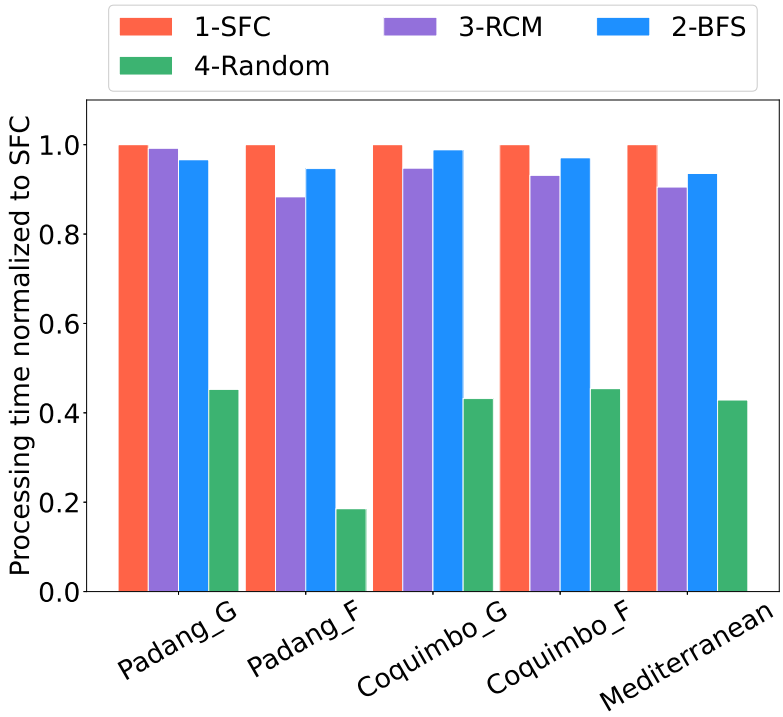


FIGURE 9.13 – Deuxième expérimentation : mesure du temps d'exécution du code TsunAWI complet sur un processeur AMD EPYC 7707P avec 128 threads en fonction de l'ordre des données du maillage. Les temps d'exécution sont normalisés. Pour chaque maillage, la valeur de normalisation est le temps d'exécution obtenu pour le tri SFC.

optimale soit la même pour les deux types de ressources.

Nous pouvons observer que le tri aléatoire induit un ralentissement du traitement bien moins important que pour la première expérimentation. Cette observation s'explique par l'architecture des mémoires caches des processeurs qui est plus adaptée pour gérer des accès mémoires fortement aléatoires. Premièrement, parce que les transferts de données entre les différents niveaux du système mémoire se font à la granularité d'une ligne de cache, ce qui représente un volume de données moins important que pour un *chunk*. En effet, la taille d'une ligne de cache est de 64 octets pour le processeur que nous avons utilisé, comme pour beaucoup d'autres processeurs. Pour la première expérimentation de cette sous-section, un *chunk* à un volume de 256 octets, soit 4 fois plus qu'une ligne de cache. Deuxièmement, parce que la taille de chaque cache de données L1 du processeur utilisé est de 2 Mio, alors que la taille cumulée de la fenêtre glissante et des deux tampons est de seulement 4608 octets. Ainsi, le processeur permet de contenir un plus grand volume de données à proximité

des unités de calcul.

Enfin, nous pouvons observer que contrairement à la première expérimentation, le tri BFS permet d'obtenir de meilleures performances que le tri RCM.

Conclusions

Ces deux expérimentations mettent en lumière l'importance de l'ordonnancement des données pour une application opérant sur des maillages non structurés. L'utilisation d'une courbe de remplissage pour définir l'ordre des sommets d'un maillage permet d'obtenir les meilleures performances pour l'exécution du code TsunAWI sur un CPU et pour l'exécution du noyau de calcul *compute gradient* sur un FPGA avec la MVP et l'architecture proposée. L'effet de l'ordonnancement des données est particulièrement marqué pour l'exécution sur un FPGA où l'on peut observer qu'un tri utilisant une courbe de remplissage de l'espace offre une vitesse de traitement 10 fois plus rapide qu'un tri aléatoire.

Discussion

Les expérimentations présentées dans les chapitres 8 et 9 ont évalué un modèle de programmation permettant à des noyaux de calcul instanciés sur un accélérateur reconfigurable d'initier l'accès à des données distribuées. Le modèle de programmation a pour objectif de masquer les latences d'accès à ces données. Le scénario d'expérimentation utilisé est le plus défavorable puisque la mémoire de l'accélérateur était vide au démarrage et que toutes les données devaient être transférées pendant l'exécution. Les deux cas d'études ont permis d'observer un comportement du modèle de programmation comparable.

Premièrement, lorsque le parallélisme augmente, un phénomène de famine de données se produit. Il s'explique par un goulot d'étranglement formé par la bande passante de la mémoire du FPGA. Les nouvelles gammes de cartes FPGA intégrant des mémoires HBM offrent une plus grande bande passante mémoire. Celles-ci devraient pouvoir repousser la formation de ce goulot d'étranglement et ainsi autoriser une plus grande vitesse de traitement pour les deux cas d'études.

Deuxièmement, lorsque la charge de travail des noyaux de calcul est suffisamment importante, le modèle de programmation permet de masquer efficacement les latences d'accès aux données distantes. Pour y parvenir, il utilise la superposition temporelle des flux de données transférés et des flux de données traités. Au démarrage, la mémoire du FPGA étant vide, le traitement des données peut s'initier seulement après la réception des premiers *chunks*. Ainsi, le temps de chargement de ces *chunks* ne peut pas être masqué. Si ce temps représente une part significative du temps d'exécution du noyau de calcul, alors le ralentissement de la vitesse de calcul est important. Les latences d'accès aux *chunks* dépendent majoritairement de la topologie du système, et par conséquent, ne peuvent pas être compressées. Ainsi, l'augmentation de la charge de travail est le seul moyen de réduire la part du temps de traitement

que représente le chargement des premiers *chunks*. En pratique, les applications de calcul scientifique produisent une charge de travail importante. Les matrices creuses les plus volumineuses de la collection de l'université de Floride [7] dépassent la centaine de millions d'éléments non nuls et sont issues de cas applicatifs réels. La taille du jeu de données complet utilisé pour la simulation de tsunami est au moins dix fois plus volumineuse que le sous-ensemble impliqué dans le calcul du gradient seulement. Pour l'exécution des quatre noyaux du code *TsunAWI*, chaque élément implique une centaine de calculs flottants par itération.

Troisièmement, la MVP logicielle agit comme un cache de données pour le FPGA. Ainsi, elle permet d'apporter de l'intelligence dans la gestion des données de manière transparente en conservant les *chunks* dans la mémoire du FPGA. Ce système peut implémenter différentes politiques de remplacement des données et plusieurs modèles de consistances mémoires en même temps pour minimiser les transferts de données tout en répondant aux exigences d'une application. Cependant, une bonne localité des données à la fois spatiale et temporelle est nécessaire pour exploiter efficacement le système. Cette assertion se vérifie pour toutes les applications de calcul. Le partitionnement en *chunks* des structures de données irrégulières et l'emploi d'une méthode de tri efficace pour les systèmes mémoires intégrant un cache permettent d'améliorer la localité des données.

7. `https://sparse.tamu.edu/`

Conclusion et perspectives

Conclusion

Contexte

Pour gagner en efficacité énergétique, les systèmes de calcul deviennent de plus en plus hétérogènes. Cette tendance s'observe dans les systèmes de calcul haute performance, notamment à travers l'architecture des superordinateurs du TOP500. Elle s'observe également à d'autres échelles, comme dans les systèmes embarqués pour les véhicules autonomes ou dans l'Internet des objets (*Internet of Things* (IoT)) avec des systèmes sur puces hétérogènes. L'hétérogénéité d'un système de calcul peut provenir de l'association de processeurs et de coprocesseurs. Dans ce cas, les coprocesseurs sont utilisés pour accélérer le traitement des portions d'une application dont l'intensité arithmétique est élevée. Cependant, l'hétérogénéité complexifie le modèle de programmation de ces systèmes, notamment du point de vue de la gestion des données.

Le sujet de cette thèse est de faciliter la programmation des systèmes hétérogènes dont les mémoires sont physiquement réparties. Pour y répondre, nous avons proposé d'étudier l'intégration d'accélérateurs reconfigurables dans une Mémoire Virtuellement Partagée (MVP) logicielle. Cette orientation des travaux de la thèse est motivée par deux observations exposées dans la première partie du manuscrit.

Le chapitre 2 a présenté une classification des algorithmes de calcul scientifique. Cette classification a mis en avant que la majorité des familles d'algorithmes exhibe, soit des motifs d'accès aux données irréguliers, soit des motifs de calcul complexes, ou les deux. Une analyse de l'état de l'art nous a montré que les FPGA sont le type de coprocesseur permettant d'obtenir le meilleur rendement énergétique pour le traitement de ces familles d'algorithmes. Cette

efficacité s'explique par la capacité d'adaptation de l'architecture des FPGA. Dans le même temps, ces familles d'algorithmes ont une intensité arithmétique relativement faible et par conséquent stressent les systèmes mémoires et d'interconnexion. Les applications qui utilisent ces algorithmes sont *memory-bound* : leurs performances sont limitées par celles du système mémoire et du système d'interconnexion. Une stratégie de gestion des données optimisée doit être adoptée pour réduire les transferts de données entre les différents niveaux de la hiérarchie mémoire.

La mémoire partagée est un paradigme pratique pour la programmation d'une application dont plusieurs fils d'exécution accèdent de manière aléatoire aux données partagées. Une MVP logicielle permet d'unifier des mémoires physiquement réparties dans un espace d'adressage logique. Ces systèmes sont en charge de répliquer les données et de les maintenir cohérentes. Une MVP logicielle agit comme un cache de données et peut apporter de l'intelligence de manière transparente dans la gestion des données.

Contributions

Les travaux de cette thèse ont abouti à trois contributions et à des expérimentations menées sur deux cas d'études.

Modèle d'intégration d'accélérateurs reconfigurables dans une MVP logicielle Pour faciliter la programmation des FPGA, nous avons proposé un modèle d'intégration d'accélérateurs reconfigurables dans une MVP logicielle. Celui-ci autorise les FPGA d'être initiateurs des accès aux données qu'ils traitent. Il permet une gestion locale de la mémoire de l'accélérateur et autorise les accès aux données avec une granularité adaptée aux besoins des applications.

Modèle de programmation pour noyaux de calcul irréguliers sur accélérateur reconfigurable dans un système distribué hétérogène La première contribution de cette thèse nous a conduis à proposer un modèle de programmation dans lequel les noyaux de calcul exécutés sur un accélérateur reconfigurable initient l'accès aux données distribuées. En masquant la distribution des données, une MVP logicielle peut conduire à des accès lents. Pour répondre à ce problème de latence, le modèle de programmation utilise

le préchargement des données afin de masquer les temps de transfert en superposant temporellement les flux de données transférés et les flux de données traités. Il vise à faciliter la programmation de noyaux de calcul irréguliers. Pour permettre ce préchargement, un partitionnement en *chunks* des structures de données est réalisé afin d'en abstraire l'irrégularité. Ce partitionnement peut être optimisé en appliquant une méthode de tri des données adaptée aux systèmes mémoires disposant d'un cache.

Outil de simulation Pour réaliser un prototype fonctionnel du modèle d'intégration d'accélérateurs reconfigurables dans une MVP logicielle et pour développer le modèle de programmation, nous avons conçu un outil de simulation. Cet outil utilise une méthode hybride. Elle consiste à reproduire l'activité d'un FPGA avec un moteur de simulation et à générer des latences d'accès aux données distribuées à partir d'une exécution réelle de la MVP logicielle. Cet outil nous a permis d'expérimenter le modèle de programmation pour en valider le comportement.

Expérimentations Nous avons utilisé deux cas d'études pour valider le modèle de programmation : la multiplication de matrices creuses et un code simulation de tsunami. Les résultats obtenus ont montré que le modèle de programmation permet de masquer efficacement les latences d'accès aux données distribuées lorsque la charge de travail est suffisamment importante pour minimiser le coût d'accès aux premières données. Ils ont montré également que la bande passante de la mémoire du FPGA forme un goulot d'étranglement lorsque le parallélisme des noyaux de calcul augmente. Ainsi, le modèle de programmation permet d'atteindre la puissance de calcul maximale autorisée par les performances de la mémoire du FPGA modélisé.

Perspectives

Les travaux menés dans cette thèse ouvrent la voie à plusieurs perspectives. Parmi celles-ci, deux ont été identifiées dans la section 4.5 comme propositions d'améliorations du modèle d'intégration d'accélérateurs reconfigurables dans une MVP logicielle. La première est d'intégrer le support pour la programmation évènementielle existant dans l'API SAT. Si le FPGA est émetteur d'évènements, ce support rendrait possible de superviser son activité. S'il est

souscripteur, le support pourrait permettre de réduire la fréquence d'horloge d'un noyau de calcul lorsqu'il est dans l'attente d'un évènement. La deuxième perspective est de supporter l'utilisation de mémoire HBM. Les expérimentations présentées dans la troisième partie du manuscrit ont montré que les vitesses de traitement pour les deux cas d'études avec un FPGA embarquant une mémoire de type DDR sont limités par la bande passante mémoire. Une mémoire HBM devrait repousser ce goulot d'étranglement.

La programmation d'un FPGA requiert une description matérielle avec un langage HDL. Ces langages paraissent beaucoup plus complexes que ceux utilisés pour programmer un processeur et ne favorisent pas la portabilité des codes. Les outils de synthèse haut niveau ont apporté une réponse à cette complexité en permettant de générer une description matérielle à partir d'un langage de programmation impératif généraliste (langage C) ou d'un langage de programmation orientée objet (langage C++). Une perspective pour le modèle de programmation est de l'intégrer dans un de ces outils. Cette intégration nécessite de pouvoir exprimer le flux de données qui traverse un noyau de calcul sous la forme de *chunks*. Elle doit également permettre de créer l'interface avec le module *FPGA-server*.

Enfin, une dernière perspective pour les travaux de cette thèse est d'expérimenter le modèle de programmation pour un panel plus large d'applications de calcul exhibant des motifs d'accès aux données irréguliers ou des méthodes de calcul complexes. Par exemple, expérimenter des algorithmes opérant des convolutions en trois dimensions. Les conclusions de ces expérimentations pourraient conduire à faire évoluer le modèle de programmation pour l'adapter à un nombre plus grand d'algorithmes.

Bibliographie

[1] Agullo (E.), Aumage (O.), Faverge (M.), Furmento (N.), Pruvost (F.), Sergent (M.) et Thibault (S. P.). – Achieving high performance on supercomputers with a sequential task-based programming model. *IEEE Transactions on Parallel and Distributed Systems*, 2017, pp. 1–1.

[2] Alen (E.), Chase (D.), Hallet (J.), Luchangco (V.), Maessen (J.-W.), Ryu (S.), Steele Jr (G. L.) et Tobin-Hochstadt (s.). – *The Fortress Language Specification.* – Rapport technique, Sun Microsystems, Inc., 2008. Version 1.0.

[3] Amza (C.), Cox (A.), Dwarkadas (S.), Keleher (P.), Lu (H.), Rajamony (R.), Yu (W.) et Zwaenepoel (W.). – Treadmarks : shared memory computing on networks of workstations. *Computer*, vol. 29, n 2, 1996, pp. 18–28.

[4] Anderson (D. W.), Sparacio (F. J.) et Tomasulo (R. M.). – The ibm system/360 model 91 : Machine philosophy and instruction-handling. *IBM Journal of Research and Development*, vol. 11, n1, 1967, pp. 8–24.

[5] Andres (E.), Widhalm (M.) et Caloto (A.). – Achieving high speed cfd simulations : Optimization, parallelization, and fpga acceleration for the unstructured dlr tau code. – In *Proceedings of the 2020 4th International Conference on High Performance Compilation, Computing and Communications*, 2009.

[6] Antoniu (G.) et Bougé (L.). – *DSM-PM2 : A Portable Implementation Platform for Multithreaded DSM Consistency Protocols.* – Research Report nRR-4108, LIP RR-2000-39, INRIA, LIP, 2001.

[7] Augonnet (C.). – *Scheduling Tasks over Multicore machines enhanced with acelerators : a Runtime System's Perspective.* – Theses, Université Bordeaux 1, 2011.

[8] Augonnet (C.), Aumage (O.), Furmento (N.), Thibault (S.) et Namyst (R.). – *StarPU-MPI : Task Programming over Clusters of Machines Enhanced with Accelerators.* – Research Report nRR-8538, INRIA, 2014.

[9] Augonnet (C.), Thibault (S.), Namyst (R.) et Wacrenier (P.-A.). – StarPU : a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation : Practice and Experience*, vol. 23, n 2, 2011, pp. 187–198.

[10] Ayguadé (E.), Badia (R. M.), Bellens (P.), Cabrera (D.), Duran (A.), Ferrer (R.), González (M.), Igual (F. D.), Jiménez-González (D.) et Labarta (J.). – Extending openmp to survive the heterogeneous multi-core era. *Int. J. Parallel Program.*, vol. 38, n 5-6, 2010, pp. 440–459.

[11] Bader (M.). – *Space-Filling Curves - An Introduction with Applications in Scientific Computing.* – Springer, 2013, *Texts in Computational Science and Engineering*, volume 9.

[12] Barrio (P.) et Carreras (C.). – Mesh traversal and sorting for efficient memory usage in scientific codes. – In Zhong (S.), Dou (D.) et Wang (Y.) (édité par), *30th IEEE International Performance Computing and Communications Conference, IPCCC 2011, Orlando, Florida, USA, November 17-19, 2011*, pp. 1–8. IEEE Computer Society, 2011.

[13] Barrio (P.), Carreras (C.), López (J. A.), Robles (Ó.), Jevtic (R.) et Sierra (R.). – Memory optimization in fpga-accelerated scientific codes based on unstructured meshes. *J. Syst. Archit.*, vol. 60, n 7, 2014, pp. 579–591.

[14] Beck (K.), Beedle (M.), van Bennekum (A.), Cockburn (A.), Cunningham (W.), Fowler (M.), Grenning (J.), Highsmith (J.), Hunt (A.), Jeffries (R.), Kern (J.), Marick (B.), Martin (R. C.), Mellor (S.), Schwaber (K.), Sutherland (J.) et Thomas (D.). – Manifesto for agile software development. – https ://agilemanifesto.org/, 2001.

[15] Bershad (B.), Zekauskas (M.) et Sawdon (W.). – The midway distributed shared memory system. – In *Digest of Papers. Compcon Spring*, pp. 528–537, 1993.

[16] Binkert (N. L.), Beckmann (B. M.), Black (G.), Reinhardt (S. K.), Saidi (A. G.), Basu (A.), Hestness (J.), Hower (D.), Krishna (T.), Sardashti (S.), Sen (R.), Sewell (K.), Altaf (M. S. B.), Vaish (N.), Hill (M. D.)

et Wood (D. A.). – The gem5 simulator. *SIGARCH Comput. Archit. News*, vol. 39, n2, 2011, pp. 1–7.

[17] Blumrich (M. A.), Li (K.), Alpert (R.), Dubnicki (C.), Felten (E. W.) et Sandberg (J.). – Virtual memory mapped network interface for the SHRIMP multicomputer. – In Patterson (D. A.) (édité par), *Proceedings of the 21st Annual International Symposium on Computer Architecture. Chicago, IL, USA, April 1994*, pp. 142–153. IEEE Computer Society, 1994.

[18] Buluç (A.), Fineman (J. T.), Frigo (M.), Gilbert (J. R.) et Leiserson (C. E.). – Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. – In auf der Heide (F. M.) et Bender (M. A.) (édité par), *SPAA 2009 : Proceedings of the 21st Annual ACM Symposium on Parallelism in Algorithms and Architectures, Calgary, Alberta, Canada, August 11-13, 2009*, pp. 233–244. ACM, 2009.

[19] Cantor (G.). – Ein beitrag zur mannigfaltigkeitslehre. *Journal für die reine und angewandte Mathematik*, vol. 84, 1878, pp. 242–258.

[20] CAPS Enterprise, Cray Inc., NVIDIA, and the Portland Group. – *The OpenACC Application Programming Interface.* – Rapport technique, 2011. Version 1.0.

[21] Carlson (W. W.), Draper (J. M.), Culler (D. E.), Yelick (K.), Brooks (E.) et Warren (K.). – *Introduction to UPC and language specification.* – Rapport technique, Technical Report CCS-TR-99-157, IDA Center for Computing Sciences, 1999.

[22] Carter (J. B.), Bennett (J. K.) et Zwaenepoel (W.). – Implementation and performance of munin. *SIGOPS Oper. Syst. Rev.*, vol. 25, n5, sep 1991, p. 152–164.

[23] Chamberlain (B.), Callahan (D.) et Zima (H.). – Parallel programmability and the chapel language. *The International Journal of High Performance Computing Applications*, vol. 21, n3, 2007, pp. 291–312.

[24] Charles (P.), Grothoff (C.), Saraswat (V.), Donawa (C.), Kielstra (A.), Ebcioglu (K.), von Praun (C.) et Sarkar (V.). – X10 : An object-oriented approach to non-uniform cluster computing. – In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, *OOPSLA '05*, OOPSLA '05, p. 519–538. Association for Computing Machinery, 2005.

[25] Choi (Y.), Cong (J.), Fang (Z.), Hao (Y.), Reinman (G.) et Wei (P.). – In-depth analysis on microarchitectures of modern heterogeneous CPU-FPGA platforms. *ACM Trans. Reconfigurable Technol. Syst.*, vol. 12, n 1, 2019, pp. 4 :1–4 :20.

[26] Choi (Y.-k.), Cong (J.), Fang (Z.), Hao (Y.), Reinman (G.) et Wei (P.). – A quantitative analysis on microarchitectures of modern CPU-FPGA platforms. – In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2016.

[27] Christodoulis (G.). – *Adapting a HPC runtime system to FPGAs. (Adaption d'un système HPC pour intégrer des FPGAs).* – Thèse de PhD, Grenoble Alpes University, France, 2019.

[28] Colella (P.). – Software engineering for scientific computing. – https ://www.krellinst.org/doecsgf/conf/2013/pres/pcolella.pdf, 2004. Présentation.

[29] Cong (J.), Fang (Z.), Gill (M.) et Reinman (G.). – PARADE : A cycle-accurate full-system simulation platform for accelerator-rich architectural design and exploration. – In Marculescu (D.) et Liu (F.) (édité par), *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2015, Austin, TX, USA, November 2-6, 2015*, pp. 380–387. IEEE, 2015.

[30] Corda (S.), Veenboer (B.), Awan (A. J.), Kumar (A.), Jordans (R.) et Corporaal (H.). – Near memory acceleration on high resolution radio astronomy imaging. – In *9th Mediterranean Conference on Embedded Computing, MECO 2020, Budva, Montenegro, June 8-11, 2020*, pp. 1–6. IEEE, 2020.

[31] Cudennec (L.). – Software-Distributed Shared Memory over Heterogeneous Micro-server Architecture. – In *Euro-Par 2017 : Parallel Processing Workshops*, *Lecture Notes in Computer Science*, volume 10659, pp. 366–377. Springer, 2017.

[32] Cudennec (L.). – Merging the publish-subscribe pattern with the shared memory paradigm. – In *Euro-Par 2018 : Parallel Processing Workshops*, *Lecture Notes in Computer Science*, volume 11339, pp. 469–480. Springer, 2018.

[33] Cudennec (L.). – Adaptive message passing polling for energy efficiency : Application to software-distributed shared memory over heterogeneous

computing resources. *Concurrency and Computation : Practice and Experience*, vol. 32, n24, 2020, p. e5960.

[34] Cudennec (L.). – Software-distributed shared memory for heterogeneous machines : Design and use considerations. *CoRR*, vol. abs/2009.01507, 2020.

[35] Cudennec (L.) et Trabelsi (K.). – Experiments using a software-distributed shared memory, MPI and 0mq over heterogeneous computing resources. – In *Euro-Par 2020 : Parallel Processing Workshops*, *Lecture Notes in Computer Science*, volume 12480, pp. 237–248. Springer, 2020.

[36] Cuthill (E.) et McKee (J.). – Reducing the Bandwidth of Sparse Symmetric Matrices. – In *Proceedings of the 1969 24th National Conference*, *ACM '69*, ACM '69, p. 157–172, New York, NY, USA, 1969. Association for Computing Machinery.

[37] Davis (T. A.) et Hu (Y.). – The university of Florida sparse matrix collection. *ACM Trans. Math. Softw.*, vol. 38, n1, 2011, pp. 1 :1–1 :25.

[38] De Wael (M.), Marr (S.), De Fraine (B.), Van Cutsem (T.) et De Meuter (W.). – Partitioned global address space languages. *ACM Comput. Surv.*, vol. 47, n4, 2015.

[39] Dennard, Robert H.and Gaensslen (F. H.), Yu (H.-N.), Rideout (V. L.), Bassous (E.) et LeBlanc (A. R.). – Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, vol. 9, n5, 1974.

[40] Dolbeau (R.), Bihan (S.) et Bodin (F.). – Hmpp : A hybrid multi-core parallel programming environment. *Workshop on General Purpose Processing on Graphics Processing Units (GPGPU 2007)*, 2007.

[41] Edwards (H. C.), Trott (C. R.) et Sunderland (D.). – Kokkos : Enabling manycore performance portability through polymorphic memory access patterns. *J. Parallel Distributed Comput.*, vol. 74, n12, 2014, pp. 3202–3216.

[42] Eeckhout (L.). – Heterogeneity in response to the power wall. *IEEE Micro*, vol. 35, n4, 2015.

[43] Eisenstat (S. C.), Gursky (M. C.), Schultz (M. H.) et H.Sherman (A.). – *Yale Sparse Matrix Package. I. The Symmetric Codes.* – Rapport tech-

nique nADA047724, Yale University, Department of Computer Science, 1977. Research report.

[44] Escobar (F. A.), Chang (X.) et Valderrama (C.). – Suitability analysis of fpgas for heterogeneous platforms in HPC. *IEEE Trans. Parallel Distributed Syst.*, vol. 27, n2, 2016, pp. 600–612.

[45] Esmaeilzadeh (H.), Blem (E.), St. Amant (R.), Sankaralingam (K.) et Burger (D.). – Dark silicon and the end of multicore scaling. – In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, *ISCA '11*, ISCA '11, 2011.

[46] Etancelin (J.-M.) et Kraus (J.). – Multi-GPU programming with OpenACC and MPI. – In *GPU Technology Conference*, Amsterdam, Netherlands, septembre 2016.

[47] Feng (L.), Liang (H.), Sinha (S.) et Zhang (W.). – Heterosim : A heterogeneous CPU-FPGA simulator. *IEEE Comput. Archit. Lett.*, vol. 16, n1, 2017, pp. 38–41.

[48] Forum (M. P. I.). – *MPI : A Message-Passing Interface Standard.* – Rapport technique, Message Passing Interface Forum, 1994. Report.

[49] Forum (T. O.). – *OpenMP Fortran Application Program Interface.* – Rapport technique, The OpenMP Forum, 1997. Version 1.0.

[50] Gabriel (E.), Fagg (G. E.), Bosilca (G.), Angskun (T.), Dongarra (J. J.), Squyres (J. M.), Sahay (V.), Kambadur (P.), Barrett (B.), Lumsdaine (A.), Castain (R. H.), Daniel (D. J.), Graham (R. L.) et Woodall (T. S.). – Open MPI : goals, concept, and design of a next generation MPI implementation. – In Kranzlmüller (D.), Kacsuk (P.) et Dongarra (J. J.) (édité par), *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary, September 19-22, 2004, Proceedings*, *Lecture Notes in Computer Science*, volume 3241, pp. 97–104. Springer, 2004.

[51] Gao (Y.) et Zhang (P.). – A survey of homogeneous and heterogeneous system architectures in high performance computing. – In *2016 IEEE International Conference on Smart Cloud, SmartCloud 2016, New York, NY, USA, November 18-20, 2016*, pp. 170–175. IEEE Computer Society, 2016.

[52] Geer (D.). – Chip makers turn to multicore processors. *Computer*, vol. 38, n5, 2005, pp. 11–13.

[53] Gelado (I.), Cabezas (J.), Navarro (N.), Stone (J. E.), Patel (S. J.) et Hwu (W. W.). – An asymmetric distributed shared memory model for heterogeneous parallel systems. – In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2010, Pittsburgh, Pennsylvania, USA, March 13-17, 2010*, pp. 347–358, 2010.

[54] Goodman (J. R.). – Using cache memory to reduce processor-memory traffic. *SIGARCH Comput. Archit. News*, vol. 11, n3, 1983, p. 124–131.

[55] Goubier (T.), Martinovic (J.), Dubrulle (P.), Ganne (L.), Louise (S.), Martinovic (T.) et Slaninová (K.). – Real-Time Model of Computation over HPC/Cloud Orchestration - The LEXIS Approach. – In *Proceedings of the 14th International Conference on Complex, Intelligent and Software Intensive Systems (CISIS-2020)*, *Advances in Intelligent Systems and Computing*, volume 1194, 2020.

[56] Goubier (T.), Rakowsky (N.) et Harig (S.). – Fast Tsunami Simulations for a Real-Time Emergency Response Flow. – In *2020 IEEE/ACM HPC for Urgent Decision Making, UrgentHPC@SC 2020*, pp. 21–26. IEEE, 2020.

[57] Gropp (W.), Lusk (E. L.), Doss (N. E.) et Skjellum (A.). – A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Comput.*, vol. 22, n6, 1996, pp. 789–828.

[58] Group. (T. P.). – *PGI Fortran & C : Accelerator Programming Model* . – Rapport technique, The Portland Group., 209. Version 1.0.

[59] Gustavson (F. G.). – Two Fast Algorithms for Sparse Matrices : Multiplication and Permuted Transposition. *ACM Trans. Math. Softw.*, vol. 4, n3, 1978, pp. 250–269.

[60] Hannak (H.), Hochstetter (H.) et Blochinger (W.). – A hybrid parallel barnes-hut algorithm for gpu and multicore architectures. – In *Euro-Par 2013 Parallel Processing*, pp. 559–570, 2013.

[61] Hilbert (D.). – Ueber die stetige abbildung einer line auf ein flächenstück. *Mathematische Annalen*, vol. 38, 1891, pp. 459–460.

[62] Iftode (L.), Dubnicki (C.), Felten (E. W.) et Li (K.). – Improving release-consistent shared virtual memory using automatic update. – In *Proceedings of the Second International Symposium on High-Performance Computer Architecture, San Jose, CA, USA, February 3-7, 1996*, pp. 14–25. IEEE Computer Society, 1996.

[63] Kamalavasan (K.), Mudalige (G. R.), Reguly (I. Z.) et Fahmy (S. A.). – High-level FPGA accelerator design for structured-mesh-based explicit numerical solvers. – In *35th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2021, Portland, OR, USA, May 17-21, 2021*, pp. 1087–1096. IEEE, 2021.

[64] Kaxiras (S.), Klaftenegger (D.), Norgren (M.), Ros (A.) et Sagonas (K.). – Turning centralized coherence and distributed critical-section execution on their head : A new approach for scalable distributed shared memory. – In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*. Association for Computing Machinery, 2015.

[65] Khronos OpenCL Working Group. – *The OpenCL Specification.* – Rapport technique, 2008. Version 1.0.

[66] Kilby (J. S. C.). – Turning potential into realities : The invention of the integrated circuit (nobel lecture). *ChemPhysChem*, vol. 2, n8-9, 2001.

[67] Kirchsteiger (C. M.), Schweitzer (H.), Trummer (C.), Steger (C.), Weiss (R.) et Pistauer (M.). – A software performance simulation methodology for rapid system architecture exploration. – In *15th IEEE International Conference on Electronics, Circuits and Systems, ICECS 2008, St. Julien's, Malta, August 31 2008-September 3, 2008*, pp. 494–497. IEEE, 2008.

[68] Kontothanassis (L. I.), Hunt (G. C.), Stets (R.), Hardavellas (N.), Cierniak (M.), Parthasarathy (S.), Jr. (W. M.), Dwarkadas (S.) et Scott (M. L.). – Vm-based shared memory on low-latency, remote-memory-access networks. – In Pleszkun (A. R.) et Mudge (T. N.) (édité par), *Proceedings of the 24th International Symposium on Computer Architecture, Denver, Colorado, USA, June 2-4, 1997*, pp. 157–169. ACM, 1997.

[69] Lee (J.) et Sato (M.). – Implementation and performance evaluation of xcalablemp : A parallel programming language for distributed memory

systems. – In *2010 39th International Conference on Parallel Processing Workshops*, pp. 413–420, 2010.

[70] Lee (S.) et Eigenmann (R.). – Openmpc : Extended openmp programming and tuning for gpus. – In *Conference on High Performance Computing Networking, Storage and Analysis, SC 2010, New Orleans, LA, USA, November 13-19, 2010*, pp. 1–11. IEEE, 2010.

[71] Lee (S.), Kim (J.) et Vetter (J. S.). – Openacc to FPGA : A framework for directive-based high-performance reconfigurable computing. – In *2016 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2016, Chicago, IL, USA, May 23-27, 2016*, pp. 544–554. IEEE Computer Society, 2016.

[72] Lee (Y.), Waterman (A.), Cook (H.), Zimmer (B.), Keller (B.), Puggelli (A.), Kwak (J.), Jevtic (R.), Bailey (S.), Blagojevic (M.), Chiu (P.), Avizienis (R.), Richards (B. C.), Bachrach (J.), Patterson (D. A.), Alon (E.), Nikolic (B.) et Asanovic (K.). – An agile approach to building RISC-V microprocessors. *IEEE Micro*, vol. 36, n2, 2016, pp. 8–20.

[73] Lenormand (E.), Cudennec (L.) et Charles (H.-P.). – Unification des mémoires réparties dans un système hétérogène avec accélérateur reconfigurable : exposé de principe. – In *Conférence d'informatique en Parallélisme, Architecture et Système (Compas'2019)*, Anglet, France, juin 2019. LIUPPA - Laboratoire d'Informatique de l'Université de Pau et des Pays de l'Adour (EA 3000).

[74] Lenormand (E.), Goubier (T.), Cudennec (L.) et Charles (H.-P.). – A combined fast/cycle accurate simulation tool for reconfigurable accelerator evaluation : application to distributed data management. – In *2020 IEEE International Workshop on Rapid System Prototyping (RSP)*, pp. 1–7, Hamburg, Germany, septembre 2020. IEEE.

[75] Li (K.). – IVY : A shared virtual memory system for parallel computing. – In *Proceedings of the International Conference on Parallel Processing, ICPP '88, The Pennsylvania State University, University Park, PA, USA, August 1988. Volume 2 : Software*, pp. 94–101. Pennsylvania State University Press, 1988.

[76] Liang (T.), Feng (L.), Sinha (S.) et Zhang (W.). – PAAS : A system level simulator for heterogeneous computing architectures. – In Santambrogio

(M. D.), Göhringer (D.), Stroobandt (D.), Mentens (N.) et Nurmi (J.) (édité par), *27th International Conference on Field Programmable Logic and Applications, FPL 2017, Ghent, Belgium, September 4-8, 2017*, pp. 1–8. IEEE, 2017.

[77] Liu (W.-H.) et Sherman (A. H.). – Comparative analysis of the cuthill - mckee and the reverse cuthill-mckee ordering algorithms for sparse matrices. *SIAM Journal on Numerical Analysis*, vol. 13, n 2, 1976, pp. 198–213.

[78] Moore (G. E.). – Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff. *IEEE Solid-State Circuits Society Newsletter*, vol. 11, n 3, 2006.

[79] Moore (G. E.). – Progress in digital integrated electronics [technical literaiture, copyright 1975 ieee. reprinted, with permission. technical digest. international electron devices meeting, ieee, 1975, pp. 11-13.]. *IEEE Solid-State Circuits Society Newsletter*, vol. 11, n 3, 2006.

[80] Morton (G. M.). – *A computer oriented geodetic data base and a new technique in file sequencing.* – Rapport technique, IBM, 1966. Research report.

[81] Mustafa Kemal, Badrul et Putra, H. – Exploring key issues related to tsunami shelter in padang city - indonesia. *E3S Web Conf.*, vol. 156, 2020, p. 04002.

[82] Nakao (M.), Murai (H.), Shimosaka (T.), Tabuchi (A.), Hanawa (T.), Kodama (Y.), Bokut (T.) et Sato (M.). – Xcalableacc : Extension of xcalablemp pgas language using openacc for accelerator clusters. – In *Proceedings of the First Workshop on Accelerator Programming Using Directives*, *WACCPD '14*, WACCPD '14, p. 27–36, 2014.

[83] Nelson (J.), Holt (B.), Myers (B.), Briggs (P.), Ceze (L.), Kahan (S.) et Oskin (M.). – Latency-tolerant software distributed shared memory. – In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pp. 291–305. USENIX Association, juillet 2015.

[84] Numrich (R. W.) et Reid (J.). – Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, vol. 17, n 2, aug 1998, p. 1–31.

[85] Oleksiak (A.), Kierzynka (M.), Piatek (W.), Agosta (G.), Barenghi (A.), Brandolese (C.), Fornaciari (W.), Pelosi (G.), Cecowski (M.), Plestenjak

(R.), Cinkelj (J.), Porrmann (M.), Hagemeyer (J.), Griessl (R.), Lachmair (J.), Peykanu (M.), Tigges (L.), vor dem Berge (M.), Christmann (W.), Krupop (S.), Carbon (A.), Cudennec (L.), Goubier (T.), Philippe (J.), Rosinger (S.), Schlitt (D.), Pieper (C.), Adeniyi-Jones (C.), Setoain (J.), Ceva (L.) et Janssen (U.). – M2DC - modular microserver datacentre with heterogeneous hardware. *Microprocess. Microsystems*, vol. 52, 2017.

[86] Papamichael (M.), Hoe (J. C.) et Mutlu (O.). – FIST : A fast, lightweight, fpga-friendly packet latency estimator for noc modeling in full-system simulations. – In Marculescu (R.), Kishinevsky (M.), Ginosar (R.) et Chatha (K. S.) (édité par), *NOCS 2011, Fifth ACM/IEEE International Symposium on Networks-on-Chip, Pittsburgh, Pennsylvania, USA, May 1-4, 2011*, pp. 137–144. ACM/IEEE Computer Society, 2011.

[87] Peano (G.). – Sur une courbe, qui remplit toute une aire plane. *Mathematische Annalen*, vol. 36, 1890, pp. 157–160.

[88] Peng (B.), Wang (T.), Jin (X.) et Wang (C.). – An accelerating solution for n-body MOND simulation with fpga-soc. *Int. J. Reconfigurable Comput.*, vol. 2016, 2016, pp. 4592780 :1–4592780 :10.

[89] Rabenseifner (R.), Hager (G.) et Jost (G.). – Hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes. – In *2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, pp. 427–436, 2009.

[90] Rinard (M.), Scales (D.) et Lam (M.). – Heterogeneous parallel programming in jade. – In *Supercomputing '92 :Proceedings of the 1992 ACM/IEEE Conference on Supercomputing*, pp. 245–256, 1992.

[91] Ringlein (B.), Abel (F.), Ditter (A.), Weiss (B.), Hagleitner (C.) et Fey (D.). – Programming reconfigurable heterogeneous computing clusters using mpi with transpilation. – In *2020 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*, pp. 1–9, 2020.

[92] Samanta (R.), Bilas (A.), Iftode (L.) et Singh (J. P.). – Home-based SVM protocols for SMP clusters : Design and performance. – In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture, Las Vegas, Nevada, USA, January 31 - February 4, 1998*, pp. 113–124. IEEE Computer Society, 1998.

[93] Sanaullah (A.) et Herbordt (M. C.). – FPGA HPC using opencl : Case study in 3d FFT. – In *Proceedings of the 9th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies, HEART 2018, Toronto, ON, Canada, June 20-22, 2018*, pp. 7 :1–7 :6. ACM, 2018.

[94] Sastry (S. P.), Kultursay (E.), Shontz (S. M.) et Kandemir (M. T.). – Improved cache utilization and preconditioner efficiency through use of a space-filling curve mesh element- and vertex-reordering technique. *Eng. Comput.*, vol. 30, n4, 2014, pp. 535–547.

[95] Shao (Y. S.), Xi (S. L.), Srinivasan (V.), Wei (G.) et Brooks (D. M.). – Co-designing accelerators and soc interfaces using gem5-aladdin. – In *49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2016, Taipei, Taiwan, October 15-19, 2016*, pp. 48 :1–48 :12. IEEE Computer Society, 2016.

[96] Slota (M.). – Opencapi technology. – https ://openpowerfoundation.org/wp-content/uploads/2018/04/Myron-Slota.pdf, 2018.

[97] Smith (J. E.). – A study of branch prediction strategies. – In *Proceedings of the 8th Annual Symposium on Computer Architecture*, p. 135–148, 1981.

[98] Snyder (W.). – Verilator : the fast free verilog simulator, 2012.

[99] Soltaniyeh (M.), Martin (R. P.) et Nagarakatte (S.). – Synergistic CPU-FPGA acceleration of sparse linear algebra. *CoRR*, vol. abs/2004.13907, 2020.

[100] Sozzo (E. D.), Rabozzi (M.), Tucci (L. D.), Sciuto (D.) et Santambrogio (M. D.). – A scalable FPGA design for cloud n-body simulation. – In *29th IEEE International Conference on Application-specific Systems, Architectures and Processors, ASAP 2018, Milano, Italy, July 10-12, 2018*, pp. 1–8. IEEE Computer Society, 2018.

[101] Sozzo (E. D.), Tucci (L. D.) et Santambrogio (M. D.). – A highly scalable and efficient parallel design of n-body simulation on FPGA. – In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS Workshops 2017, Orlando / Buena Vista, FL, USA, May 29 - June 2, 2017*, pp. 241–246. IEEE Computer Society, 2017.

[102] Springmeyer (R.), Still (C.), Schulz (M.), Ahrens (J.), Hemmert (S.), Minnich (R.), McCormick (P.), Ward (L.) et Knoll (D.). – *From Petascale to Exascale : Eight Focus Areas of R&D Challenges for HPC Simulation Environments.* – Rapport technique, OSTI, 2011.

[103] Srivastava (N. K.), Jin (H.), Liu (J.), Albonesi (D. H.) et Zhang (Z.). – Matraptor : A sparse-sparse matrix multiplication accelerator based on row-wise product. – In *53rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2020, Athens, Greece, October 17-21, 2020*, pp. 766–780. IEEE, 2020.

[104] Stan (O.), Cudennec (L.) et Syoën (L.). – Attribute-Based Encryption and Its Application to a Software-Distributed Shared Memory. – In *17th International Conference on Security and Cryptography*, pp. 625–631, Lieusaint - Paris, France, juillet 2020. SCITEPRESS - Science and Technology Publications.

[105] Stone (J. E.), Gohara (D.) et Shi (G.). – Opencl : A parallel programming standard for heterogeneous computing systems. *Comput. Sci. Eng.*, vol. 12, n 3, 2010, pp. 66–73.

[106] Tomasulo (R. M.). – An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, vol. 11, n 1, 1967, pp. 25–33.

[107] Trabelsi (K.), Cudennec (L.) et Bennour (R.). – Application topology definition and tasks mapping for efficient use of heterogeneous resources. – In *Euro-Par 2019 : Parallel Processing Workshops*, *Lecture Notes in Computer Science*, volume 11997, pp. 258–269. Springer, 2019.

[108] Ubal (R.), Jang (B.), Mistry (P.), Schaa (D.) et Kaeli (D. R.). – Multi2sim : a simulation framework for CPU-GPU computing. – In Yew (P.), Cho (S.), DeRose (L.) et Lilja (D. J.) (édité par), *International Conference on Parallel Architectures and Compilation Techniques, PACT '12, Minneapolis, MN, USA - September 19 - 23, 2012*, pp. 335–344. ACM, 2012.

[109] Ventroux (N.), Guerre (A.), Sassolas (T.), Moutaoukil (L.), Blanc (G.), Bechara (C.) et David (R.). – Sesam : An mpsoc simulation environment for dynamic application processing. – In *2010 10th IEEE International Conference on Computer and Information Technology*, pp. 1880–1886, 2010.

[110] Vogel (P.). – *Shared Virtual Memory for Heterogeneous Embedded Systems on Chip.* – Thèse de PhD, École polytechnique fédérale de Zurich, 2018.

[111] Wang (Q.), Danilov (S.), Sidorenko (D.), Timmermann (R.), Wekerle (C.), Wang (X.), Jung (T.) et Schröter (J.). – The Finite Element Sea Ice-Ocean Model (FESOM) v.1.4 : formulation of an ocean general circulation model. *Geoscientific Model Development*, vol. 7, n 2, 2014, pp. 663–693.

[112] Wicaksana (A.), Charif (A.), Andriamisaina (C.) et Ventroux (N.). – Hybrid prototyping methodology for rapid system validation in HW/SW co-design. – In *2019 Conference on Design and Architectures for Signal and Image Processing, DASIP 2019, Montreal, QC, Canada, October 16-18, 2019*, pp. 35–40. IEEE, 2019.

[113] Wienke (S.), Springer (P. L.), Terboven (C.) et an Mey (D.). – Openacc - first experiences with real-world applications. – In *Euro-Par 2012 Parallel Processing - 18th International Conference, Euro-Par 2012, Rhodes Island, Greece, August 27-31, 2012. Proceedings*, *Lecture Notes in Computer Science*, volume 7484, pp. 859–870. Springer, 2012.

[114] Xilinx. – *AXI Reference Guide*, 2011. v13.1.

[115] Xilinx. – *7 Series FPGAs Memory Resources : User Guide*, 2019. v1.14.

[116] Xilinx. – *VC707 Evaluation Board for the Virtex-7 FPGA : User Guide*, 2019. v1.8.

[117] Xilinx. – *DMA/Bridge Subsystem for PCI Express : Product Guide*, 2021. v4.1.

[118] Yelick (K. A.), Semenzato (L.), Pike (G.), Miyamoto (C.), Liblit (B.), Krishnamurthy (A.), Hilfinger (P. N.), Graham (S. L.), Gay (D.), Colella (P.) et Aiken (A.). – Titanium : A high-performance java dialect. *Concurr. Pract. Exp.*, vol. 10, n 11-13, 1998, pp. 825–836.

[119] Zeni (A.), O'Brien (K.), Blott (M.) et Santambrogio (M. D.). – Optimized implementation of the HPCG benchmark on reconfigurable hardware. – In Sousa (L.), Roma (N.) et Tomás (P.) (édité par), *Euro-Par 2021 : Parallel Processing - 27th International Conference on Parallel and Distributed Computing, Lisbon, Portugal, September 1-3, 2021, Pro-*

ceedings, *Lecture Notes in Computer Science*, volume 12820, pp. 616–630. Springer, 2021.

[120] Zenker (E.), Worpitz (B.), Widera (R.), Huebl (A.), Juckeland (G.), Knüpfer (A.), Nagel (W. E.) et Bussmann (M.). – Alpaka - an abstraction library for parallel kernel acceleration. *CoRR*, vol. abs/1602.08477, 2016.

[121] Zhang (G.), Attaluri (N.), Emer (J. S.) et Sánchez (D.). – Gamma : leveraging gustavson's algorithm to accelerate sparse matrix multiplication. – In Sherwood (T.), Berger (E.) et Kozyrakis (C.) (édité par), *ASPLOS '21 : 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, pp. 687–701. ACM, 2021.

[122] Zhou (S.), Stumm (M.), Li (K.) et Wortman (D.). – Heterogeneous distributed shared memory. *IEEE Transactions on Parallel and Distributed Systems*, vol. 3, n5, 1992, pp. 540–554.

[123] Zohouri (H. R.), Podobas (A.) et Matsuoka (S.). – Combined spatial and temporal blocking for high-performance stencil computation on fpgas using opencl. – In Anderson (J. H.) et Bazargan (K.) (édité par), *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA 2018, Monterey, CA, USA, February 25-27, 2018*, pp. 153–162. ACM, 2018.

Liste des figures

Liste des tableaux

Liste des acronymes

ACP	*Accelerator Coherency Port*
ALU	*Arithmetic Logic Unit*
API	*Application Programming Interface*
AXI	*Advanced eXtensible Interface*
BFS	*Breadth First Search*
BRAM	*Block RAM*
COTS	*Components-Off-the Shelf*
CPU	*Central Processing Unit*
CSB	*Compressed Sparse Blocks*
CSC	*Compressed Sparse Column*
CSR	*Compressed Sparse Row*
DDR	*Double Data Rate*
DMA	*Direct Memory Access*
DRAM	*Dynamic Random Access Memory*
DSM	Distributed Shared Memory
DSP	*Digital signal processor*
EDA	*Electronic design automation*
FIFO	*First-In, First-Out*
FLOP	*Floating-Point Operations*
FPGA	*Field-Programmable Gate Array*
GPGPU	*General-purpose computing on Graphics Processing Unit*

GPU	*Graphics Processing Unit*
HBM	*High Bandwidth Memory*
HDL	*Hardware Description Language*
HPC	*High Performance Computing*
LAN	*Local-Area Network*
LRU	*Least Recently Used*
MMU	*Memory Management Unit*
MPI	*Message Passing Interface*
MpSoC	*Multiprocessor system on a chip*
MVP	Mémoire Virtuellement Partagée
NUMA	*Non Uniform Memory Access*
OpenCAPI	*Open Coherent Accelerator Processor Interface*
PCI	*Peripheral Component Interconnect*
PE	*Processing Element*
PGAS	*Partitioned Global Address Space*
RCM	*Reverse Cuthill-McKee*
RDMA	*Remote Direct Memory Access*
RMA	*Remote Memory Access*
RPC	*Remote Procedure Call*
RTL	*Register Transfer Level*
SDRAM	*Synchronous Dynamic Random Access Memory*
SIMD	*Single Instruction, Multiple Data*
TDP	*Thermal Design Power*

Glossaire

MESI	Protocole de cohérence à quatre états: Modified indiquant que les données sont accessibles par un seul processus et qu'ils les a modifié; Exclusive indiquant qu'un seul processus à accès aux données, mais qu'il ne les a pas modifié.; Shared indiquant que plusieurs processus ont un accès en lecture au données; Invalid indiquant que la version des données n'est pas à jour.
big.LITTLE	Architecture de système sur puce hétérogène se composant d'un premier ensemble de cœurs de processeur à faible consommation énergétique et à faible puissance de calcul et d'un second ensemble de cœurs plus puissants et plus énergivores.
largeur de bande d'une matrice creuse	largeur de bande d'une matrice creuse A est la somme de sa largeur basse de bande k_b et de sa largeur haute de bande k_h. k_b est le plus petit entier, supérieur où égal à 0, tel que tout élément a_{ij} où $j < i - k_b$ est nul. k_h est le plus petit entier, supérieur où égal à 0, tel que tout élément a_{ij} où $j > i + k_h$ est nul.
memory-bound	Caractéristique d'une application dont les performances sont limitées par celles du système mémoire.