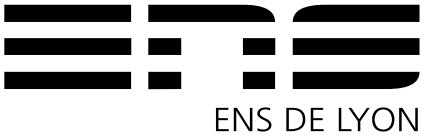
Portage des chaînes de traitement sonar sur architecture hétérogène : conception et évaluation d'un environnement de programmation basé sur les tâches moldables

Pierre-Etienne Polet



THESE

en vue de l'obtention du grade de Docteur, délivré par

l'ÉCOLE NORMALE SUPÉRIEURE DE LYON

École Doctorale N° 512

INFOMATHS Informatique Mathématiques

Discipline : Informatique

Soutenue publiquement le 03 Avril 2024, par :

Pierre-Étienne POLET

Portage des chaînes de traitement sonar sur architecture hétérogène : conception et évaluation d'un environnement de programmation basé sur les tâches moldables

---

Devant le jury composé de :

COTI, Camille	Professeure	ETS Montréal	Rapporteure
THIBAULT, Samuel	Professeur des Universités	Université de Bordeaux, Inria	Rapporteur
RAFFIN, Bruno	Directeur de Recherche	Inria Grenoble	Éxaminateur
FANTAR, Ramy	Ingénieur	Thales DMS	Éxaminateur
GAUTIER, Thierry	Chargé de recherche	Inria Lyon	Directeur de thèse

Sommaire

Table des figures

Liste des tableaux

Résumé

L'augmentation des besoins en calcul dans les chaînes de traitement SONAR incite au choix d'architectures hétérogènes à base de GPGPU. La complexité de ces architectures rend l'implémentation d'algorithmes difficile pour des personnes qui ne sont pas à la fois spécialistes du domaine d'application et de la programmation parallèle.

Cette thèse propose de répondre à cette problématique en exploitant des notions de programmation par tâches. Des méthodes d'analyse statique du code nous ont permis de regrouper les appels de fonction sur un GPU afin de limiter certains surcoûts en augmentant la granularité des tâches. Afin d'étendre cette approche pour permettre l'exploitation de plusieurs GPU tout en contrôlant l'utilisation mémoire, nous avons étudié un modèle à base de tâches moldables. Celui-ci a été décliné en une nouvelle directive OpenMP qui permet d'unifier plusieurs directives de parallélisation plus anciennes. Dans ce modèle les tâches moldables ou les sous-tâches conservent la capacité d'avoir des dépendances. La conception d'un prototype de support exécutif pour la gestion de ces tâches moldables s'est attachée à gérer l'équilibrage de charge sur une architecture hétérogène ainsi qu'à définir un algorithme pour détecter les dépendances entre de telles tâches. Des expérimentations sur un algorithme de formation de voies issu des chaînes de traitement SONAR et sur l'algorithme de factorisation de Cholesky ont mis en avant l'intérêt de la méthode et certaines faiblesses des choix d'implémentation.

Mots-clés: Programmation à base de tâches, Calcul hétérogène, Support d'exécution, Factorisation de cholesky, Groupement de fonction, GPU.

Abstract

The increasing computational demands in SONAR processing chains lead to the choice of heterogeneous architectures based on GPGPU. The complexity of these architectures makes algorithm implementation challenging for individuals who are not specialists in both the application domain and parallel programming.

This thesis aims to address this issue by leveraging task-based programming concepts. Static code analysis methods allowed us to group function calls on a GPU to limit certain overheads by increasing task granularity.

To extend this approach to enable the utilization of multiple GPUs while controlling memory usage, we explored a moldable task model. This was instantiated into a new OpenMP directive that unifies several older parallelization directives. In this model, moldable tasks or sub-tasks retain the ability to have dependencies. The design of a prototype executive support for managing these moldable tasks focused on load balancing on a heterogeneous architecture and defining an algorithm to detect dependencies between such tasks.

Experiments on a SONAR processing beamforming algorithm and the Cholesky factorization algorithm highlighted the method's benefits and some weaknesses in the implementation choices.

Keywords: Task based programming, Heterogeneous computing, Runtime system, Cholesky factorisation, Function batch, GPU.

Chapitre 1

Introduction

1.1 Motivation

Les chaînes de traitement SONAR regroupent l'ensemble des opérations nécessaires pour détecter, classifier et tracer des objets évoluant sur ou sous l'eau. L'algorithme de formation des voies, qui a pour objectif d'estimer l'énergie sonore reçue en fonction d'une direction, est central dans la chaîne de traitement. En particulier, cela s'explique par les besoins en puissance de calcul relativement élevés par rapport au reste de la chaîne de traitement.

Les dimensions physiques telles que la taille des antennes et les fréquences d'échantillonnage des données, ainsi que les dimensions applicatives telles que le nombre de directions observées et la finesse des voies, définissent les besoins en puissance de calcul de la chaîne de traitement. Cependant, l'évolution des systèmes SONAR actuels tend à augmenter l'ensemble de ces dimensions, ce qui induit un besoin en puissance de calcul important pour traiter l'ensemble des données. De plus, le cadre opérationnel d'utilisation des SONAR implique des contraintes de temps réel molles (soft-real time) pour le traitement des données, ainsi que la capacité à s'exécuter dans un environnement avec des contraintes de consommation énergétique et d'encombrement.

L'évolution des besoins et des contraintes incitent à faire évoluer l'architecture historique des serveurs de traitements basés sur des noeuds de calculs multi-CPU. Les coprocesseurs de type GPU sont des candidats intéressants pour répondre à ces contraintes, de par la puissance de calcul importante disponible en rapport avec la consommation énergétique. Quelques résultats sur l'utilisation de GPU dans un contexte de traitement SONAR sont disponibles [32] [60] [13], néanmoins le domaine reste peu exploré.

Néanmoins, une exploitation efficace des GPU reste un défi. D'une part, l'aspect fortement parallèle de l'architecture complique l'implémentation de kernels par des personnes non spécialistes. De même, les interactions entre le système et plusieurs coprocesseurs contrôlés de manière asynchrone ne sont pas triviales. La problématique d'offrir des outils permettant d'exploiter efficacement les architectures fortement parallèles, seules ou dans des nœuds de calcul, est en évolution rapide aussi bien du côté de l'industrie que de la recherche. D'autres architectures pourraient avoir leurs intérêts, tels que les FPGA qui permettent une maîtrise fine des latences des calculs et une spécialisation importante. De plus, ces composants étant fréquemment utilisés dans l'industrie du traitement du signal, certaines équipes sont déjà spécialisées dans leur utilisation. L'introduction de ces nouveaux composants dans les nœuds de calcul induit une forte hétérogénéité, ce qui nécessitera des outils et des méthodes pour pouvoir exploiter les plateformes d'exécution.

1.2 Problématique

La complexification des nœuds de calculs nécessite de s'abstraire des architectures et de mettre en œuvre des solutions de programmation de haut niveau, séparant les préoccupations des spécialistes de différents domaines. En particulier, les numériciens conçoivent l'algorithme en fonction des aspects mathématiques et physiques du problème à résoudre. Les spécialistes du HPC ont pour rôle de paralléliser

les algorithmes, tandis que les spécialistes du hardware implémentent des solutions de bas niveau pour exploiter au mieux le matériel.

Dans le cadre de ces travaux de thèse, nous avons identifié trois problématiques :

1. La définition d'un outil pour automatiser certaines optimisations, en particulier la capacité de regrouper des calculs similaires entre eux pour mieux tirer parti des architectures de type GPU. L'objectif est de permettre aux numériciens d'implémenter de manière classique l'algorithme tout en permettant d'adapter la granularité du problème à une exécution sur GPU par une annotation du code.
2. La définition d'un modèle pour exploiter le parallélisme des applications et architectures, permettant d'adapter le grain de calculs aux ressources pour respecter les contraintes mémoires et utiliser l'ensemble du matériel à disposition. La capacité à exploiter des fonctions de bibliothèques externes telles que cuBLAS est aussi primordiale pour conserver les performances de ces implémentations. Cela s'est traduit par l'adaptation d'un modèle de tâches moldables avec la proposition d'une extension d'une directive OpenMP de génération de tâches pour supporter ce modèle.
3. La conception d'un support exécutif adapté aux tâches moldables en ayant la capacité d'exploiter des architectures hétérogènes multi-CPU/multi-GPU tout en gérant la répartition de la charge, les transferts de données et la gestion des dépendances entre ces tâches.

1.3 Objectifs et contributions

Les travaux de thèse ont consisté à étudier la réalisation d'un algorithme de traitement du signal sur des architectures hétérogènes et à la définition d'outils théoriques, conceptuels et logiciels pour permettre une exécution efficace. Ce travail fait le pivot entre les solutions historiques à base de CPU et de nouvelles solutions de traitement plus hétérogènes, avec pour perspective de mettre à profit ces acquis pour porter un ensemble d'algorithmes de traitement du signal utilisés par Thales sur des plateformes hétérogènes.

Les différentes contributions de ces travaux sont les suivantes :

— Le portage manuel et l'optimisation d'un algorithme de formation de voies pour une exécution sur GPU, permettant un speedup de 10 entre une exécution naïve sur GPU et une exécution optimisée sur le même GPU.
— La conception d'un outil permettant d'effectuer automatiquement certaines optimisations précédentes à l'aide d'annotations sur un code provenant des équipes de numériciens. L'objectif principal est d'augmenter la granularité des tâches exécutées sur le GPU. Ces optimisations permettent d'obtenir automatiquement un speedup de 7.5 par rapport à la version naïve.
— L'introduction d'une directive de génération de tâches à OpenMP pour exprimer la moldabilité des tâches dans ce runtime, avec l'ajout supplémentaire de clauses permettant de définir les accès mémoires des sous-tâches.
— L'implémentation d'un runtime d'ordonnancement, sous forme de prototype, exécutant des tâches moldables sur des architectures hétérogènes.
— La mise en œuvre d'un algorithme permettant de détecter rapidement les dépendances entre régions mémoire dans le contexte de ce runtime.
— Une série d'expérimentations sur différentes plateformes hétérogènes de Thales et de Grid5000 [5], permettant de mettre en avant le comportement des optimisations automatiques et du runtime.

1.4 Plan du document

Pour la suite du document, les travaux seront présentés dans l'ordre suivant : les bases des algorithmes de traitements ciblés dans cette thèse et une présentation des architectures de traitement sont faites dans le chapitre 2. L'introduction de l'outil d'optimisation automatique de code annoté pour une exploitation du GPU sera présentée en chapitre 3. Il sera appliqué à un algorithme de formation de voies pour justifier les gains de performances atteignables. Le chapitre 4 aborde les problèmes de granularité introduits par le chapitre précédent, en proposant d'ajouter une notion de moldabilité à la syntaxe OpenMP. Il présente une nouvelle directive, sa syntaxe, un schéma d'implémentation et des utilisations potentielles. Le chapitre

5 présente l'implémentation d'un runtime de tâche moldable, permettant de mettre en œuvre les concepts introduits précédemment et une série d'expériences avec des algorithmes de multiplication de matrices par batch, de formation de voies et de factorisation de Cholesky. Finalement, nous résumons les travaux et ouvrons les perspectives dans le chapitre 6.

Chapitre 2

Contexte

Ces travaux se sont déroulés dans le cadre d'un partenariat entre Thales et INRIA. La première section 2.1 présentera les chaînes de traitements SONAR telles qu'utilisées à Thales. Et justifiera les choix d'algorithmes à prendre en compte dans cette thèse. La section 2.2 présentera les contraintes opérationnelles des applications Thales et les choix d'architectures qui en découlent. La section 2.3 présentera les co-processeurs. Finalement, la section 2.4 reviendra plus en détail sur l'algorithme classique de formation de voies introduit dans la section 2.1.

2.1 Traitement SONAR

Le traitement SONAR regroupe l'ensemble des méthodes pour traiter des signaux sonores sous-marins dans l'objectif de détecter, localiser et classifier des objets sous l'eau ou proches de la surface. Ces méthodes peuvent servir à des objectifs variés, dans le cadre civil on note par exemple la localisation de bancs de poissons pour la pêche, la cartographie sous-marine. La majorité des traitements SONAR sont utilisés à des fins militaires, on s'en sert entre autres pour détecter des sous-marins, des navires ou des aéronefs proches de la surface.

2.1.1 Chaines actives et passives

Une chaîne de traitement est une suite d'opérations effectuées sur des données d'entrée, en principe les signaux des hydrophones, pour obtenir un résultat donné. Différentes chaînes de traitement sont utilisées en fonction des antennes à disposition, des ressources de calcul disponibles, des objectifs visés et des contraintes extérieures. On distingue deux types de cibles à identifier :

— Les bruiteurs, ils vont générer un signal sonore, ce sont par exemple des poissons, des bateaux ou des sous-marins.

— Les objets inertes, par exemple les mines, les fonds marins, les épaves.

Deux grandes catégories de chaînes de traitement sont utilisées : les chaînes de traitement actives et les chaînes de traitement passives.

Chaîne de traitement active. Le principe du traitement actif est d'émettre un ou plusieurs sons et d'analyser les échos, il est souvent comparé aux méthodes de positionnement des chauves-souris ou de certains mammifères marins. Les réflexions du son sur les objets sous-marins vont générer des échos qui permettent de déduire la position et le déplacement des objets sous-marins ou en surface. La chaîne de SONAR actif nécessite de disposer de systèmes capables d'émettre des signaux sonores et d'une étape pour définir la forme du signal à envoyer en fonction de ce que l'on veut observer. L'émission du signal actif empêche toute utilisation furtive de ce type de chaîne de traitement, en particulier, elles sont rarement utilisées par des sous-marins qui ont pour objectif de rester discrets. En revanche, le signal émis permet de détecter des objets inertes, de plus, les échos ont une intensité proportionnelle à celle du signal émis, ce

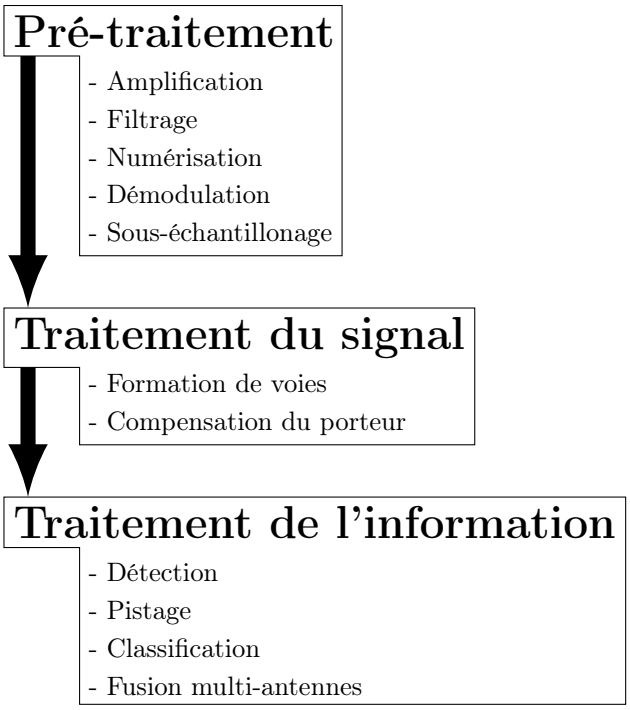


FIGURE 2.1 – Chaîne SONAR passive

qui permet de les distinguer facilement du bruit de fond marin. Cela facilite les traitements d'acquisition et d'analyse du signal.

Chaîne de traitement passive. Le traitement passif écoute seulement les sons émis par des bruiteurs, on se base sur les fréquences émises pour classifier le bruiteur. La position et la vitesse de déplacement sont souvent définies à l'aide de plusieurs mesures et de méthodes de triangulation. Contrairement à la chaîne active, aucun son n'est émis, on limite donc les objets étudiés aux bruiteurs. Ces chaînes sont en principe utilisées par les sous-marins car elles permettent de conserver la furtivité. Le traitement du signal est en revanche plus complexe, car les sons des bruiteurs peuvent être faibles par rapport au bruit ambiant. Cela nécessite aussi des antennes avec plus de capteurs pour extraire les signaux du bruit.

Pour la suite, nous allons nous concentrer sur la chaîne de traitement passive, car elle représente la majorité des besoins en calculs dans les systèmes d'informations des navires. En particulier, elle est utilisée pour effectuer une surveillance continue de l'environnement du navire disposant du Sonar. Une chaîne de traitement SONAR est présentée en figure 2.1. La chaîne est décomposée en trois grandes étapes : le pré-traitement qui consiste à préparer les données analogiques sorties des hydrophones en données numériques prêtes à être utilisées pour le traitement. La seconde étape, appelée "traitement du signal", consiste à estimer l'énergie reçue par les hydrophones pour chaque fréquence et pour chaque direction, elle sera détaillée en sous-section 2.1.2. Enfin, la dernière étape est appelée "traitement des données", elle consiste au traitement des informations énergétiques pour classifier, localiser et traquer les bruiteurs pertinents, elle sera détaillée en sous-section 2.1.3.

2.1.2 Traitement du signal

Les opérations de traitement du signal ont pour but d'estimer l'intensité des signaux reçus en fonction de leurs fréquences et de leurs directions. Pour cela, on va utiliser deux étapes : une formation des voies qui effectue ses opérations dans le repère du porteur (le sous-marin) et une compensation des mouvements du porteur qui consiste à replacer dans le même repère toutes les voies calculées dans un intervalle donné.

Une voie représente le signal reçu par une antenne en provenance d'une direction ou d'un point donné. L'algorithme de formation de voies est un filtrage spatial qui a pour objectif d'estimer les voies suivant plusieurs directions données. L'algorithme exploite les interférences entre les signaux reçus par plusieurs capteurs hydrophoniques. Cette étape de la chaîne de traitement est considérée comme la partie la plus coûteuse en termes de puissance de calcul. Elle est caractérisée par de nombreux paramètres tels que la forme et les dimensions de l'antenne, la plage de fréquence traitée et le nombre de voies à observer. Cet algorithme étant le cœur de la majorité des chaînes de traitement, il sera l'algorithme cible dans le cadre des travaux de la thèse. Une description détaillée est présentée en section 2.4.

La compensation du mouvement des porteurs consiste généralement à faire une interpolation des résultats en sortie de la formation de voies en fonction des déplacements du porteur, donc de l'antenne, entre les différentes mesures. Cette étape peut être fusionnée avec la formation de voies.

2.1.3 Traitement des données

Le traitement des données regroupe toutes les étapes qui suivent le traitement du signal et qui permettent de fournir des données utilisables par un opérateur. Les traitements les plus basiques peuvent se restreindre à extraire une voie spécifique des données pour générer un signal audio à l'opérateur. Ou à la génération d'une heatmap qui montre l'énergie reçue dans chaque direction en fonction du temps.

Les étapes suivantes sont historiquement réalisées directement par les opérateurs mais peuvent être automatisées en partie ou totalement : la détection des bruiteurs "pertinents", le pistage dans le temps et dans l'espace de ces bruiteurs, la classification des bruiteurs et le regroupement d'informations de différentes antennes ou de différents porteurs.

Alors que les opérations du traitement du signal sont assez proches de l'algèbre linéaire, le traitement des données regroupe des types de traitement plus variés tels que le calcul de normes, la détection des maximums locaux, les parcours de graphes et les méthodes de traitement des données ou d'apprentissage automatique. En revanche, l'étape de détection des bruiteurs "pertinents" réduit grandement l'espace de travail, ce qui fait que les étapes suivantes se font sur une quantité de données faible et nécessitent donc une puissance de calcul peu préoccupante par rapport à la formation de voies évoquée en section précédente.

C'est pourquoi, la suite des travaux va uniquement se concentrer sur l'algorithme de formation de voies détaillé en section 2.4.

2.2 Contraintes opérationnelles

Cette section va discuter des contraintes spécifiques aux développements logiciels et matériels dans le cadre des activités à Thales.

2.2.1 Calcul embarqué

Les produits de traitement SONAR sont principalement conçus pour être embarqués. Les porteurs cibles sont distingués en fonction de leurs tailles :

— Les objets autonomes qui disposent d'une autonomie énergétique limité. Ces porteurs ont des contraintes de calculs embarqués classiques.
— Les navires qui contrairement aux drones ont à disposition des sources d'énergie beaucoup plus importantes.

De ces cibles découlent deux contraintes classiques : la consommation énergétique et l'encombrement.

Consommation énergétique. Les objets autonomes rencontrent une problématique évidente en termes de consommation énergétique : la consommation du système de traitement des données affecte l'autonomie, et donc les missions réalisables. Pour les navires, la problématique diffère : la source d'énergie est beaucoup plus importante et la consommation des serveurs de calcul est faible par rapport aux besoins de la propulsion et du système de vie de l'équipage. Cependant, les calories générées par les calculs doivent

être dissipées, et les pompes nécessaires à cette dissipation entraînent une augmentation du niveau sonore du navire.

Dans le cas des sous-marins, qui doivent rester silencieux pour leurs missions, la consommation énergétique des serveurs doit être limitée pour des raisons de furtivité plutôt que pour des raisons d'autonomie.

Encombrement. Les objets autonomes et les navires ont des dimensions limitées par rapport aux puissances opératives demandées. L'objectif de limiter le volume des serveurs de traitement est donc un point important.

2.2.2 Temps réel

Les traitements SONAR sont généralement utilisés pour surveiller l'environnement du navire en temps réel, également pour prendre des décisions rapides dans le cas d'un évitement d'obstacle ou d'engagement de combat. En revanche, les données sont destinées à une prise de décision par un opérateur humain. On a donc des contraintes temporelles souples (*soft real-time*) sur la chaine de traitement, par exemple on peut vouloir une sortie à une image par seconde avec une latence de l'ordre de la seconde entre l'acquisition des premières données hydrophoniques et la sortie de l'étape de traitement du signal.

Les latences tolérées pour la partie de traitement des données sont plus importantes, car on utilise plusieurs itérations successives de l'étape de traitement du signal.

Certains calculs sont effectués sans contrainte de temps réel : par exemple, le recalcul d'une formation de voies à partir de données antérieurement acquises avec des paramètres spécifiques. Dans ce cas, les opérations ne sont pas contraintes par l'acquisition des données, mais l'objectif est d'effectuer le calcul le plus rapidement possible, avec une faible contrainte de latence.

2.2.3 Dimension des problèmes

La dimension des problèmes est fortement dépendante de la taille de l'antenne et du traitement que l'on souhaite effectuer. Ces dimensions imposent des contraintes de débit et de puissance de calcul nécessaires. Ces paramètres sont impactés par les dimensions suivantes : le nombre d'hydrophones n_{hydro}, la fréquence d'échantillonnage frq, le nombre de voies observées n_{obs}, le nombre de fréquences différentes observées n_{frq}.

Flux de données. Le débit d'entrée de l'étape de formation de voies est proportionnel au nombre d'hydrophones multiplié par la fréquence d'échantillonnage : $n_{hydro} \times frq$. Sur des antennes récentes, il peut atteindre plusieurs dizaines de Gigabits par seconde. De même, le débit de sortie est égal au nombre de voies observées multiplié par le nombre de fréquences observées multiplié par la fréquence de génération des données. En pratique, le nombre de voies observées est largement supérieur au nombre d'hydrophones. En revanche, les voies sont généralement intégrées sur une durée de l'ordre de la seconde, ce qui permet d'avoir un débit de sortie proche du débit d'entrée.

Les forts besoins en puissance de calcul peuvent imposer l'utilisation de différents serveurs de calcul pour effectuer un calcul complet. Dans ce cas, ces débits importants doivent transiter par un réseau. L'acquisition de tels débits, combinée aux contraintes de temps réel, nécessite une attention particulière lors de l'implémentation des communications et de la priorité des applications sur les différents cœurs du CPU.

Limitations des CPU. Les caractéristiques des traitements évoqués dans ce chapitre imposent une puissance de calcul importante. De plus, on souhaite réserver une partie des coeurs CPU pour maintenir des performances correctes sur les aspects réseau.

Ces différentes contraintes nécessitent l'utilisation de co-processeurs en plus du CPU. Dans notre cas, ils peuvent permettre de libérer des cœurs CPU pour les applications réseau, ou d'exécuter avec une meilleure efficacité énergétique certaines parties de la chaîne de traitement, comme les opérations d'algèbre linéaire.

2.3 Co-processeurs

2.3.1 Généralités

Les co-processeurs sont des composants de calculs distincts du CPU, avec les avantages suivants.
— Ils permettent de soulager le CPU d'une partie des calculs.
— Ils permettent d'avoir à disposition une architecture différente plus efficace que le CPU pour certains types de calculs.
— Ils peuvent avoir une mémoire dédiée avec des débits plus importants que celui permis par les canaux DDR d'un CPU.

En revanche, les co-processeurs sont souvent connectés au CPU via un lien PCI express, qui a un débit limité et qui peut devenir un goulot d'étranglement dans certains cas. La limitation de ce bus de donnée est levée sur certaines architectures telles que les processeurs IBM Power8/Power9 qui intègrent un bus NVLINK, utilisés par exemple dans le supercalculateur SUMMIT [42]. Le bus Infinity fabric [52] d'AMD permet aussi un plus grand débit entre les CPU et les GPU, néanmoins ces technologies nécessitent des cartes mère adaptées qui ne sont pas forcément adaptées aux contraintes "embarquées" des SONAR. Les cartes de calcul visent à réduire cette limitation en groupant le CPU et le GPU sur une même carte porteuse : le Nvidia Grace Hopper [49] interconnecte les deux avec un bus NVLINK, tandis que le AMD MI300 [1] utilise une mémoire HBM commune pour les deux composants.

Une deuxième limitation des co-processeurs est leur programmabilité. Chaque type de co-processeur dispose d'une architecture différente et peut nécessiter de maîtriser des notions avancées de calcul parallèle. En particulier, cela nécessite de gérer plusieurs mémoires, de devoir pipeliner des opérations et de programmer avec des API asynchrones. De plus, chaque constructeur fournit une API dédiée pour utiliser son matériel, et les API unifiées telles que OpenCL sont parfois supportées à minima.

Certains co-processeurs sont également intégrés avec le CPU, souvent orientés pour des applications à faible consommation. Dans ce cas, les bancs mémoires sont partagés et on perd l'avantage d'avoir une bande passante élevée au profit de ne plus avoir besoin de gérer la cohérence des données et les communications PCIe entre deux bancs mémoires distincts.

2.3.2 FPGA

Un premier type de co-processeur sont les FPGA. Ils sont disponibles sous la forme de carte PCIe tels que la gamme Alveo de Xilinx. Ce type d'architecture comporte un circuit logique programmable qui peut être reconfiguré à la volée. Contrairement au CPU ou au GPU qui ont une architecture fixe, le FPGA permet de générer des blocs de calcul dédiés avec une architecture désignée par le développeur.

L'avantage est de pouvoir créer des architectures spécifiques dédiées pour traiter un problème précis qui n'est pas adapté à d'autres architectures. Le design sur mesure de l'architecture du FPGA permet aussi de chaîner les opérations directement et d'obtenir des latences d'exécution faibles et maîtrisées, de même pour la consommation énergétique.

Les FPGA ont été largement utilisés pour les traitements SONAR [33] et RADAR. Ils sont encore largement utilisés et des développements récents existent [62] pour des antennes de tailles réduites.

Bien que les FPGA aient été étudiés dans le cadre de la veille technologique liée à la thèse, ils ne seront pas utilisés dans les expérimentations ni évoqués comme cibles d'exécution potentielles pour les raisons suivantes : les performances théoriques sont beaucoup plus faibles que celles des GPU pour des tâches classiques telles que l'algèbre linéaire. De plus, la programmation et l'utilisation de ce type de matériel sont laborieuses, en particulier dans le cadre de cartes utilisées comme des co-processeurs. Il est difficile de trouver des ingénieurs maîtrisant à la fois les spécificités des FPGA et celles liées au système d'exploitation et à la programmation asynchrone permettant d'exploiter la carte efficacement.

2.3.3 GPGPU

Les coprocesseurs cibles de cette thèse sont les Graphic Processing Units (GPU), les unités de traitement graphiques sont utilisées pour effectuer des calculs généraux depuis le milieu des années 2000.

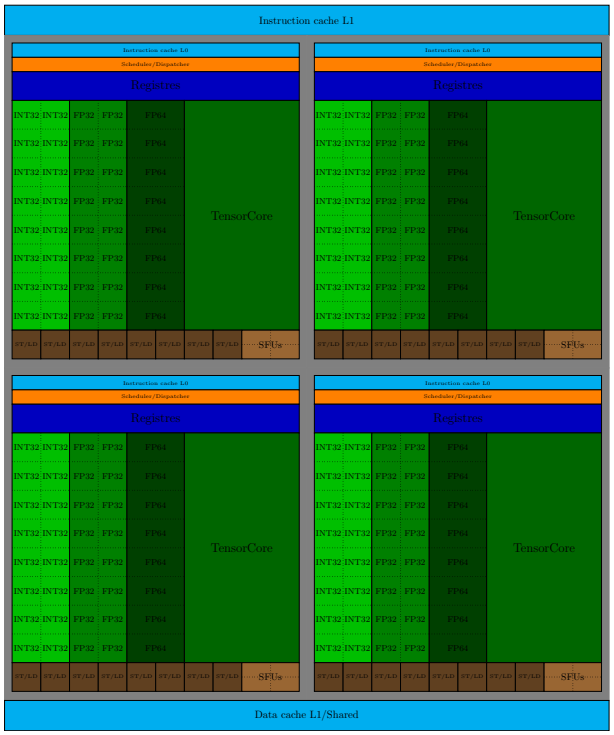


FIGURE 2.2 – GA100 Streaming Multiprocessor

Pour présenter l'architecture des GPU, nous allons nous baser sur la puce Nvidia GA100 [48] présentée en figures 2.3 et 2.2.

Architecture. L'architecture GPU se distingue de celle d'un CPU par son grand nombre d'unités de calcul, il est conçu pour fonctionner selon un principe Single Instruction Multiple Data (SIMD). Cela permet de mettre en commun les structures de contrôle. La structure des GPUs Nvidia est fortement liée à la hiérarchie mémoire. La figure 2.2 présente l'élément de base du GPU : le `streaming-multiprocesseur`, il comporte un cache L1 et quatre `compute units`.

Les `compute units` sont ce qui se rapproche le plus d'un cœur CPU, elles comportent des unités de contrôle pour traiter et répartir les instructions, un cache d'instruction, des registres et un ensemble d'unités de calculs. Les unités de calculs sont spécialisées pour différents types de calculs basiques : arithmétique entière ou flottante, des unités de load/store. Des unités plus exotiques peuvent être présentes, par exemple : les TensorCore qui permettent d'effectuer des multiplications de matrices de manière plus efficace. On notera que le nombre de cœurs d'un GPU est généralement associé au nombre d'unités arithmétiques FP32 de la puce, dans notre exemple chaque `compute unit` comporte donc 16 cœurs et chaque `streaming-multiprocesseur` comporte 64 cœurs.

La figure 2.3 montre la vue d'ensemble de la puce GA100, elle comporte 128 `streaming-multiprocesseur` reliés à un cache L2 commun pour un total de 8192 coeurs. Le `GigaThread Engine` est une structure de contrôle qui a pour objectif de répartir le travail entre les différents `streaming-multiprocesseur`.

De plus, le GPU comporte de nombreuses interconnexions pour communiquer à des débits élevés, en particulier les interfaces avec la mémoire HBM qui permettent d'obtenir des débits de l'ordre de 2 TO/s, ce qui est plus rapide d'un facteur 4 que les débits possibles via les mémoires DDR communément utilisées par les CPU.

Les architectures des GPU d'autres constructeurs, par exemple AMD, sont similaires et les légères différences sont masquées par le modèle de programmation.

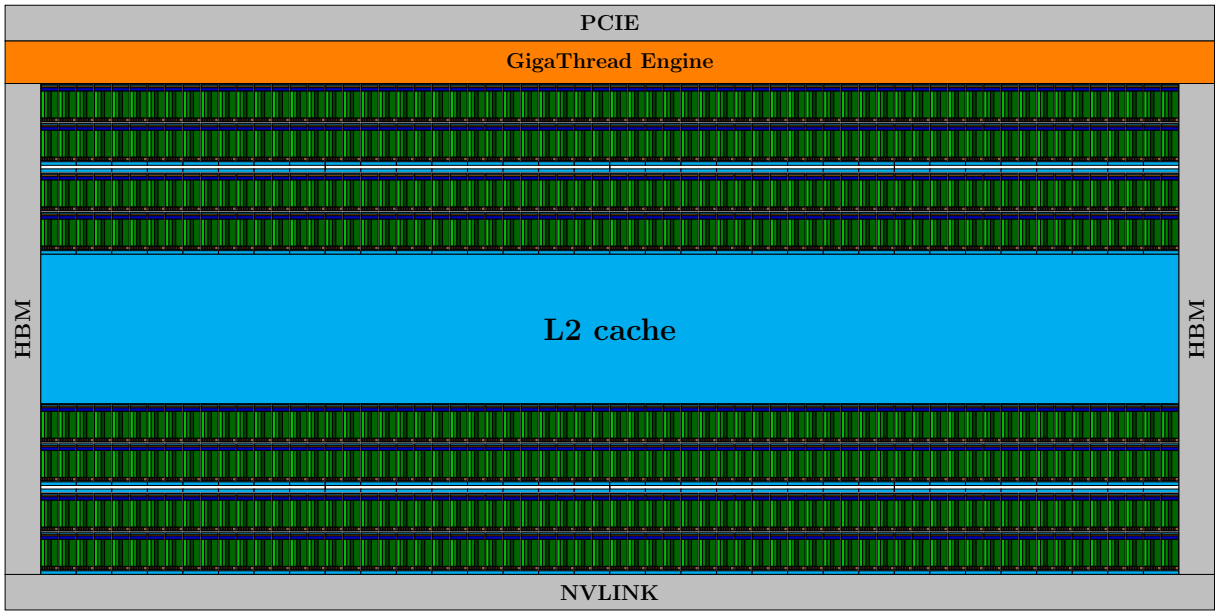


FIGURE 2.3 – Architecture du GPU Nvidia GA100

Modèle de programmation. La mise en commun des unités de contrôle et le grand nombre de cœurs d'un GPU impliquent de devoir utiliser des méthodes particulières pour exploiter le matériel. Lorsque l'utilisateur voudra exécuter une fonction (appelée `kernel`) sur le GPU, il va soumettre une `grille` de calculs au runtime CUDA. Cette grille représente un ensemble de threads qui vont exécuter la même fonction sur le GPU : tous les threads auront les mêmes arguments en entrée, néanmoins ils ont connaissance de leur position dans la grille grâce à l'accès à certains registres.

La grille est découpée en blocs de threads par l'utilisateur, les threads d'un même bloc vont s'exécuter sur le même `streaming-multiprocesseur` ce qui permet de tirer avantage du cache commun et d'instructions de synchronisation entre ces threads. Les blocs sont eux aussi divisés en groupes de 32 threads appelés `warp`, tous les threads d'un warp s'exécutent sur la même `compute unit` et en général ils s'exécutent de manière synchrone (la même instruction au même cycle). Les architectures récentes permettent néanmoins une désynchronisation entre les threads.

Ce modèle de programmation permet d'exécuter un grand nombre de threads avec des unités de contrôle simplifiées car tous les threads vont exécuter les mêmes instructions.

Programmation. Les programmes exploitant les GPUs sont implémentés à l'aide d'environnements de développement spécifiques tels que CUDA [18], HIP [38] ou OpenCL [50]. Ils fournissent des langages ou des extensions de langages spécifiques pour développer les kernels, ainsi qu'une API et un runtime permettant d'interagir avec le co-processeur depuis une application.

En revanche, la spécificité du modèle de programmation peut rendre l'implémentation de programmes efficaces sur GPU compliquée. La clef pour exploiter efficacement est d'avoir des données et des instructions à disposition des unités de calcul en permanence. Pour ce faire, il faut prendre en compte les différents niveaux de parallélisme du modèle lors de l'implémentation du programme.

Premièrement, l'implémentation des kernels doit prendre en compte les spécificités d'exécution d'un warp, par exemple : les données accédées par les threads d'un même warp doivent être contiguës pour tirer parti du cache L1, les threads du warp ne doivent pas diverger dans leur exécution car la `compute unit` ne va exécuter qu'une seule instruction par cycle pour les 32 threads du warp. Les architectures récentes limitent les impacts de la divergence en permettant d'exécuter en parallèle des instructions divergentes.

Deuxièmement, la ou les grilles en cours d'exécution sur le GPU doivent permettre d'occuper un maximum d'unités de calculs. On peut définir l'occupation du GPU de plusieurs manières : par exemple le nombre de `streaming-multiprocesseur` où un bloc est en cours d'exécution par rapport au nombre

de `streaming-multiprocesseur` du GPU. Une autre définition, plus fine, est de regarder le nombre d'instructions exécutées par cycle pour un type d'unité de calcul donnée : par exemple le pipeline d'arithmétique FP32. Pour maximiser ces métriques, il faut entre autres que les grilles de calculs exécutées sur le GPU aient suffisamment de blocs pour saturer tous les `streaming-multiprocesseurs`, que les blocs exécutent suffisamment de threads pour saturer les `compute units` et, enfin, que chacun des threads d'un warp ait du travail à effectuer.

Troisièmement, il faut être capable d'exécuter des kernels en continu sur le GPU. Il faut donc que le programme soumette des exécutions au runtime avant la fin des exécutions des kernels précédents, et dans le cas où les kernels ont besoin de données, qu'elles soient chargées au bon moment sur le GPU. Les GPUs permettent les transferts de données entre le CPU et le GPU en parallèle de l'exécution des kernels sur le GPU grâce à l'utilisation de Direct-Memory-Access (DMA) engines dédiés.

Ce parallélisme entre les transferts de données et les exécutions est exploité à l'aide de files d'exécutions (streams), elles permettent de soumettre des exécutions de kernels ou des transferts de mémoire de manière asynchrone. Dans le cadre d'une exécution multi-GPU, on utilise des streams différentes pour chacun des GPU. Les trois runtimes cités précédemment (CUDA, HIP et OpenCL) supportent ce formalisme.

Limitations des performances. Ces trois niveaux de parallélisme peuvent être difficiles à exploiter. Certains problèmes n'expriment pas suffisamment de parallélisme pour exploiter tous les cœurs en parallèle, d'autres ont des comportements qui rendront difficile une implémentation sans divergence des threads au sein d'un warp.

Des comportements classiques qui limitent le taux d'occupation du GPU sont mis en avant en sous-section 2.4.4 et 3.1.2. En particulier, des grilles trop petites peuvent s'exécuter trop rapidement pour masquer les surcoûts du runtime CUDA. D'autres grilles auront une découpe qui entraînera de nombreux threads inactifs dans chaque warp.

Pour limiter les problèmes d'occupation, deux solutions sont souvent utilisées : adapter la granularité des exécutions, c'est-à-dire exécuter des grilles avec plus de threads ou des kernels faisant plus de calculs. Enfin, il est aussi possible de soumettre plusieurs grilles de calculs à exécuter en même temps, en utilisant les concepts de parallélisme multi-streams proposés par les API et la capacité du matériel à gérer ce niveau de parallélisme.

Les problématiques liées à l'implémentation efficace des kernels nécessitent une implémentation minutieuse des kernels et une bonne connaissance du matériel ciblé. Lorsque l'on utilise des fonctions standards, il est préférable d'utiliser des bibliothèques où ces fonctions sont déjà implémentées par les spécialistes du domaine. Cet aspect de la programmation des co-processeurs est présenté à la sous-section 2.3.4.

2.3.4 Bibliothèque externes

Pour exploiter les coprocesseurs, une méthode simple et de bas niveau consiste à utiliser le runtime et les langages de programmation spécifiques correspondants. Par exemple, `CUDA` peut être utilisé pour les GPU Nvidia, `HIP` pour les GPU AMD, et `OpenCL` théoriquement pour tout type de coprocesseur. Cependant, ces langages sont limités car certains d'entre eux sont développés et supportés uniquement par un fabricant de matériel (comme `CUDA`), tandis que les langages ouverts ne permettent pas d'exploiter toutes les spécificités des coprocesseurs (par exemple, le support d'`OpenCL` par Nvidia est minimal).

De même, OpenCL est destiné à être compatible multiplateforme, mais les optimisations du code nécessaires pour obtenir de bonnes performances seront différentes en fonction de la cible : un CPU, un GPU ou un FPGA. Les codes dédiés aux co-processeurs sont donc peu portables et l'évolution rapide du matériel nécessite de faire évoluer le code en conséquence.

Pour pallier ces contraintes de développement, une seconde méthode consiste à utiliser des bibliothèques pour exploiter les co-processeurs. Cela permet d'exécuter certaines fonctions avec un code développé et optimisé par des équipes dédiées pour un ou plusieurs co-processeurs. Par exemple, les opérations d'algèbre linéaire sont standardisées avec le format BLAS, et plusieurs implémentations des fonctions BLAS existent en fonction du processeur ciblé : OpenBLAS ou Intel MKL pour les CPUs, cuBLAS pour les GPUs Nvidia, OCL-BLAS pour les co-processeurs supportant OpenCL, MAGMA, ...

L'utilisation de bibliothèques permet d'avoir des appels de fonctions standardisées et de déléguer le travail d'optimisation dédié au matériel à des équipes spécialisées. Ces bibliothèques ont été particu-

lièrement utilisées pour l'implémentation des algorithmes et benchmarks développés dans le cadre de ce travail, que ce soit pour les opérations en algèbre linéaire (essentiellement le produit de matrices) ou l'utilisation de transformée de Fourier nécessaire à l'implémentation de l'algorithme de formation de voies.

2.4 Formation de voies classique

La formation de voies est l'étape centrale des chaînes de traitement SONAR, car elle est utilisée dans la majorité des chaînes et que, de part son dimensionnement, elle a besoin d'une puissance de calcul importante en comparaison avec le reste de la chaîne. C'est pourquoi son optimisation va être un exemple récurrent dans la suite des travaux de la thèse. Cette section va détailler le principe de la formation de voies et expliciter l'algorithme utilisé comme base pour la suite de la thèse.

La formation de voies consiste à estimer un niveau d'énergie sonore en provenance d'une ou plusieurs directions données à partir d'un ensemble d'hydrophones.

2.4.1 Intuition de la méthode

Pour montrer l'intuition de ce traitement, nous allons présenter un exemple simplifié dans la figure 2.4a. Nous allons considérer une antenne de 4 hydrophones disposés en ligne : h_0, h_1, h_2 et h_3. On suppose que deux bruiteurs placés à grande distance émettent des signaux sinusoidaux, les fronts d'ondes sont représentés en vert et en bleu. Chacun des hydrophones va recevoir un signal qui sera la superposition du signal vert et du signal bleu, ainsi que d'un bruit, représenté comme un bruit blanc et indépendant pour chaque hydrophone.

En bas de la figure, l'antenne (en noir) est représentée avec les deux fronts d'onde. Les quatre premiers graphiques représentent le signal reçu par chacun des hydrophones, tandis que le graphique du bas représente la somme des signaux reçus. Le signal reçu est en noir, tandis que les composantes verte, bleue et du bruit sont représentées dans leurs couleurs respectives.

Dans cet exemple (2.4a), la somme des signaux fait ressortir la fréquence du signal vert, car le front d'onde arrive de manière synchrone sur tous les hydrophones. Le signal bleu arrive avec un retard sur chacun des hydrophones, ce qui crée des interférences destructives lors de la sommation des signaux. Dans le cas du bruit, il ne s'amplifie pas car il est indépendant d'un hydrophone à l'autre.

Pour mettre en valeur le signal bleu, il faudrait aligner l'antenne avec ce signal. L'idée derrière la formation de voies est d'ajouter un retard temporel sur chacun des signaux hydrophoniques pour compenser l'écart de distance entre la source et l'hydrophone. On va donc créer une antenne "virtuelle" perpendiculaire à la direction observée.

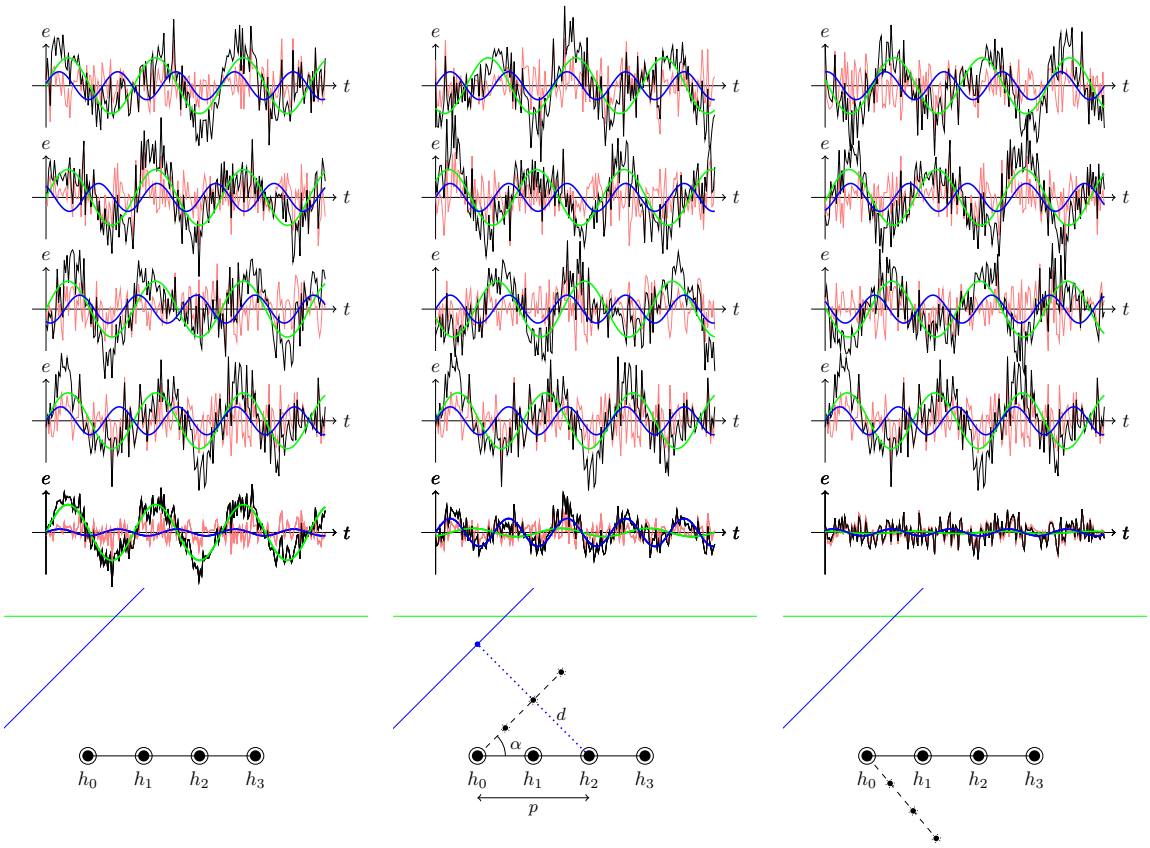
Les figures 2.4b et 2.4c montrent l'application d'un retard équivalent à tourner l'antenne de respectivement -45° et +50°. L'antenne virtuelle associée est représentée en pointillé. Tourner l'antenne de -45° permet de s'aligner avec la source bleue et de l'amplifier à la sommation des signaux. Tandis que la rotation de 50° ne s'aligne avec aucune source, on observe donc un signal plus faible.

La figure 2.5 représente le résultat obtenu en appliquant des retards équivalents à la rotation des antennes entre -90° et 90°, puis en intégrant la valeur absolue du signal obtenu. On considère que la valeur de cette intégrale est une estimation de l'énergie reçue depuis la direction ciblée. Pour ces exemples, le signal vert a une fréquence de 4 Hz et une amplitude de 0.5, le signal bleu a une fréquence de 6 Hz et une amplitude de 0.25, et le bruit est un bruit blanc réparti sur l'ensemble des fréquences avec une amplitude de 0.3.

On note que malgré un bruit d'amplitude supérieur au signal bleu, un pic correspondant à ce signal est clairement visible sur le graphique 2.5 à l'angle de -45°.

Le délai à appliquer à chaque hydrophone dépend de la célérité du son dans l'eau, de la position des capteurs et de la direction observée. Les valeurs sont déduites avec des règles trigonométriques de base. L'objectif est que, pour estimer le signal émis en un point donné, on applique des retards pour former une antenne virtuelle où le front d'onde atteint tous les hydrophones au même instant.

Dans l'exemple de la figure 2.4b, le front d'onde doit parcourir une distance d entre l'instant où il passe l'antenne virtuelle et celui où il passe l'hydrophone h_2. Si l'on définit c comme la célérité du son



(a) 0° (b) -45° (c) 50°

FIGURE 2.4 – Retard des hydrophones en fonction de l'angle observé

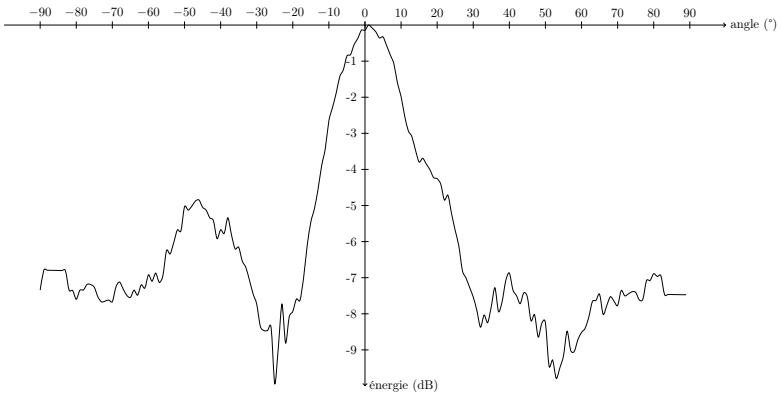


FIGURE 2.5 – Énergie estimée en fonction de l'angle observé

dans l'eau, le signal émis par la source bleue sera donc reçu avec un délai $t = \frac{d}{c}$ par l'hydrophone h_2 par rapport aux hydrophones de l'antenne virtuelle. Dans le cas de notre antenne, on peut simplement calculer d avec des règles trigonométriques de base : $d = p \sin \alpha$. Dans ce cas, le signal forme des interférences constructives. Cette approximation fonctionne car, si l'émetteur est suffisamment éloigné de l'antenne, on peut approximer que le front d'onde touchant l'antenne est plan localement. L'étude de bruiteurs proches de l'antenne nécessiterait un calcul de retard différent.

2.4.2 Formation de voies fréquentielle

Dans la sous-section précédente 2.4.1, le principe de la formation de voies a été présenté sous une forme dite temporelle. Cette méthode présente des défauts pour le traitement numérique : afin de pouvoir sommer deux hydrophones, il est nécessaire que les échantillons coïncident après l'ajout du délai. Ce n'est pas toujours le cas, en fonction des directions observées et de la position des hydrophones. En pratique, avec cette forme temporelle, un pré-traitement sur-échantillonne les signaux et des interpolations sont effectuées pour obtenir des résultats satisfaisants.

Une autre méthode pour effectuer la formation de voies est de travailler dans le domaine fréquentiel. En appliquant une transformée de Fourier au signal des hydrophones, on obtient un signal complexe représentant une décomposition du signal sur une base de fonctions sinusoïdales. Le passage du domaine temporel au domaine fréquentiel d'un hydrophone permet d'obtenir un vecteur où chaque valeur complexe représente l'amplitude du signal et son déphasage pour une fréquence donnée.

Appliquer un délai à un signal temporel revient à déphaser le signal dans le domaine fréquentiel, le déphasage dépend du délai et de la fréquence. Il peut être effectué en multipliant la valeur complexe du signal par un coefficient complexe de déphasage.

Pour traiter une antenne de M hydrophones, chaque hydrophone m est représenté par un vecteur $\mathbf{h}_m$ où $\mathbf{h}_m(t)$ est l'amplitude du signal reçu par l'hydrophone m à l'instant t. On construit les vecteurs $\mathbf{x}_m$ en appliquant une transformée de Fourier à $\mathbf{h}_m$:

$$\mathbf{x}_i = \mathcal{F}(\mathbf{h}_i)$$

Pour calculer une voie $\mathbf{y}_\theta$, on applique les coefficients de déphasage $\mathbf{d}_{i,\theta}$ à chacun des vecteurs $\mathbf{x}_i$:

$$\mathbf{y}_\theta = \sum_{i=0}^{i=M-1} \mathbf{x}_i \odot \mathbf{d}_{i,\theta}$$

Les tailles des vecteurs $\mathbf{d}_{i,\theta}$ et $\mathbf{x}_i$ dépendent du nombre de fréquences que l'on veut observer. Dans le cas où l'on veut observer les fréquences de manière séparée, $\mathbf{y}_\theta$ est un vecteur et le produit $\odot$ est un produit point à point des vecteurs. Sinon on peut directement appliquer un produit scalaire.

Les fréquences sont généralement séparées en résultat d'une formation de voies, car les traitements suivants peuvent tirer parti de cette information.

2.4.3 Cas particulier des antennes rectangulaires

Pour les travaux de la thèse, on se limite au cas où les hydrophones des antennes sont placés sur une grille régulière 2D. Les voies formées vont aussi être réparties sur une grille régulière dans le plan *site*, *azimut*.

On définit les grandeurs dimensionnantes suivantes pour le problème :

— $n_{hydro} = n_{ligne} \times n_{colonne}$: le nombre d'hydrophone en fonction des dimensions de l'antenne
— $n_{voies} = n_{site} \times n_{gisement}$: le nombre de voies observées
— n_{ech} : le nombre d'échantillons temporels traités à chaque itérations
— n_{frq} : le nombre de fréquences observées
— $n_{spectre}$: le nombre d'itérations consécutives fusionnées

La structure régulière des hydrophones et des voies permet de décomposer les coefficients de déphasage et de trouver des facteurs communs, par exemple pour les hydrophones d'une même ligne ou d'une même colonne.

Le calcul se fait alors en 4 étapes clefs :

— Analyse : passage du domaine temporel à fréquentiel avec des FFT.
— Calculs des pseudo-gisement : application d'une matrice de coefficient de déphasage $\mathbf{D}_{gisement}$ avec une multiplication de matrices ($GEMM$).
— Calculs des voies : application d'une matrice de coefficient de déphasage $\mathbf{D}_{site}$ avec un $GEMM$.
— Stabilisation et calcul de l'énergie : Interpolation du résultat des différents spectres pour compenser les mouvements de l'antenne et intégration.

```
1  func formation_de_voie()
2  {
3
4      for (i_spectre = 0 ... n_spectre)
5      {
6          H[i_spectre] = load_data()
7          for(i_hydro = 0 ... n_hydro)
8          {
9              X[i_spectre][i_hydro] = fft( H[i_spectre][i_hydro] )
10         }
11     }
12
13     X = selectSubarray( X );
14
15     D_gisement = place( D_gisement_0 )
16     D_site = place( D_site_0 )
17
18     for(i_grp = 0 ... n_grp)
19     {
20
21         D_frq = place( D_gisement )
22
23         for(i_frq = 0 ... n_frq)
24         {
25             for(i_spectre = 0 ... n_spectre)
26             {
27                 P_gisement[i_spectre] = gemm( X[i_grp][i_frq][i_spectre], D_frq )
28             }
29
30             for(i_gisement = 0 ... n_gisement)
31             {
32                 Y[i_gisement] = gemm( P_gisement[i_gisement], D_site )
33             }
34
35             D_frq = ponderate( D_frq, D_frq_delta)
36         }
37
38         D_gisement = ponderate( D_gisement, D_gisement_delta )
39         D_site = ponderate( D_site, D_site_delta )
40
41     }
42
43     Y = interpolation_norme( Y )
44 }
```

Code 2.1 – Pseudo-code de la formation de voies

Le code 2.1 est le pseudo-code de référence utilisé pour la suite de la thèse. Les opérations de réordonnancement (transpositions) de données ont été retirées pour simplifier le code.

On retrouve les quatre étapes : analyse ligne 9, calculs des pseudo-gisements ligne 27, calculs des voies lignes 32, et l'interpolation ligne 43. Les coefficients de déphasages $\mathbf{D}_{frq}$ et $\mathbf{D}_{site}$ sont calculés à la volée à l'aide de matrices pré-calculées : $\mathbf{D}_{gisement_0}$, $\mathbf{D}_{site_0}$ et $\mathbf{D}_{frq\_delta}$.

2.4.4 Optimisation manuelle

Une version Matlab équivalente au pseudo-code précédent a été fournie par les équipes de numériciens. Le premier travail de la thèse a consisté à porter ce code Matlab en une implémentation C++/CUDA pour une exécution sur GPU. Et d'analyser les performances du code pour trouver des optimisations permettant d'exploiter au mieux la cible.

Pour cette implémentation, nous avons essayé de limiter autant que possible l'utilisation de kernels CUDA faits à la main, afin de nous concentrer sur l'utilisation des bibliothèques cuBLAS et cuFFT. L'objectif est de faciliter la maitrise du code pour un développeur non spécialiste des GPUs et de profiter des optimisations réalisées par les constructeurs dans ces différentes bibliothèques.

Analyse des performances. Les performances sont observées à l'aide des outils de développement Nvidia tels que `nsight-system` et `nsight-compute`. On procède en 3 phases : localiser les sections qui prennent le plus de temps dans la timeline avec `nsight-system`, estimer les performances théoriques attendues (par exemple, une multiplication de matrices effectue n^3 opérations), profiler les kernels concernés avec `nsight-compute` et essayer de comprendre les écarts avec les performances théoriques.

La phase d'analyse (ligne 9) consiste en des calculs de FFT 1D. Elle est théoriquement bornée par la bande passante de la mémoire GPU. Les mesures montrent une sous-utilisation du GPU car les FFT sont trop petites pour masquer les surcoûts du runtime CUDA et pour saturer le GPU. Ces problématiques sont détaillées en sous-section 3.1.2.

Les opérations de multiplication de matrices (lignes 27 et 32) sont théoriquement limitées par la puissance de calcul. En pratique, les dimensions des antennes sont trop petites pour saturer le GPU, tout comme pour les FFT. De plus, la matrice $\mathbf{X}$ est dimensionnée en fonction de la taille de l'antenne, dont la grille peut avoir une hauteur de l'ordre de la dizaine d'hydrophones. Dans ce cas, la génération des pseudo-gisements (27) est moins performante car les dimensions du problème font que la découpe par la bibliothèque cuBLAS ne permet pas d'exploiter efficacement le GPU.

La phase d'interpolation a été implémentée avec un kernel personnalisé car les bibliothèques ne proposaient pas la norme désirée. Il est théoriquement limitée par la bande passante du GPU et les performances mesurées sont proches des performances théoriques attendues.

Les fonctions de pondérations sont implémentées avec un kernel personnalisé, qui est théoriquement limité par la bande passante du GPU. Les mesures montrent que la taille des opérations ne permet pas d'amortir les surcoûts du runtime CUDA.

Optimisations proposées : Les analyses évoquées au paragraphe précédent mettent en avant la difficulté à exploiter efficacement le GPU. Ces limitations proviennent de la faible granularité des opérations soumises au GPU.

Nous avons proposé trois types de modification du code pour réduire le temps de traitement de cette formation de voies.

Premièrement, **(A)** les coefficients de déphasage, mis-à-jours aux lignes 35, 38 et 39, peuvent être pré-calculés, car ils ne dépendent pas des données hydrophoniques en entrée du problème. Ils sont alors réutilisés à chaque itération de la formation de voies.

La deuxième modification **(B)** consiste à exécuter des blocs de fonctions, par exemple pour exécuter l'ensemble des FFT en un seul appel au runtime CUDA. Cette modification est appliquée aux FFT et aux multiplications de matrice, elle nécessite de casser les nids de boucles.

La dernière modification **(C)** modifie la fonction de calcul des pseudo-gisements (ligne 27). On peut observer que pour chaque itération $i_{spectre}$, la matrice $\mathbf{X}[i_{grp}][i_{frq}][i_{spectre}]$ de dimension (n_{lignes}, $n_{colonnes}$) est multipliée par la même matrice $\mathbf{D}_{frq}$ pour générer la matrice $\mathbf{P}_{gisement}[i_{spectre}]$ de dimension ($n_{gisement}$, n_{lignes}). Si l'on modifie le positionnement des données de $\mathbf{X}[i_{grp}][i_{frq}]$ pour obtenir une matrice de dimension ($n_{lignes} \times n_{spectres}$, $n_{colonnes}$), on peut générer la matrice $\mathbf{P}_{gisement}$ de dimension ($n_{gisement}$, $n_{lignes} \times n_{spectres}$) en une seule multiplication de matrice. Les principaux avantages sont : augmenter la taille du problème, ce qui permet une découpe plus efficace par la fonction cuBLAS, réduire le nombre d'appels au runtime CUDA et avoir une meilleure réutilisation des données de $\mathbf{D}_{frq}$ lorsqu'elles sont en cache du GPU.

Ces modifications nécessitent d'ordonner les données de façon spécifique pour que le format des matrices puisse être compatible avec les fonctions des bibliothèques. Pour des raisons de performances, il est préférable de ne pas avoir besoin d'appliquer des opérations de transposition entre les différentes étapes. Un travail pour choisir un placement des données qui permet d'éviter certaines transpositions a été effectué. En particulier une disposition naive des données implique plusieurs transpositions, entre l'analyse et le calcul des pseudo-gisement et entre le calcul des pseudo-gisement et le calcul des voies et après le calcul des voies.

En plaçant les données de manière entrelacée en sortie de l'étape d'analyse et en utilisant les bons paramètres pour les calculs des pseudo-gisement et des voies, nous pouvons éviter les deux dernières transpositions. Du fait des faibles tailles des matrices, les temps d'exécution des transpositions ne sont pas négligeables par rapport aux multiplications de matrices, les supprimer permet de réduire le temps d'exécution total de la chaîne d'environ 10% dans nos configurations.

Ces différentes optimisations ont permis une réduction du temps d'exécution du code d'un facteur 10, mais entraînent une augmentation des allocations mémoires importante par rapport au code initial. Les mesures sont détaillées en section 3.5.

2.5 Travaux connexes

Les algorithmes de traitement SONAR ont été étudiés dans les années 1990 et 2000 [35], [43], [59]. Une méthode commune pour obtenir des performances maîtrisées en termes de latence pour le traitement SONAR est d'implémenter les traitements sur cible FPGA [33], [62].

Plus récemment, des travaux pour déporter une partie des calculs sur GPU ont été effectués. Des premiers résultats ont montré l'intérêt d'effectuer des formations de voies sur GPU [47]. Une formation de voies GPU en traitement actif a été implémentée en 2014 [13] avec des gains de performances par rapport à une exécution CPU séquentielle, néanmoins le taux d'utilisation du bus FP32 du GPU était de l'ordre de 5%. Un ordre de grandeur similaire est mis en avant récemment par [60], pour une implémentation d'une formation de voies dans le domaine médical qui attribue ces limitations à une sous-optimisation pour conserver un code flexible. Des travaux pour le traitement des images issues du SONAR actifs existent [6], [32], ces étapes de la chaîne sont constituées d'algorithmes de traitement d'images qui se portent efficacement sur GPU. De même, les avancées récentes sur les technologies à base de réseaux de neurones ont conduit à des travaux sur la formation de voies à base de réseaux de neurones [57], [67].

Les travaux effectués au cours de cette thèse proposent d'approfondir l'exploration des traitements SONAR sur GPU, en particulier de la phase de formation de voies. Nous proposons de conserver l'approche initiée par [47] consistant à utiliser en priorité des bibliothèques externes (par exemple cuBLAS, cuFFT) et nous proposons des modifications automatiques de ces appels de fonctions pour éviter les limitations de performances mises en avant par [13] et [60], tout en partant d'un code initial faiblement optimisé. Les travaux de [47] ont conduit à un speedup de 11 à 15 entre une exécution sur quatre cœurs CPU et une exécution sur un GPU. D'un autre côté, les travaux de [60] ont conduit à un speedup de 12. Nos travaux ont également conduit à l'implémentation de formations de voies exécutables sur des cibles multi-GPU. Nous obtenons des speedups de 2 entre 64 cœurs et 1 GPU et jusqu'à 4,5 en utilisant 8 GPU. La comparaison avec ces travaux reste hasardeuse, car des différences persistent au niveau des dimensions des problèmes traités et du matériel à disposition.

2.6 Bilan

Dans ce chapitre, nous avons introduit le contexte applicatif des travaux effectués durant la thèse. Nous avons présenté l'algorithme de formation de voies qui représente la grande majorité du poids en calcul d'une chaîne de traitement SONAR telle que définie à Thales. La puissance de calcul nécessaire pour assurer des contraintes (même "molles") de temps réel nous incite à nous orienter vers les architectures de type GPGPU que nous avons présentées. Elles représentent à l'heure actuelle un excellent compromis en termes de puissance dans le cas d'une utilisation sous contrainte énergétique.

Le développement de tels algorithmes sur des architectures GPUs nécessite une expertise que les

numériciens spécialistes des schémas de calcul n'ont pas. Pour ne pas devoir faire ces optimisations manuellement pour chaque algorithme de formation de voies, nous nous sommes orientés vers la conception d'un outil permettant d'appliquer automatiquement les optimisations **(A)** et **(B)**, ainsi que les choix de placement, pour réduire les mouvements de données. L'optimisation **(C)** n'a pas été traitée automatiquement, car elle est fortement liée à une opération spécifique et peu générique. En effet, l'optimisation **(C)** découle des propriétés de la multiplication de matrices, une généralisation à d'autres fonctions serait compliquée.

L'objectif de l'outil est de fournir un code fonctionnel, avec des propositions d'optimisation et de placement mémoire avant une implémentation finale.

Le chapitre suivant 3 présente cet outil automatisant l'optimisation de codes issus des équipes de numériciens. Ces optimisations présentent des limitations, telles qu'une allocation importante de mémoire sur le GPU et une exécution limitée à un seul GPU à la fois. Ces limitations seront levées par la suite, avec d'une part l'introduction d'un modèle à base de tâche moldable au chapitre 4 et d'autre part l'étude d'un support exécutif adapté au chapitre 5.

Chapitre 3

Détection statique de code parallélisable

3.1 Introduction

Nous traitons ici du portage des algorithmes SONAR de Thales sur des cibles GPUs. Historiquement, l'implémentation de ces algorithmes de traitement SONAR a été réalisée pour des cibles CPU et multi-CPU. Le portage de ces codes sur des architectures basées sur des co-processeurs GPGPU soulève des questions sur les gains de performance et la facilité d'implémentation sur ces cibles spécifiques par les équipes de développement déjà en place.

Ce chapitre présente des modifications d'un code d'appels de kernels CUDA qui permettent d'augmenter l'occupation du GPU, ainsi qu'un outil permettant d'appliquer ces modifications de manière statique à l'étape de pré-processing de la compilation. Cette section expose les motivations d'utiliser un tel outil dans l'environnement de développement de Thales. La section 3.2 introduit la vue d'ensemble de l'outil, la section 3.3 propose des optimisations et leur implémentation dans l'outil, ainsi que leurs limitations. La section 3.4 détaille l'implémentation de l'outil, et enfin, la section 3.5 porte sur les mesures des performances du code généré par cet outil.

3.1.1 Organisation

L'organisation classique du développement d'une chaîne SONAR est représentée en figure 3.1, elle peut être décomposée en trois parties : premièrement des équipes de spécialistes du traitement du signal font un travail en amont pour définir les algorithmes à utiliser. À cette étape, ils fournissent un code séquentiel avec un langage de haut niveau, par exemple en Matlab ou en C, numériquement satisfaisant qui servira de référence pour la suite du développement. La seconde phase consiste à la réimplémentation séquentielle ou parallèle de cet algorithme en C par des équipes de développement en ajoutant toutes les interactions avec le système, par exemple les communications réseau pour l'acquisition des données. Finalement, si les performances obtenues sont en deçà des exigences, un support de personnes spécialisées dans le domaine de l'optimisation matériel est apporté pour optimiser les parties sensibles du code.

Les premiers travaux en amont de la thèse ont consisté à effectuer cette dernière étape pour un algorithme de formation de voies classique dans le cadre d'une implémentation mono-gpu. Ces travaux nous ont montré que certaines optimisations pertinentes pour cette nouvelle cible pouvaient être laborieuses à mettre en place par des personnes non spécialisées dans l'utilisation des co-processeurs. En particulier, grouper certains appels de fonctions pour limiter les surcoûts du runtime CUDA comme évoqué dans le chapitre précédent 2 et définir les patterns mémoire des données pour éviter les ré-ordonnancements en cours de calculs.

Pour la suite, nous proposons une nouvelle organisation où l'on effectue une passe d'optimisation automatisée sur un code de référence annoté pour servir de base à l'implémentation de la version mono-GPU de l'algorithme. Cette organisation est représentée en figure 3.2.

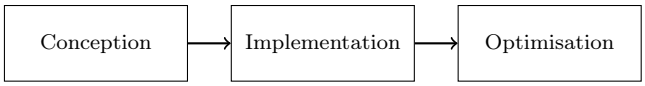


FIGURE 3.1 – Organisation classique du développement

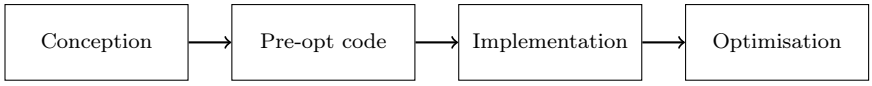


FIGURE 3.2 – Organisation proposée

3.1.2 Performances

Les performances d'un GPU dépendent fortement du travail qui lui est fourni, en particulier, on veut avoir le maximum de cœurs actifs en parallèle lors d'une exécution. Dans cette sous-section, nous allons mettre en avant deux causes de sous-utilisation du GPU rencontrées lors du portage sur GPU de la formation de voies présentée en section 2.4.4.

Surcoût du runtime CUDA et saturation du bus mémoire. Ces différentes étapes sont généralement masquées lorsque l'on exécute plusieurs kernels. Les appels au runtime CUDA étant effectués de manière asynchrone via les streams, chacune des couches peut préparer les kernels suivants pendant l'exécution du premier kernel. Néanmoins, dans le cas où les kernels sont trop rapides à exécuter, ces temps de préparation peuvent impacter les performances car le GPU sera inactif en attendant que les kernels soient prêts à être exécutés.

L'exemple suivant est l'exécution d'un kernel qui pondère un vecteur A par un vecteur B, on applique donc une multiplication point-à-point pour chaque élément. Le kernel utilisé pour effectuer ce calcul est basique : l'appel au kernel va calculer un 'batch' de n éléments, la grille comportera n threads calculant chacun 1 élément du vecteur.

Pour mettre en avant l'impact de la taille des kernels sur les performances, on va traiter l'ensemble des éléments du vecteur avec plusieurs appels successifs au kernel. En fonction de la taille du 'batch' n traité à chaque appel du kernel on aura un nombre d'appel au runtime différent pour traiter l'ensemble du vecteur.

Plusieurs mesures sont effectuées : le temps d'exécution des kernels sur le GPU [a] est la somme des temps d'exécution des kernels mesuré par l'outil de monitoring Nsight-System de Nvidia. Le temps pour soumettre les kernels à la stream [b] représente le temps passé dans la fonction `cudaLaunchKernel`. Le temps passé dans la fonction `cudaDeviceSynchronize` [c] qui est équivalent au temps nécessaire pour que tous les kernels soient terminés après l'exécution du dernier appel à `cudaLaunchKernel`. Le temps total de l'exécution est la somme des temps [b] + [c] .

Le kernel CUDA utilisé est implémenté de façon à ce que chaque thread calcule un seul élément du vecteur, les threads d'un même warp travaillent sur des éléments contigus en mémoire. Lors des appels, la grille CUDA est unidimensionnelle, les blocs CUDA sont de taille 256, et la taille des 'batchs' est un multiple de 80×256 pour que le nombre de blocs soit un multiple du nombre de `streaming-multiprocesseurs` du GPU Nvidia V100 utilisé. La taille du vecteur à traiter est de 670 millions d'éléments, elle est suffisamment grande pour éviter toute réutilisation des caches.

Les mesures ont été effectuées sur un nœud Gemini de Grid5000 [5], l'exécution utilise un GPU Nvidia V100. Les temps d'exécution ont été récupérés avec l'outil `nsys` de Nvidia [a] et avec la fonction `clock_gettime` [b] et [c], les mesures sont précédées d'exécutions "de chauffe" pour ne pas mesurer les surcoûts d'initialisation du runtime et de compilation des kernels. Les valeurs du tableau sont des moyennes sur 100 exécutions. Les résultats sont fournis dans le tableau 3.1.

On peut observer sur les 'batchs' de petite taille (20k et 40k) que l'appel à la fonction `cudaLaunchKernel` [b] est entre 5 et 10% plus long que le temps d'exécution des kernels sur le GPU [a]. Augmenter la taille des 'batchs' permet d'une part d'augmenter le temps d'exécution individuel des kernels sans augmenter le temps d'appel de la fonction `cudaLaunchKernel`. D'autre part, on observe aussi une augmentation de l'efficacité des kernels, car l'augmentation du nombre de blocs exécutés sur un `streaming-multiprocesseur`

Taille des batchs	20k	40k	80k	160k	320K	650k	1.3M	2.6M
kernel_vmul (ms) [a]	115	62	38	24	18	14	12	11
cudaLaunchKernel (ms) [b]	125	65	36	19	10	4	2	2
cudaDeviceSynchronize (ms) [c]	4	4	5	7	10	11	10	10
kernel_vmul [moyen/element] (ps/element) [d]	175	95	58	37	27	21	18	16
cudaLaunchKernel [moyen] (µs) [e]	3.8	3.9	4.3	4.6	4.6	4.1	4.7	6.2

TABLE 3.1 – Multiplication point à point de vecteurs de tailles 670M

permet de masquer les mouvements de données initiés par un bloc avec les opérations du bloc précédent.

De plus, on observe que le temps de synchronisation [c], c'est-à-dire le temps entre la soumission du dernier kernel et la fin de l'exécution, augmente. Cela signifie que l'appel au runtime CUDA s'exécute plus rapidement que le kernel sur le GPU. La diminution du temps moyen de calcul par élément [d] (en picosecondes, ce temps est mesuré dans le code CPU) nous indique que traiter des plus gros batch est bénéfique du point de vue performance.

Ces mesures montrent que l'exécution d'un kernel de taille trop faible nuit aux performances obtenues. En revanche, les mesures ne permettent pas de savoir si les surcoûts sont majoritairement dus au runtime, au driver ou à la gestion des kernels dans le GPU. On note aussi une irrégularité dans le coût de la fonction `cudaLaunchKernel` qui n'est pas expliquée mais qui reste régulière entre les exécutions. Des mesures plus précises pourraient être intéressantes pour détailler ces points et établir un modèle de performance à faible grain.

Utilisation de la grille CUDA Lors de l'exécution d'une grille, si la dimension où la découpe de la grille ne coincide pas avec le problème à traiter, les threads d'un même warp peuvent diverger. Pour illustrer cela, nous allons prendre pour exemple la multiplication de matrices $C := A \times B$, l'implémentation utilisée par Nvidia est présentée dans le blogpost [2].

La découpe de la grille CUDA est basée sur la forme de la matrice C, chaque bloc CUDA va calculer les points d'une sous-matrice de C. Les blocs sont donc en 2D, chaque warps (32 threads) calcule une sous-matrice 16×8 du bloc et chaque thread calcule 4 éléments de cette sous-matrice.

Une divergence a lieu lorsque les dimensions de la matrice C ne sont pas des multiples de la taille 16×8 traitée par un warp. Dans ce cas, les warps qui traitent les bords de la matrice auront moins de valeurs à calculer et certains threads seront donc inactifs.

Un exemple pour mettre en avant ce phénomène est présenté en figure 3.3. Nous avons exécuté une série de multiplications de matrices avec la fonction `cublasSgemmStridedBatched` qui calcule un ensemble de multiplications de matrices $C_i = A_i \times B_i$ avec $i \in [\![0, 127]\!]$. Les matrices A, B et C sont respectivement de tailles : $m \times k$, $k \times n$ et $m \times n$, les valeurs $n = k = 256$ sont fixées tandis que m est variable dans l'intervalle $[\![1, 256]\!]$.

L'utilisation d'une fonction "batch" permet de fournir suffisamment de blocs au GPU pour masquer les surcoûts du runtime mis en avant précédemment. L'objectif est de mesurer le temps d'exécution de 128 appels à cette fonction (donc un total de 128×128 matrices calculées), les données sont déjà présentes sur le GPU.

La figure 3.3 présente les résultats obtenus en exécutant ce benchmark sur un serveur 'gemini' de la plateforme Grid5000 [5]. L'exécution se fait sur une carte Nvidia Tesla V100 avec $fp_{32} = 7.5\ TFP32/s$ de puissance théorique. L'axe des abscisses représente la dimension m des opérations, la courbe bleue représente le temps d'exécution mesuré en millisecondes, la courbe rouge représente le temps d'exécution théorique attendu $\frac{n_{ops}}{fp_{32}}$ avec $n_{ops} = m \times n \times k \times 128 \times 128$. La courbe hachée noire est la division du temps mesuré par le temps théorique, elle représente le taux d'utilisation des opérateurs FP32 du GPU lors du calcul, l'échelle est en % à droite de la figure.

Les performances mesurées sont clairement sous forme de paliers. Cela s'explique par une grille qui reste identique pour plusieurs tailles de m différentes, mais le nombre de threads inactifs diminue au fur et à mesure que m augmente. Nous pouvons aussi observer que les matrices avec un m inférieur à 96 peinent à atteindre 50% de la performance théorique de la carte.

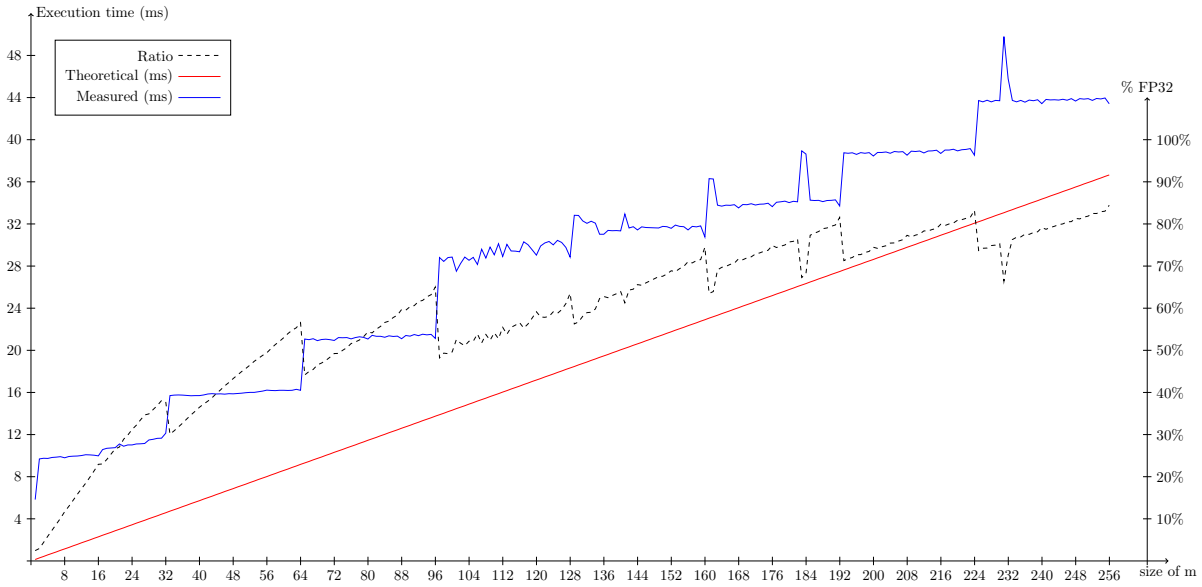


FIGURE 3.3 – Multiplications de matrices avec $n = k = 256$ et m variable sur Tesla V100

Conclusion sur ces deux exemples : Ces exemples ont été inspirés de problématiques de performances rencontrées lors des travaux d'optimisations qui ont initié la thèse. Ils montrent l'intérêt à regrouper plusieurs calculs ensemble, soit en utilisant des batchs comme dans le premier exemple, soit en modifiant la forme du problème dans le second. Cela permet d'exploiter plus efficacement le GPU à l'aide de bibliothèques externes (ici cuBLAS) et valide les travaux proposant des extensions d'API pour regrouper les appels à des kernels en algèbre linéaire [20]. Néanmoins, l'utilisation de ces API, découlant de contraintes d'exploitation efficace, n'est pas prise en compte par les équipes de « numériciens » dont l'objectif premier est la production d'un code valide numériquement et pour lesquelles l'optimisation des performances est déléguée aux équipes suivantes.

3.2 Un outil semi-automatique d'aide à l'optimisation

Les optimisations apportées au code de formation de voies ont majoritairement consisté à grouper des appels à des fonctions similaires entre elles et à gérer l'ordre des données en mémoire pour limiter les réordonnancements. Ces modifications étant suffisamment génériques pour s'appliquer à d'autres algorithmes de la chaîne de traitement SONAR, nous avons décidé de fournir un outil capable de mettre en œuvre automatiquement ces modifications.

Le principe général de l'outil est, à partir d'un code C annoté par un expert, de générer un graphe qui représente les soumissions des kernels GPUs et les dépendances mémoires entre ces soumissions, d'appliquer des modifications automatiques à ce graphe et de générer un code compilable et exécutable à partir du code annoté et du graphe. En particulier, l'outil doit générer les allocations mémoires et la fonction qui soumet toutes les tâches au GPU. De plus, les modifications apportées au graphe doivent induire que le code généré par le nouveau graphe donne un résultat numérique identique à celui généré par le graphe d'origine.

Un schéma décrivant le fonctionnement de l'outil est disponible en figure 3.4, les cases `input.c`, `output.c` et `output.dot` représentent des fichiers d'entrée et de sortie du programme, les cases `parser`, `optimizer`, `C generator`, `dot generator` représentent les différentes fonctions du code et la case `graph` est la représentation abstraite du code, en interne du programme.

En fonction des arguments en entrée du programme, différentes modifications seront appliquées au `graph`, on pourra donc générer pour un même code d'entrée, différents `graph` et différentes sorties associées.

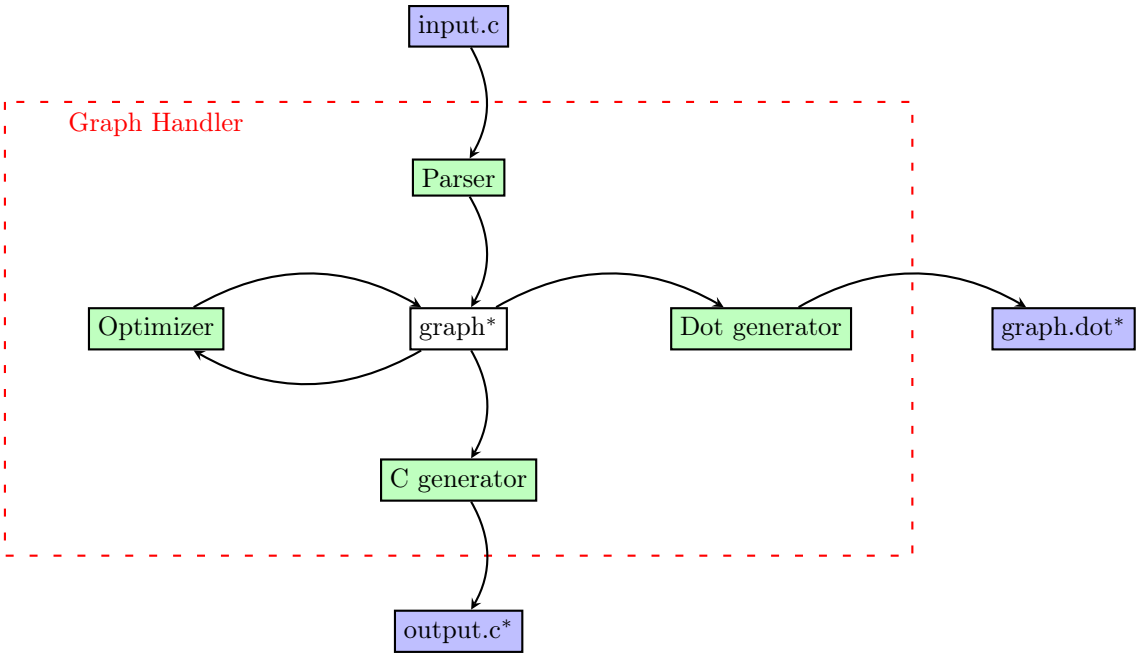


FIGURE 3.4 – Vue d'ensemble de l'outil

3.2.1 Annotation du code

Nous avons choisi d'utiliser un code C annoté pour appliquer les optimisations. Les optimisations se basent sur le fait que les fonctions accèdent à des données stockées sous forme de tableaux multi-dimensionnels, les annotations permettent de déclarer explicitement ces dimensions et les tableaux associés. La mise en œuvre d'analyse de code pour inférer ces informations nous semblait plus complexe à implémenter, là où les numériciens sont à l'aise avec les notions de dimensions car elles se mappent avec les outils mathématiques utilisés. De plus, les annotations permettent de décrire les liens entre les arguments et les dimensions auxquelles une fonction accède, il nous serait impossible d'inférer ces informations pour des fonctions de bibliothèques tierces.

Le code fourni en entrée via le fichier `input.c` doit être annoté par l'utilisateur, avec pour objectif de générer le graphe à partir du code. Cette sous-section décrit les différents pragma introduits. On utilise des pragma pour l'annotation qui ont une forme qui rappelle celle utilisée par OpenMP :

```
#pragma PS directive [clauses...]
```

Déclaration des tableaux. Une première directive `data` permet de définir des variables numériques qui vont représenter les dimensions du problème. Elles sont liées aux dimensions physiques du problème, par exemple le nombre de colonnes de l'antenne, ou son nombre de lignes. On utilise la clause `atomicDim([name...])` pour déclarer une ou plusieurs dimensions élémentaires. Par exemple :

```
#pragma PS data atomicDim(Row, Col)
```

déclare les dimensions `Row` et `Col` qui seront associées a des variables de taille n_{row} et n_{col} indiquant la taille.

De plus, certaines dimensions peuvent être déduites à partir de plusieurs autres dimensions. Par exemple, le nombre de capteurs d'une antenne rectangulaire est calculé en multipliant le nombre de lignes par le nombre de colonnes de l'antenne. Pour déclarer de telles dimensions, on utilise la clause `composedDim( name, [dimensions...] )` pour déclarer une dimension à partir de plusieurs autres dimensions. Par exemple :

```
#pragma PS data composedDim(Hydro, Row, Col)
```

signifie que la dimension `Hydro` est composée des dimensions `Row` et `Col`, sa taille est le produit $n_{hydro} = n_{row} \times n_{col}$.

À partir de ces dimensions, on va pouvoir déclarer des tableaux multi-dimensionnels avec la clause `array(type, name, [dimensions...])`. Ces dimensions et ces tableaux seront initialisés et alloués par une fonction générée automatiquement. L'exemple suivant montre la déclaration d'un tableau `ArrayHydro` de dimensions `Hydro,Freq` contenant des valeurs de type `float`.

```
#pragma PS data atomicDim(Row, Col, Freq)
#pragma PS data composedDim(Hydro, Row, Col)

#pragma PS array(float, ArrayHydro, Hydro, Freq)
```

Lors du code final, ce tableau sera lié à un pointeur sur un espace mémoire de taille $|Hydro| \times |Freq| \times$ sizeof(float).

Annotation des fonctions. Une deuxième étape consiste à annoter les fonctions utilisées pour les appels GPU. L'objectif est de savoir comment elles accèdent aux données.

Pour cela, on annote la déclaration de la fonction par la directive `function` afin de spécifier l'accès aux tableaux effectué par la fonction. Les clauses `input` et `output` permettent de définir les paramètres formels de la fonction, qui sont des tableaux, en précisant leurs modes d'accès en lecture ou en écriture. La clause `volatile` permet de préciser que les données générées par une fonction ne dépendent pas seulement des paramètres en entrée. Cela signifie qu'à chaque appel à cette fonction, les tableaux accédés en écriture auront des données différentes, et donc qu'il ne sera pas possible de pré-calculer les valeurs de ces tableaux. Cette clause est utilisée, par exemple, pour les fonctions qui chargent des données depuis le réseau.

Finalement, la directive `data` est également utilisée pour définir les schémas d'accès aux données. Pour cela, on utilise des clauses spécifiques `blas(array, dim0, dim1)`, `blasT(array, dim0, dim1)` et `fixed(array, [dim...])`. Elles indiquent respectivement que le tableau accédé peut être stocké sous un format 'BLAS', qui peut être transposé, avec un format imposé.

On appelle un format 'BLAS' le stockage d'un tableau à deux dimensions `dim0,dim1` où, pour chaque itération sur la dimension `dim0`, `dim1` éléments sont stockés de manière consécutive en mémoire. La distance entre les deux premiers éléments de deux itérations consécutives sur `dim0` est appelée *leading dimension* et est supérieure mais pas forcément égale à `dim1`. Ce formalisme permet d'utiliser des matrices où les lignes/colonnes ne sont pas placées de manière contigüe en mémoire. Lorsque l'on supporte la transposition, les données peuvent aussi être stockées de la même façon en inversant les rôles de `dim0` et `dim1`.

Ci-dessous, nous présentons un exemple d'annotation pour la fonction `cublasCgemm` 3.1 : on redéclare la fonction de la bibliothèque cuBLAS en ajoutant les annotations décrites ci-dessus.

```
1  #pragma PS data blasT(A, M, K) blasT(B, K, N) blas(C, M, N)
2  #pragma PS function output(C) input(A, B)
3  cublasStatus_t cublasCgemm(cublasHandle_t handle,
4                             cublasOperation_t transa, cublasOperation_t transb,
5                             int M, int N, int K,
6                             const cuComplex *alpha,
7                             const cuComplex *A, int lda,
8                             const cuComplex *B, int ldb,
9                             const cuComplex *beta,
10                            cuComplex *C, int ldc);
```

Code 3.1 – Annotation de la fonction cublasCgemm

Fonction génératrice du graphe. La dernière étape consiste à définir la fonction qui va être transformée en graphe afin d'être optimisée. On déclare cette fonction avec la directive `optimize` placée devant la définition de la fonction. La syntaxe utilisée dans la fonction est restreinte : les fonctions appelées doivent être uniquement des fonctions précédemment annotées et il ne doit pas y avoir de structures conditionnelles (`if`).

Les boucles sont effectuées de manière restreinte, on les limite à une itération sur une dimension précédemment déclarée avec les clauses `atomicDim` ou `composedDim`, la syntaxe est la suivante :

```
for(;dim_name;)
{
    <structured-block>
}
```

On introduit aussi des fonctions utilitaires : `ps_ld`, `ps_stride` et `ps_blas_op`, qui seront remplacées par les valeurs des leading dimensions, des strides et des opérations de transpositions à la génération du code en fonction des choix de placement de données effectués par le programme.

Par exemple, l'appel à la fonction `cublasCgemm` déclarée dans le code 3.1 sera le suivant :

```
#pragma PS data atomicDim(dimM, dimN, dimK)

#pragma PS array(cuComplex, A, dimM, dimK)
#pragma PS array(cuComplex, B, dimK, dimN)
#pragma PS array(cuComplex, C, dimM, dimN)

cublasCgemm( handle, ps_blas_op(A, dimM, dimK), ps_blas_op(B, dimK, dimN),
        dimN, dimM, dimK,
        &alpha, A, ps_ld(A, dimM, dimK),
                B, ps_ld(B, dimK, dimN),
        &beta,  C, ps_ld(C, dimM, dimN) );
```

Les fonctions `ps_ld` et `ps_blas_op` seront remplacées dans le fichier final afin de correspondre au placement des données en mémoire.

3.2.2 Représentation abstraite

La fonction à optimiser est représentée sous forme de deux graphes. Premièrement, un graphe hiérarchique proche de l'arbre de syntaxe abstrait (AST) est composé de différents types de nœuds : la racine est le corps de la fonction, les feuilles sont les appels de fonction et les nœuds intermédiaires sont les corps des différentes boucles. En plus de conserver la structure des nids de boucle, ce graphe conserve la dimension d'itération de chacune des boucles.

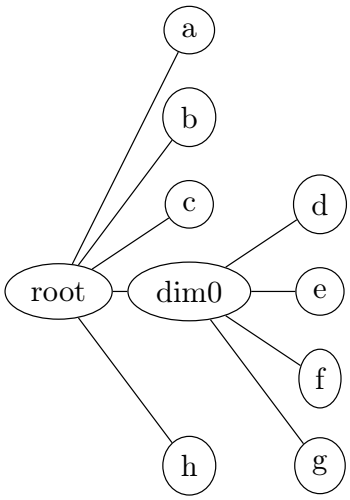
Un second graphe, le graphe des dépendances, représente les dépendances entre les appels de fonction. Il est donc composé des feuilles du graphe hiérarchique, les liens entre deux nœuds sont générés si une dépendance de données existe entre ces nœuds. Le graphe est orienté et acyclique. Les arcs du graphe conservent l'information des tableaux qui ont induit la dépendance entre les deux nœuds.

La figure 3.5 représente un pseudo-code 3.5a, le graphe hiérarchique 3.5b associé et le graphe des dépendances 3.5c. La représentation du graphe des dépendances permet aussi de représenter le graphe de hiérarchie en encapsulant les nœuds dans des rectangles représentant le corps des boucles et de la fonction. Les données soulignées dans le pseudo-code sont accédées en écriture.

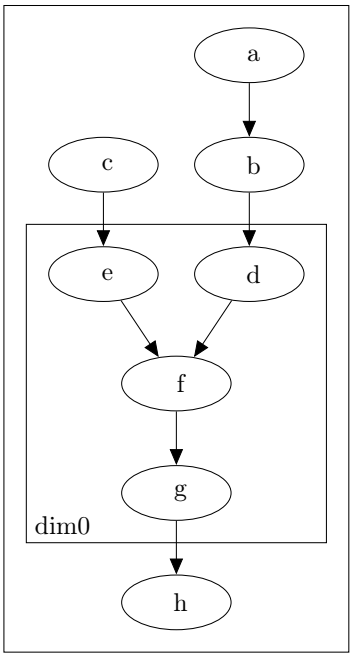
Le graphe de hiérarchie permet de connaître le nombre d'appels à la fonction et les données impactées, tandis que le graphe de dépendance permet de conserver l'ordre d'appel des fonctions et donc l'ordre de génération des appels dans le code. De plus, le graphe de dépendance induit les contraintes de placement des données en fonction des fonctions qui accèdent à un tableau.

```
#pragma PS optimize
void function()
{
     a(A);
     b(A,B);
     c(D);
     for(;dim0;)
     {
          d(B,E);
          e(B,F);
          f(E,F,G);
          g(G,H);
     }
     h(H);
}
```

(a) code



(b) Graphe hierachique



(c) Grape des dépendances

FIGURE 3.5 – Représentation du graphe et du code associé

3.3 Modifications des graphes

Les travaux d'optimisations manuelles ont permis d'identifier plusieurs problèmes de performances qui peuvent être couramment observés, cette section présentera chaque optimisation et une méthode pour l'appliquer au graphe des tâches. Elle terminera par un point sur les contraintes mémoires découlant de ces modifications.

3.3.1 Pré-calculs

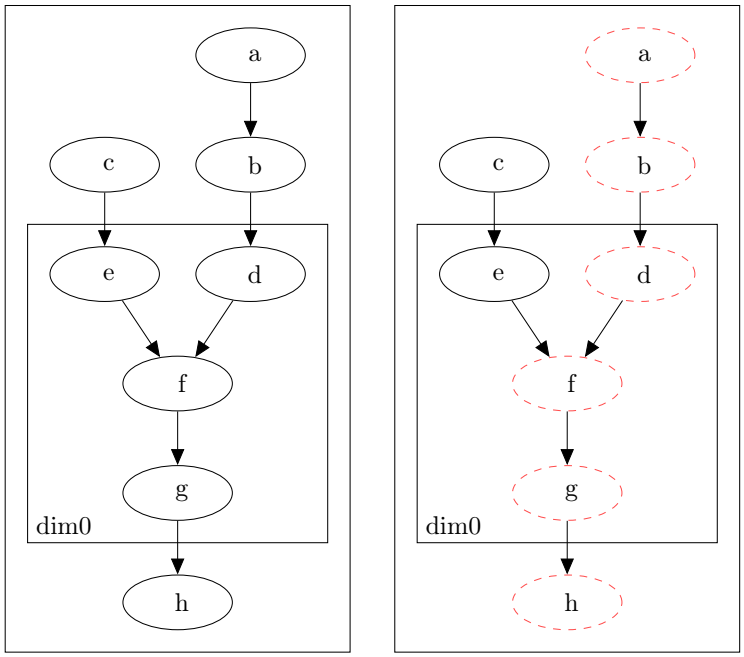
Une optimisation basique est d'éviter de recalculer plusieurs fois les mêmes données. Les chaînes de traitements de données étant des algorithmes itératifs qui exécutent un code identique à intervalle régulier, certains coefficients peuvent ne pas être recalculés à chaque itération. L'objectif est alors de générer un code qui détecte si ces données sont déjà présentes en mémoire pour éviter de les recalculer.

Pour détecter les fonctions concernées, une méthode basique est de séparer les nœuds du graphe en deux groupes : les nœuds 'volatiles' qui produisent des données potentiellement différentes à chaque appel de fonction, les nœuds 'pré-calculables' qui peuvent ne pas être ré-exécutés un nombre fini de fois. Un nœud est 'volatile' si la fonction exécutée est associée à la directive `volatile` présentée en section 3.2.1 ou s'il travaille sur des données produites par un nœud 'volatile'.

La détection de ces nœuds est faite en coloriant tous les nœuds du graphe associés à la directive `volatile` à l'initialisation et en parcourant récursivement les enfants des nœuds 'volatiles' pour les colorier à leur tour. On sépare donc l'ensemble des nœuds en temps linéaire en fonction de la taille du graphe. La figure 3.6 présente un exemple de coloriage du graphe dans le cas où la fonction a est volatile. On peut donc appliquer un algorithme de parcours en largeur ou en profondeur du graphe à partir des nœuds initiaux.

3.3.2 Batch d'appels de fonctions

Comme expliqué dans la section 3.1.2, l'exploitation efficace du GPU nécessite de soumettre des tâches d'un grain suffisamment élevé afin de saturer les cœurs de calcul et masquer les surcoûts du runtime



(a) Graphe initial (b) Graphe colorié

FIGURE 3.6 – Exemple de coloriage du graphe pour le pré-calcul

CUDA. Cependant, les algorithmes proposés par les équipes de traitements SONAR sont dépendants des dimensions physiques du problème, telles que la taille des antennes, le nombre de voies et de fréquences observées, etc. Il s'avère que les dimensions effectives de ces opérations sont alors trop faibles pour saturer un GPU.

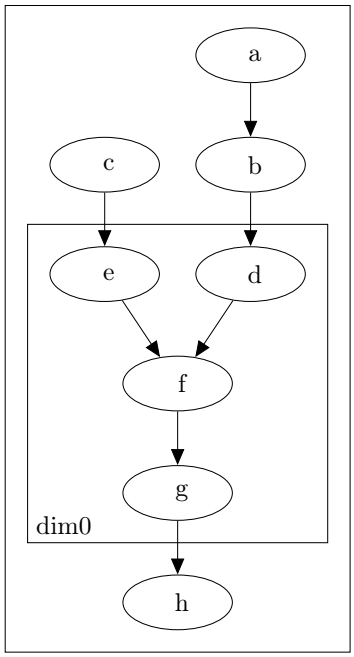
Une méthode proposée pour augmenter les taux d'utilisation du GPU est de regrouper les appels de fonctions par 'batch'. Les kernels cuBLAS ont des alternatives 'batch' qui permettent de soumettre et calculer plusieurs opérations identiques sur des données différentes en une seule soumission de tâche au GPU. La structure du code en nids de boucle fait que chaque nœud du graphe représente une fonction qui est appelée plusieurs fois en fonction du nombre d'itérations des boucles qui l'incluent. L'idée est donc de sortir le nœud de sa boucle et de remplacer l'appel de la fonction par son équivalent 'batch'.

L'exemple suivant montre la modification de code que l'on souhaite faire avec la fonction `cublasCgemm`, une version par 'batch' de cette fonction est `cublasCgemmBatch`. Le code de gauche est le code initial, tandis que le code de droite est le code que l'on souhaiterait obtenir en groupant les appels de `cublasCgemm`.

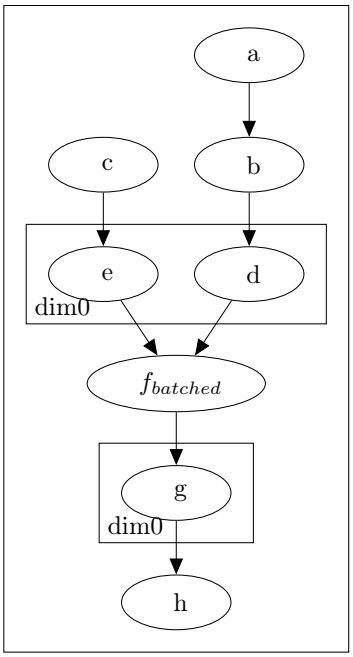
```
for(;dim0;)
{
  cublasCgemm( handle,
    ps_blas_op(A, dimM, dimK),
    ps_blas_op(B, dimK, dimN),
    dimN, dimM, dimK,
    &alpha, A, ps_ld(A, dimM, dimK),
    B, ps_ld(B, dimK, dimN),
    &beta,  C, ps_ld(C, dimM, dimN) );
}
```

```
{
  cublasCgemmStridedBatch( handle,
    ps_blas_op(A, dimM, dimK),
    ps_blas_op(B, dimK, dimN),
    dimN, dimM, dimK,
    &alpha, A, ps_ld(A, dimM, dimK),
    ps_stride(A, dimM, dimK),
    B, ps_ld(B, dimK, dimN),
    ps_stride(B, dimK, dimN),
    &beta,  C, ps_ld(C, dimM, dimN),
    ps_stride(C, dimM, dimN),
    dim0 );
}
```

Nous allons regrouper uniquement des fonctions provenant d'un même nœud dans notre graphe. Pour qu'un nœud soit "batché", la fonction associée doit être batchable : par exemple la fonction GEMM.



(a) Graphe initial



(b) Graphe des dépendances après le batch

```
#pragma PS optimize
void function()
{
     a(A);
     b(A,B);
     c(D);
     for(;dim0;)
     {
          d(B,E);
          e(B,F);
     }
     f_batch(E,F,G);
     for(;dim0;)
     {
          g(G,H);
     }
     h(H);
}
```

(c) Code après le batch

FIGURE 3.7 – Modification de graphe pour les batchs.

Pour suivre l'exemple proposé en figure 3.5, nous allons supposer que la fonction f a un équivalent 'batch' $f_{batched}$. La figure 3.7 présente les modifications appliquées au graphe et au code lors de la passe d'optimisation batch des fonctions. La sous-figure 3.7a représente le graphe initial, la sous-figure 3.7b représente le graphe après modification : on peut observer que la boucle sur dim_0 a été séparée en deux boucles pour extraire la fonction f qui a été remplacée par f_{batch}. La première boucle contient les prédécesseurs de f dans le graphe des dépendances, tandis que la seconde contient les successeurs.

Pour que la séparation soit valide, il faut que les buffers de données en entrée et en sortie de $f / f_{batched}$ soient adaptés pour contenir toutes les données produites dans le corps de la boucle. En d'autres termes, on doit agrandir certains buffers de travail. De même, si des dépendances sont présentes entre les nœuds de la première boucle créée et les nœuds de la seconde boucle créée, il faut aussi adapter la taille des buffers en interface. S'il existe des nœuds qui ne sont ni successeurs de f, ni prédécesseurs de f, il faudra les placer dans l'une des deux boucles. Un choix qui semble logique est de placer ces nœuds de façon à minimiser la taille des buffers en interface.

Dans notre exemple, le code étant passé de 3.5a à 3.7c, on remarque que les tableaux E, F et G peuvent nécessiter un redimensionnement.

Contraintes. Les modifications précédentes nécessitent potentiellement d'agrandir les buffers de travail. En effet, dans l'exemple précédent 3.7c, la fonction e produit les données F_i avec $i \in [\![0, |dim_0| - 1]\!]$. Dans le cas où la boucle n'est pas séparée 3.5a, les données générées sont directement réutilisées par la fonction f. Mais lorsque f_{batch} est sortie de la boucle, il faut conserver toutes les versions F_i générées par e. Si le tableau ne contient pas l'espace nécessaire pour contenir toutes ces données (dans le cas où dim_0 ne fait pas partie des dimensions déclarées dans le tableau), il faudra l'étendre pour que le code modifié soit valide.

De même, si un nœud peut être pré-calculé, il peut être intéressant d'agrandir les tableaux pour contenir toutes les données générées et ne pas avoir à recalculer plusieurs fois la même donnée.

L'algorithme va alors redimensionner les tableaux de manière à stocker toutes les données nécessaires. Les tableaux sont déclarés sous la forme d'un ensemble de dimensions, et la taille du tableau est le

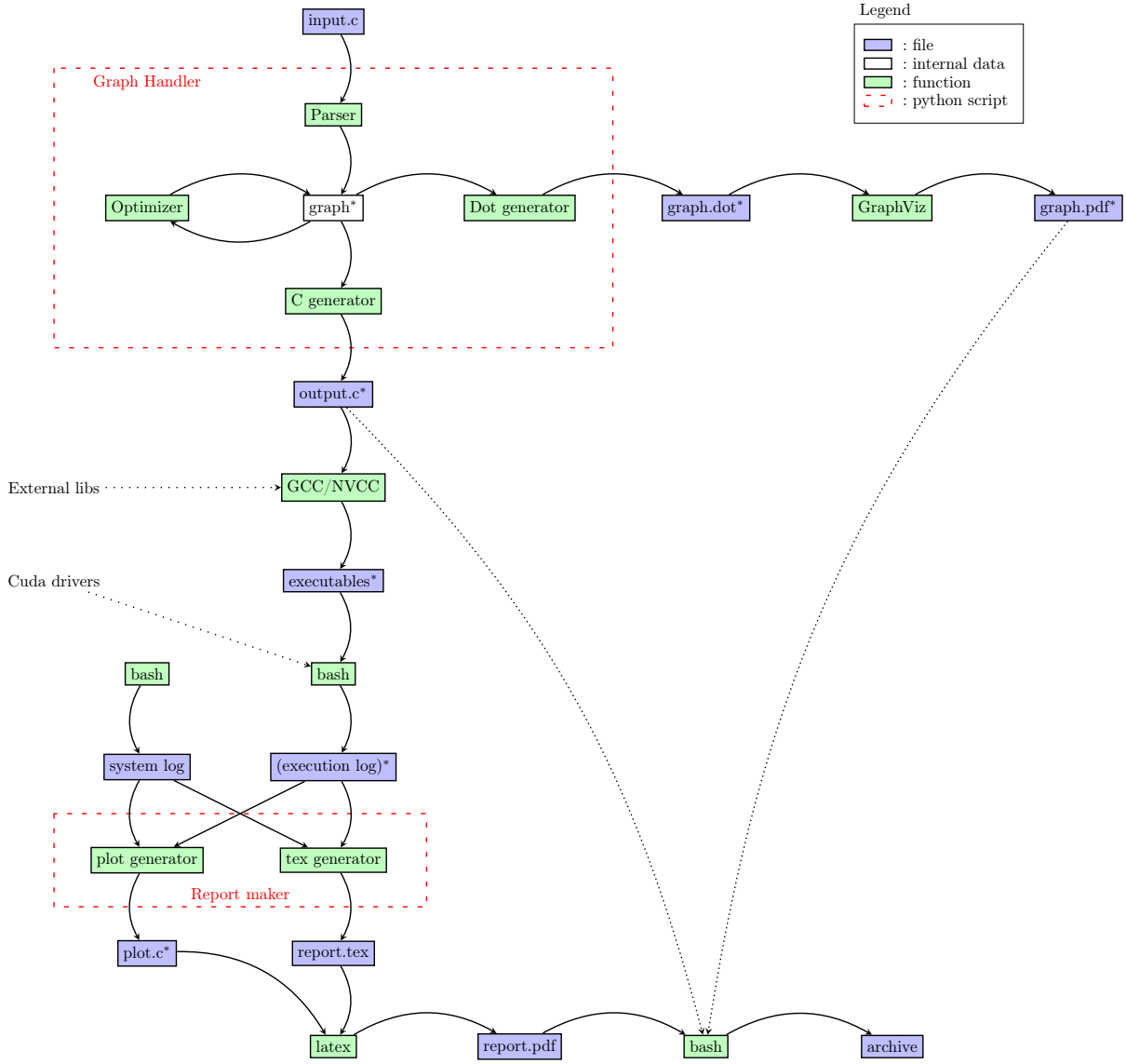


FIGURE 3.8 – Implémentation de la passe statique et des outils utilitaires

produit de ces dimensions. Si un nœud est précalculé, les tableaux en sortie se verront ajouter toutes les dimensions des boucles qui contiennent ce nœud et qui ne sont pas présentes dans l'ensemble des dimensions du tableau. Pour un nœud "batché", la dimension de la boucle séparée en deux sera ajoutée aux tableaux en entrée et en sortie du nœud.

3.4 Implémentation

Cette section présente l'implémentation de l'outil et des transformations de graphe décrites ci-dessus. Nous présentons aussi la chaîne de compilation et d'exécution qui permet de l'utiliser avec un rapport des performances. Un schéma complet est présenté en figure 3.8. La chaîne prend un fichier `input.c` en entrée et a différentes sorties : les fichiers de code générés par l'outil `output.c`, les graphes associés à ces fichiers `graph.pdf` et un rapport d'exécution de chacun de ces programmes sur une cible donnée `report.pdf`. La gestion des optimisations est implémentée par le module appelé `Graph Handler` présenté en section 3.4.1, 3.4.2 et 3.4.3. La suite de la chaîne permet d'exécuter le code généré sur une cible et de créer un

rapport d'exécution, elle est détaillée en section 3.4.4.

3.4.1 Création du graphe de tâches

Le script `GraphHandler.py` est un script Python qui accepte en entrée un fichier C annoté avec les pragmas présentés en section 3.2.1 et un ensemble d'optimisations à appliquer : les deux optimisations acceptées étant le pré-calcul (section 3.3.1) et le traitement par batch (section 3.3.2).

La première étape du script consiste à parser le fichier C en entrée. L'implémentation a été réalisée à partir de la bibliothèque pycparser. L'AST généré est ensuite visité afin de créer un contexte et un graphe. Le contexte comprend les dimensions et les tableaux déclarés dans le code ainsi que la structure du fichier : les inclusions, les fonctions, etc. Le graphe est généré à partir d'une fonction annotée par `#pragma PS optimize`. La partie hiérarchique du graphe est créée de manière simple en suivant la structure de l'AST, tandis que les liens de dépendance sont définis en fonction des tableaux auxquels les fonctions accèdent et des règles de dépendance classiques : Read after Write, Write after Read et Write after Write.

Les modifications sont appliquées sur le graphe généré par le parser, pour le pré-calcul, on applique la fonction présentée dans le code 3.2 qui va colorier les nœuds volatils : elle récupère les nœuds du graphe en ligne 2, filtre les nœuds volatils en ligne 3 et parcourt le graphe à partir des nœuds volatils avec un parcours en largeur pour appliquer la volatilité aux nœuds dépendants. Les fonctions pré-calculables sont celles qui ne sont pas coloriées.

```
1 def tag_volatile( graph ):
2     funcs = graph.get_functions_nodes()
3     roots = list( filter( NODES.Function_node.is_volatile, funcs ) )
4     f = lambda n : n.set_volatile( True )
5     search.bfs_functions( roots, f )
```

Code 3.2 – Coloriage des nœuds volatiles

Les tableaux en sortie des fonctions sont ensuite redimensionnés avec la fonction suivante 3.3, où pour chaque tableau produit (ligne 2), on ajoute les dimensions de tous les parents dans la hiérarchie de la fonction au tableau. La variable *dim* correspond à des dimensions atomiques que l'utilisateur aura déclaré avec les pragma définis en sous-section 3.2.1 et associées aux tableaux manipulés par les fonctions.

```
1 def maximize_function_output_dim( func ):
2     for ds in func.get_produced():
3         detected_dims = ds.get_dims()
4         for dim in func.get_parents_dims():
5             if dim != None and not dim in detected_dims:
6                 ds.get_array().add_dim( dim )
7                 detected_dims.add( dim )
```

Code 3.3 – Redimension des tableaux de sortie

La modification des 'batch' est effectuée dans un deuxième temps. Premièrement, les nœuds pré-calculables ne sont pas 'batch' car ils vont être exécutés seulement aux premières itérations et leur coût est considéré comme négligeable. Ensuite, le code vérifie pour chacun des nœuds restants s'ils sont éligibles à un batch en vérifiant : si la fonction est éligible au batch, si la fonction est contenue dans une boucle et si la boucle peut être divisée en deux.

Pour définir une fonction 'batchable', on doit connaître la liste de ses arguments et comment ils se transposent dans la fonction de remplacement, dans le cadre de ces travaux, nous avons ajouté les fonctions `cublas?gemm`, `cublas?gemmStridedBatch` et `cuFFT` directement dans l'outil (hardcodé) et l'ajout d'annotations pour permettre aux utilisateurs d'ajouter d'autres fonctions est envisagé.

Si la fonction peut être réalisée en mode batch, les tableaux d'entrée et de sortie du nœud sont redimensionnés, si nécessaire, par l'ajout de la dimension de la boucle découpée. La fonction de modification du graphe est présentée en 3.4, les lignes 4 à 9 définissent les nœuds des deux boucles générées, les lignes 11 à 26 génèrent les deux nouvelles boucles et changent les positions des nœuds, et les lignes 28 à 30 mettent à jour le nœud batch.

Le code est implémenté en Python, la fonction `set` crée un ensemble (liste sans duplication), la méthode `update` ajoute des éléments à l'ensemble.

```
1  def batch_function( func ):
2      global batchable_functions
3
4      pred_f = set( get_pred_node(func, [func], func.get_parent()) )
5      succ_f = set( get_succ_node(func, [func], func.get_parent()) )
6      pred_succ_f = set( get_pred_node( succ_f, [func], func.get_parent() ) )
7      succ_pred_f = set( get_succ_node( pred_f, [func], func.get_parent() ) )
8      pred_succ_f.update( succ_f )
9      succ_pred_f.update( pred_f )
10
11     i_loop = func.get_parent()
12     if len( succ_pred_f ) > 0:
13         pred_loop = NODES.Loop_node( func.get_context(), i_loop.get_dim(),
14             i_loop.get_parent() )
15         for f in succ_pred_f:
16             f.change_parent( pred_loop )
17         if i_loop.get_parent() != None:
18             i_loop.get_parent().add_node( pred_loop )
19
20     if len( pred_succ_f ) > 0:
21         succ_loop = NODES.Loop_node( func.get_context(), i_loop.get_dim(),
22             i_loop.get_parent() )
23         for f in pred_succ_f:
24             f.change_parent( succ_loop )
25         if i_loop.get_parent() != None:
26             i_loop.get_parent().add_node( succ_loop )
27
28     func.change_function( batchable_functions[ func.get_name() ] )
29     new_parent = func.get_parent().get_parent()
30     func.change_parent( new_parent )
31     i_loop.remove()
```

Code 3.4 – Découpe de la boucle

3.4.2 Choix des patterns mémoires

Après la phase de génération du graphe et la phase d'optimisation du graphe, l'outil va définir la façon de placer les données en mémoire. Les concepts de dimensions de tableaux et les annotations proposées permettent à l'utilisateur de ne pas définir l'ordre des données et de laisser cette tâche à l'outil. Avoir une définition par l'outil du placement en mémoire des données a deux intérêts : un mauvais placement (mémoire) des données peut nécessiter des réordonnancements au sein de la chaîne de traitement et la représentation des données d'un tableau multi-dimensionnel est difficile à se représenter, les définir à la main peut être compliquée. De plus, l'outil ajoute des dimensions aux tableaux manipulés pour permettre les pré-calculs et les 'batch', on ne souhaite pas que l'utilisateur ait à intervenir entre la phase d'optimisation et la phase de génération du code.

Pour illustrer cette section, nous allons nous baser sur un exemple basique disponible en figure 3.9. Cet exemple comporte deux multiplications de matrices chaînées, `C := A*B` et `E := C*D` et deux fonctions `src` et `dst` qui imposent un format aux données en entrée et en sortie.

L'objectif de la fonction de choix des patterns mémoire est d'éviter le besoin de réordonnancer les données entre deux nœuds. La première étape consiste, pour chaque tableau, à définir l'ensemble des ordres de données possibles en fonction des nœuds qui vont accéder au tableau.

Dans notre exemple : `src` demande un ordre des données de `A` : `[[p, z]]` alors que la fonction `gemm` accepte un ordre dans l'ensemble `[[p, z], [z, p]]`. L'ordre de `A` est alors choisi dans l'intersection de

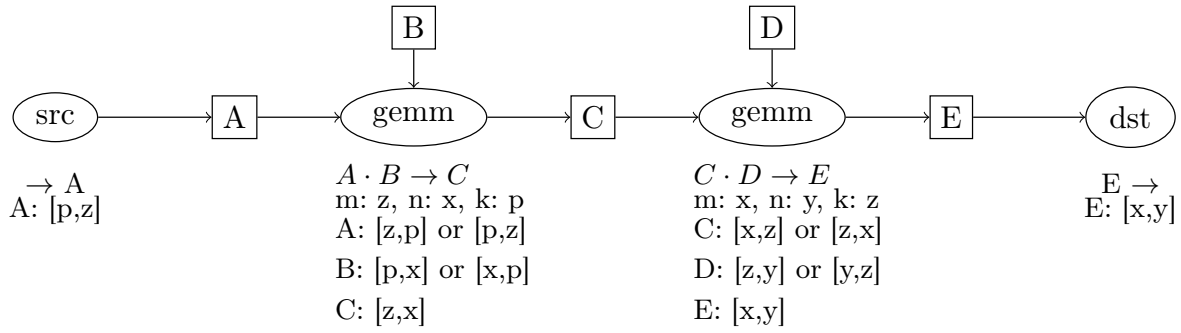


FIGURE 3.9 – Multiplications de matrices chaînées

ces deux ensembles, dans ce cas ce sera `[p, z]`. Si l'intersection est vide, une étape de réordonnancement est possible.

Dans le cas où une dimension est composée, par exemple $n_{hydro} = n_{colonne} \times n_{ligne}$, les contraintes sur l'ordre interne des dimensions atomiques sont transmises par la fonction. Pour ce faire, on crée un graphe $\mathbb{G}(V, E)$ avec les nœuds V représentant les tableaux mémoires. Une arête (u, v) est ajoutée si une dimension composée est utilisée par une fonction qui utilise les deux tableaux u et v en arguments.

Ce graphe sera alors séparé en clusters. Pour chaque cluster, le choix de l'ordre des données est indépendant : on calcule donc l'ensemble des patterns valides par intersection des patterns possibles, comme pour l'exemple précédent.

L'algorithme utilise un dictionnaire où pour chaque tableau/cluster, un ensemble de pattern mémoire valide est disponible. Dans notre exemple : `A: [[p,z]], B: [[p,x],[x,p]], C: [z,x], D: [[z,y],[y,z]], E: [x,y]`. On peut alors générer tous les placements mémoires valides en calculant l'ensemble des combinaisons. L'exemple précédent possède quatre placements mémoires valides.

3.4.3 Génération du code exécutable

Les sous-sections précédentes ont expliqué comment obtenir un contexte, un graphe des tâches et un ordre de stockage des tableaux en mémoire (placement mémoire). Nous avons également discuté précédemment de la façon de modifier le graphe pour appliquer les optimisations sur le code. La dernière étape consiste à générer un code C exécutable à partir de ces informations.

Génération du code utilisateur : Les parties du code non impactées par l'outil d'optimisation sont recopiées à partir du code d'origine (fonction main, variables globales, ...), en pratique les nœuds de l'AST du fichier d'entrée ont été sauvegardés et sont re-parcourus par le visiteur de `pycparser` qui écrit un code C à partir d'un AST. De plus, on ajoute des fonctions et des includes nécessaires au fonctionnement du code généré par le runtime (mesure de temps : `time.h`, ...).

Déclaration des dimensions et des tableaux : Une deuxième étape consiste à générer des variables globales en fonction des dimensions et des tableaux déclarés : par exemple chaque dimension atomique `dim_name` est associée à une variable `n_dim_name` que l'utilisateur doit utiliser pour indiquer les dimensions des tableaux à l'exécution.

Une fonction est générée pour allouer les tableaux mémoire `ps_allocate_memory`, elle doit être appelée par l'utilisateur. Cette fonction utilise `cudaMalloc` pour allouer les tableaux sur le GPU et définit la taille du tableau par le produit des dimensions qui le compose.

Génération de la fonction optimisée : Cette dernière étape de la génération consiste à générer une fonction exécutable en lieu et place de la fonction à optimiser déclarée par l'utilisateur. On utilise le graphe des tâches et le graphe hiérarchique comme source.

L'ordre de génération des fonctions et des nids de boucle découle d'un parcours en profondeur du graphe hiérarchique. De plus, l'ordre de génération des fils d'un même nœud est défini par les relations de dépendances au sein du graphe des dépendances. De ce fait, le parcours en profondeur assure que chaque fonction est dans le bon nid de boucles, tandis que l'application des dépendances assure que l'ordre d'exécution des fonctions sera valide vis-à-vis de l'ordre d'exécution original.

Pour chaque nœud, on génère une boucle qui itère sur la dimension associée, par exemple :

```
for (int i_dim0 = 0; i_dim0 < n_dim0; i_dim0++)
```

Ensuite, le corps de la boucle est généré.

Génération des appels de fonction : Les appels de fonction sont les feuilles du graphe hiérarchique. Pour chaque fonction, la génération est faite par l'AST par défaut de `pycparser`, sauf dans les cas des arguments spécifiques à notre outil : les fonctions `ps_blas_op`, `ps_ld` et `ps_stride` sont remplacées par des valeurs définies selon l'ordre choisi du placement mémoire des données.

Les dimensions atomiques sont remplacées par la taille de la dimension, par exemple `n_dim0` pour la dimension `dim0`. D'autre part, les dimensions composées sont remplacées par le produit des dimensions atomiques qui les composent : si `dimC = dim0,dim1` alors elle sera remplacée par `n_dim0*n_dim1`.

Finalement, les tableaux de données sont remplacés par la position dans le tableau en fonction de la position dans les boucles d'itération et de l'ordre des données en mémoire.

Si la fonction générée est pré-calculable, on ajoute une condition qui vérifie si le calcul est nécessaire. On calcule un identifiant d'itération unique en fonction des indices d'itération du nid de boucle, puis on stocke cet identifiant dans un tableau de correspondance à l'exécution de la fonction. Si le tableau contient déjà la même valeur, on peut ignorer l'exécution de la fonction.

3.4.4 Génération des fichiers de benchs

La dernière partie est un ensemble de scripts bash et d'exécutables utilisés pour exécuter automatiquement les codes générés. L'ensemble des appels est géré par un makefile et la vue générale est présentée en figure 3.8, page 29. Les fichiers sont compilés avec GCC et NVCC, avant d'être exécutés un par un sur la cible pour générer des logs de temps. Un script récupère en amont la configuration de la cible. À partir de ces deux fichiers, un script python va générer des plots à partir des logs à l'aide de Matplotlib et un rapport en latex qui résume la configuration et les résultats observés. Le rapport est compilé sous format pdf et archivé avec une représentation des graphes qui représente la fonction optimisée en fonction des optimisations. Ces graphes sont obtenus avec graphViz.

3.5 Expérimentation

Cette section présente d'une part les annotations sur le code 2.1 présenté en section 2.4.3, ainsi que les résultats de performances avec les optimisations des sections précédentes appliquées sur ce code.

3.5.1 Annotation du code

Le code annoté exécuté pour ces expériences est présenté dans les blocs de code suivants. Le code 3.5 contient les déclarations des variables et des tableaux. Le code 3.6 contient l'annotation des fonctions et la déclaration des fonctions supplémentaires. Le dernier code 3.7 contient l'annotation de la formation de voies à optimiser.

```
1 #pragma PS data atomicDim(Row, Col, Input, Fft, FreqPerGrp, SpectreToStore, Kx, Ky, Grp, Unary)
2
3 #pragma PS data composedDim(Hydro, Row, Col)
4 #pragma PS data composedDim(ColKx, Col, Kx)
5 #pragma PS data composedDim(RowK, Row, Kx, Ky)
6 #pragma PS data composedDim(RowSts, Row, SpectreToStore)
7 #pragma PS data composedDim(VoiesTotal, Ky, Kx, FreqPerGrp, Grp, SpectreToStore)
8 #pragma PS data composedDim(RowColSts, Row, SpectreToStore, Col)
9 #pragma PS data composedDim(GrpFrq, Grp, FreqPerGrp)
```

```
10
11 #pragma PS data array(float, d_InputBuffer, Hydro, Input)
12 #pragma PS data array(float, d_signalsToHandle, Hydro, Fft)
13 #pragma PS data array(float, d_ponderationFft, Fft)
14
15 #pragma PS data array(cuComplex, d_hydroSpectrum, Row, Col, Fft, SpectreToStore)
16 #pragma PS data array(cuComplex, d_hydroSpectrumFreq, Row, Col, FreqPerGrp, Grp, SpectreToStore)
17 #pragma PS data array(cuComplex, d_dthxFrq    , Col, Kx)
18 #pragma PS data array(cuComplex, d_dthxIncFrq, Col, Kx)
19 #pragma PS data array(cuComplex, d_dthxGrp    , Col, Kx)
20 #pragma PS data array(cuComplex, d_dthxGrp0   , Col, Kx)
21 #pragma PS data array(cuComplex, d_dthxIncGrp, Col, Kx)
22 #pragma PS data array(cuComplex, d_dthyIncGrp, Row, Ky, Kx)
23 #pragma PS data array(cuComplex, d_dthyGrp    , Row, Ky, Kx)
24 #pragma PS data array(cuComplex, d_dthyGrp0   , Row, Ky, Kx)
25
26 #pragma PS data array(cuComplex, d_ponderationX, Col)
27 #pragma PS data array(cuComplex, d_ponderationY, Row)
28 #pragma PS data array(cuComplex, d_Yf, Row, Kx, SpectreToStore)
29
30 #pragma PS data array(float, d_voies, Ky, Kx, FreqPerGrp, Grp, SpectreToStore)
31 #pragma PS data array(cuComplex, d_voiesCpx, Ky, Kx, FreqPerGrp, Grp, SpectreToStore)
```

Code 3.5 – Déclaration des dimensions et tableaux

```
1  #pragma PS data blasT(A, M, K) blasT(B, K, N) blas(C, M, N)
2  #pragma PS function output(C) input(A, B)
3  cublasStatus_t cublasCgemm(cublasHandle_t handle,
4                             cublasOperation_t transa, cublasOperation_t transb,
5                             int M, int N, int K,
6                             const cuComplex *alpha,
7                             const cuComplex *A, int lda,
8                             const cuComplex *B, int ldb,
9                             const cuComplex *beta,
10                            cuComplex *C, int ldc);
11
12 #pragma PS function input(A) output(B)
13 cufftResult cufftExecR2C( cufftHandle plan, float* A, cuComplex* B );
14
15 #pragma PS data fixed(A, N) fixed(B, N)
16 #pragma PS function input(A) output(B)
17 void vectorCabs( cudaStream_t cudaStream, cuComplex* A, float* B, int N );
18
19 #pragma PS data fixed(C, N, M) fixed(A, N, M) fixed(B, M)
20 #pragma PS function input(A, B) output(C)
21 void ponderate( cudaStream_t cudaStream, cuComplex* C, cuComplex* A, cuComplex* B, int M, int N );
22
23 #pragma PS data fixed(C, N, M) fixed(A, N, M) fixed(B, M)
24 #pragma PS function input(A, B) output(C)
25 void ponderateRR( cudaStream_t cudaStream, float* C, float* A, float* B, int M, int N );
26
27 #pragma PS data fixed(A, N) fixed(B, N)
28 #pragma PS function input(A) output(B)
29 void place( cudaStream_t cudaStream, cuComplex* B, cuComplex* A, int N );
30
31 #pragma PS data blasT(A, M, N) blasT(B, M, K)
32 #pragma PS function input(A) output(B)
33 void selectSubarray( cublasHandle_t handle, cublasOperation_t transa, cublasOperation_t transb,
34     int M, int N, int K,
35     cuComplex* A, int lda,
36     cuComplex* B, int ldb, int offset);
37
38 #pragma PS function output(A) volatile()
39 void source( float* A );
40
41 #pragma PS function input(A)
42 void sink( float* A );
43
44 void source( float* A ){
45     float* input0 = d_signalsToHandle;
46     float* input1 = &d_signalsToHandle[nRow*nCol*nInput];
47
48     cudaMemcpyAsync( input1, input0, nRow*nCol*nInput*sizeof( float ),
49                      cudaMemcpyDeviceToDevice, cudaStream );
50
51     cudaMemcpyAsync( input0, hostInput, nRow*nCol*nInput*sizeof( float ),
52                      cudaMemcpyHostToDevice, cudaStream );
53 }
54
```

```
55 void sink( float* A ){
56     float* output = d_voies;
57     cudaError_t error;
58 
59     unsigned long size = nKy*nKx*nFreqPerGrp*nGrp*nSpectreToStore*sizeof(float);
60     error = cudaMemcpyAsync( hostOutput, output, size, cudaMemcpyDeviceToHost, cudaStream );
61     gpuErrchk(error, "sink copy");
62 }
63 
64 void init_variables(){
65         // Dims
66     nRow = 37;
67     nCol = 300;
68     nInput = 4096;
69     nFft = 4096;
70     nFreqPerGrp = 10;
71     nSpectreToStore = 10;
72     nKx = 4*nCol;
73     nKy = 4*nRow;
74     nGrp = 20;
75     nUnary = 1;
76 
77     ps_allocate_memory();
78 }
79 
80 int main()
81 {
82     init_variables();
83     init();
84 
85     int COUNT = 10;
86     for(int i = 0; i < COUNT; i++){
87 
88         for( int i = 0; i < nRow*nCol*nInput; i++){
89             int rd = rand() % (1<<24);
90             hostInput[i] = (float) rd;
91         }
92 
93         traitement();
94     }
95 
96     return 0;
97 }
```

Code 3.6 – Déclaration des fonctions

```
1  #pragma PS optimize
2  void traitement(){
3  
4          for (;SpectreToStore;){
5  
6                  source( d_signalsToHandle );
7                          //source( d_InputBuffer );
8  
9                          //place( d_signalsToHandle, d_InputBuffer, 0 );
10                 ponderateRR( cudaStream, d_signalsToHandle, d_signalsToHandle, d_ponderationFft,
11                                 Fft, Hydro );
12 
13                 cufftExecR2C( plan, d_signalsToHandle, d_hydroSpectrum );
14         }
15 
16         selectSubarray( cublasHandle,
17                 ps_blas_op(d_hydroSpectrum, RowColSts, Fft),
18                 ps_blas_op(d_hydroSpectrumFreq, RowColSts, GrpFrq),
19                 RowColSts, Fft, GrpFrq,
20                 d_hydroSpectrum, ps_ld(d_hydroSpectrum, RowColSts, Fft),
21                 d_hydroSpectrumFreq, ps_ld(d_hydroSpectrumFreq, RowColSts, GrpFrq), 0);
22 
23 
24         place( cudaStream, d_dthxGrp, d_dthxGrp0, ColKx);
25         place( cudaStream, d_dthyGrp, d_dthyGrp0, RowK);
26 
27         for(;Grp;){
28 
29                 place( cudaStream, d_dthxFrq, d_dthxGrp , ColKx);
30 
31                 for(;FreqPerGrp;){
32 
33                         ponderate( cudaStream, d_hydroSpectrumFreq, d_hydroSpectrumFreq,
```

```
34                                      d_ponderationX, Col, RowSts );
35
36              for(;SpectreToStore;){
37
38                  cublasCgemm( cublasHandle,
39                     ps_blas_op(d_hydroSpectrumFreq, Row, Col),
40                     ps_blas_op(d_dthxFrq, Col, Kx),
41                     Row, Kx, Col,
42                     &alpha, d_hydroSpectrumFreq, ps_ld(d_hydroSpectrumFreq, Row, Col),
43                             d_dthxFrq, ps_ld(d_dthxFrq, Col, Kx),
44                             &beta, d_Yf, ps_ld(d_Yf, Row, Kx) );
45
46                      ponderate( cudaStream, d_Yf, d_Yf, d_ponderationY, Row, Kx );
47
48                  }
49
50              for(;Kx;){
51
52                  cublasCgemm( cublasHandle,
53                          ps_blas_op(d_dthyGrp, Ky, Row),
54                                      ps_blas_op(d_Yf, Row, SpectreToStore),
55                      Ky, SpectreToStore, Row,
56                      &alpha, d_dthyGrp, ps_ld(d_dthyGrp, Ky, Row),
57                      d_Yf, ps_ld(d_Yf, SpectreToStore, Row),
58                      &beta, d_voiesCpx, ps_ld(d_voiesCpx, Ky, SpectreToStore) );
59
60              }
61
62                          ponderate( cudaStream, d_dthxFrq, d_dthxFrq, d_dthxIncFrq, ColKx, Unary);
63                  }
64
65                  ponderate( cudaStream, d_dthxGrp, d_dthxGrp, d_dthxIncGrp, ColKx, Unary );
66                  ponderate( cudaStream, d_dthyGrp, d_dthyGrp, d_dthyIncGrp, RowK, Unary );
67          }
68
69          vectorCabs( cudaStream, d_voiesCpx, d_voies, VoiesTotal);
70
71          sink( d_voies );
72  }
```

Code 3.7 – Déclaration de la fonction à optimiser

3.5.2 Performances

L'exécution s'est déroulée sur un nœud gemini du site de Lyon de la plateforme grid5000 [5]. Il comporte 2 CPU Intel Xeon E5-2698 v4 cadencés à 2.2 GHz et 8 GPU Nvidia V100 SXM à 1.2 GHz. Le programme est séquentiel et s'exécute sur un seul cœur en utilisant un seul GPU. Il n'y a pas de synchronisation entre le code utilisateur et le GPU, excepté à la fin de chaque itération.

La configuration du système est la suivante :

— kernel : Linux 5.10.0-26-amd64
— system : Debian GNU/Linux 11 (bullseye)
— gcc : gcc (Debian 10.2.1-6) 10.2.1 20210110
— nvcc : V11.2.152
— CUDA driver : NVRM version : NVIDIA UNIX x86 64 Kernel Module 460.91.03

À partir de la même source annotée, trois exécutables sont générés et exécutés, nommés : `fvClassic`, `fvClassic_p` et `fvClassic_pb`. Les optimisations appliquées sont : aucune pour `fvClassic`, le pré-calcul pour `fvClassic_p` et le pré-calcul et les batchs pour `fvClassic_pb`.

La taille du problème est définie dans la fonction utilisateur `init_variables` définie dans le code 3.6 à la ligne 64. Les valeurs expérimentales sont reportées dans le tableau 3.2, elles ne représentent pas des données d'antennes réelles.

Dimension	nRow	nCol	nInput	nFft	nFreqPerGrp	nSpectreToStore	nKx	nKy	nGrp
Valeur	37	300	4096	4096	10	10	1200	148	20

Table 3.2 – Dimensions de la formation de voies

Modifications	Aucune	Pré-calculs	Pré-calculs + batch
Temps Première itération (s)	5.21	5.21	0.95
Temps itérations (s)	4.94	4.92	0.67

TABLE 3.3 – Temps d'exécution des programmes en fonction des modifications

Allocations mémoires. L'allocation mémoire nécessaire pour l'exécution de `fvClassic` est de 10.4 Go. Les extensions des tableaux permettant le pré-calcul (exécutable `fvClassic_p`) agrandissent les tableaux liés aux coefficients de déphasage. La mémoire allouée est augmentée de 33% pour un total de 11.9 Go.

Pour l'exécutable `fvClassic_pb`, le batch de multiplications de matrices s'applique au deuxième appel à la fonction GEMM code : 3.7 page 35, à la ligne 52. Les tableaux mémoires impactés sont uniquement ceux des coefficients de déphasages, or ils ont déjà été redimensionnés à leur taille maximale lors des modifications pour le pré-calcul. On n'a donc pas d'augmentation supplémentaire de l'espace mémoire par rapport à l'exécutable `fvClassic_p`.

Bien que dans ce cas particulier, la modification des batchs ne nécessite pas d'espace mémoire supplémentaire, il n'y a pas de raisons pour que ce soit le cas sur d'autres types de code.

Temps d'exécutions. Les temps d'exécution des différentes exécutions sont présentés en table 3.3. Plusieurs mesures sont effectuées. La première est le temps d'exécution de la première itération. La seconde est le temps moyen d'exécution des 20 itérations suivantes. L'écart type des mesures est inférieur à 1% pour chacune des exécutions.

Le temps d'exécution de la première itération est plus long pour deux raisons : cette itération effectue des opérations qui ne seront pas recalculées aux itérations suivantes. Et le programme n'ayant pas de warmup, il y a un surcoût aux premières exécutions pour l'initialisation des bibliothèques et le chargement des fonctions en mémoire GPU.

On peut observer que la différence de temps d'exécution entre la version sans optimisations et la version avec pré-calculs est négligeable, -0.02s (-0.4%). En revanche, le fait d'effectuer des batches d'opérations accélère fortement le programme avec un gain de 4.27s, soit un speed-up de 7.

La figure 3.10 représente les graphes de dépendances générés pour `fvClassic` (fig. 3.10a) sans optimisations et `fvClassic_pb` (fig. 3.10b) avec le pré-calcul et l'optimisation 'batch'. Les figures représentent une sous-partie du graphe contenant les boucles de formation des gisements et des pseudo-gisements, lignes 27 à 67 du code 3.7. En plus des fonctions, les tableaux mémoires utilisés sont représentés par un rectangle avec leur nom au centre et les dimensions allouées listées.

Sur la figure 3.10b, on distingue clairement les fonctions pré-calculables (couleur bleue sur la figure). De plus, on voit que la taille de certains tableaux mémoires a été augmentée et que l'appel à la fonction `cublasCgemm` de la ligne 52 du listing 3.7 a été batché (le nœud `fn_cublasCgemm_10` de la figure 3.10a est remplacé par le nœud `fn_cublasCgemmStridedBatched_10` de la figure 3.10b) [1].

Fusion de matrices La fusion de matrices est la troisième optimisation identifiée au cours de ces travaux, n'étant pas implémentée dans l'outil, il est nécessaire de faire la modification manuellement en modifiant le code d'entrée de l'outil.

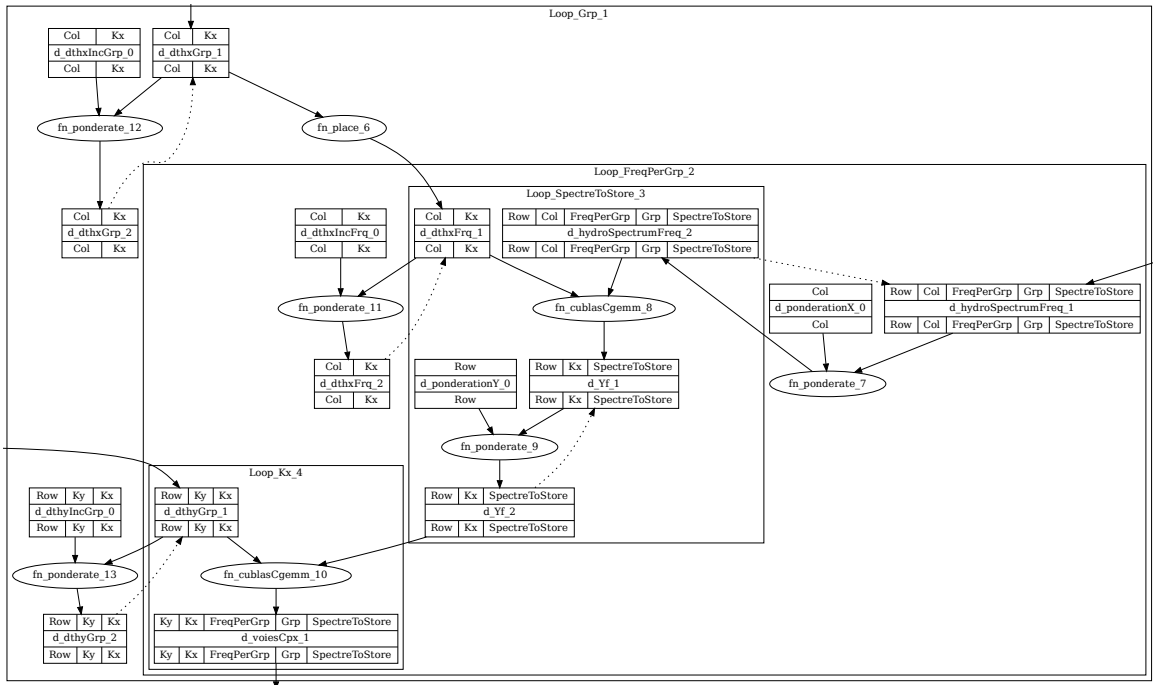
Dans ce cas, la boucle ligne 36 du code 3.7 est supprimée et la dimension `Row` est remplacée par la dimension composée `Row x SpectreToStore`.

Le graphe général du code est présenté en figure 3.11. (Pour améliorer la lisibilité, nous avons masqué les fonctions pré-calculables.) On note que toutes les fonctions ont été extraites des boucles.

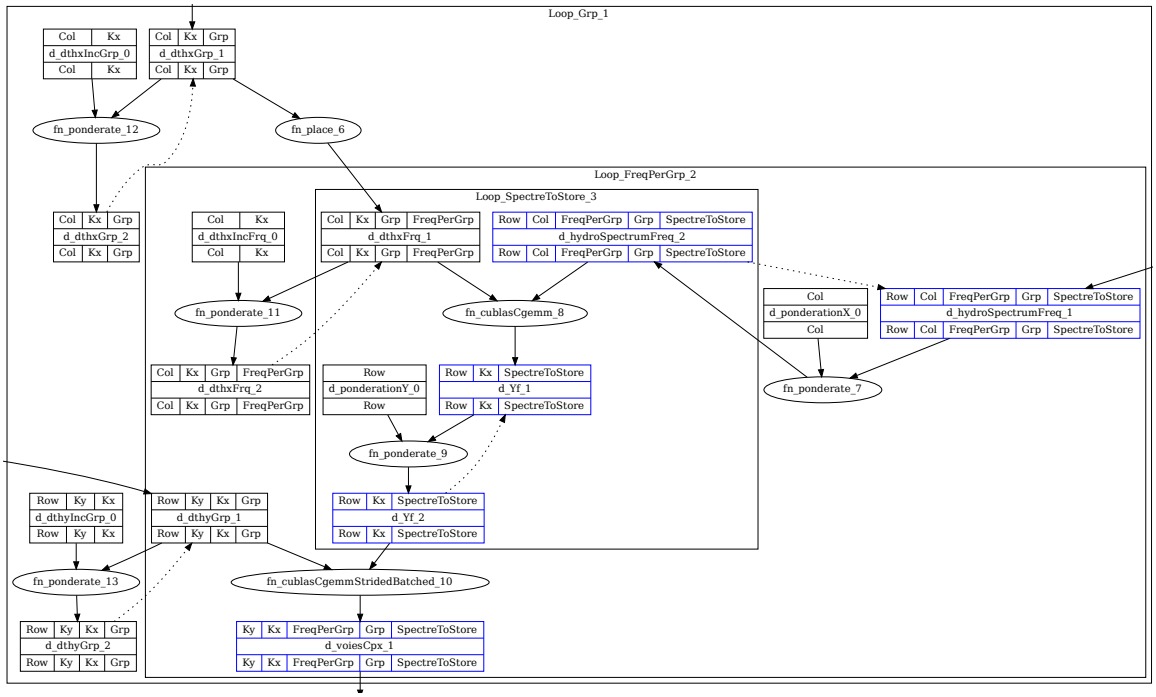
Le temps d'exécution mesuré dans cette configuration est de 0.52 seconde par itération, on a donc un speedup de x1.3 par rapport aux seules optimisations pré-calculs et batchs.

L'allocation mémoire est de 18.4 Go (+54%) par rapport à `fvClassic_pb`, en particulier, nous avons dû choisir un serveur disposant de la carte Nvidia V100 avec 32Go de mémoire pour exécuter cette instance.

1. Dans l'attente d'une meilleure mise en forme, il faut zoomer !



(a) Graphe partiel avant optimisation



(b) Graphe partiel avant optimisation

FIGURE 3.10 – Graphe représentant les boucles internes de l'algorithme de formation de voies

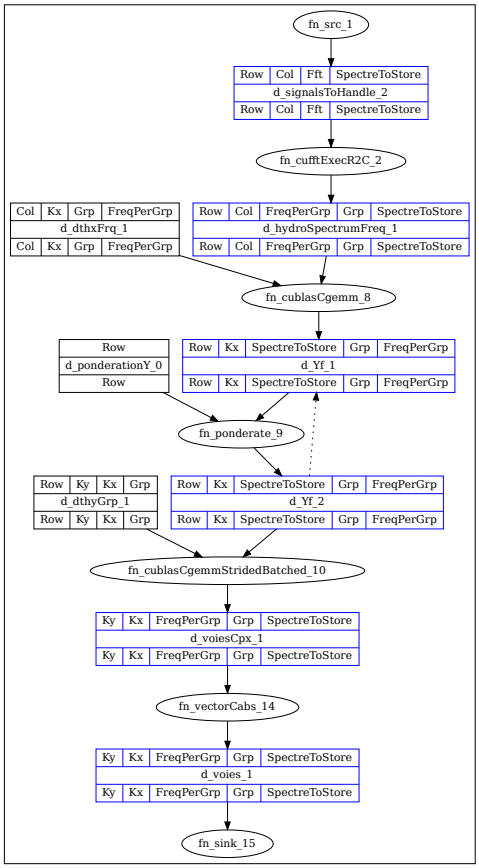


FIGURE 3.11 – Graphe simplifié après fusion de matrice

3.6 Bilan

Dans ce chapitre, nous avons présenté un outil de modification statique d'un code C annoté. L'objectif est de permettre de générer un code performant ciblant un GPU à partir d'un code non optimisé fourni par les équipes de numériciens. Dans le contexte des algorithmes de traitement SONAR, l'accent a été mis sur le regroupement d'appels de fonctions similaires pour limiter les surcoûts liés à l'exécution des kernels sur le GPU. Ces modifications permettent un gain de performances notable (de l'ordre de x7) de manière automatique dans le cadre d'algorithmes réels. En ajoutant manuellement certaines optimisations qui n'avaient pas été automatisées avant l'annotation du code, on approche un speedup de 9.5 là où les optimisations manuelles permettaient un speedup de 10. De plus, cet outil permet de définir le placement des données en mémoire et évite ainsi certains réordonnancements de données que l'utilisateur aurait pu effectuer avec une implémentation manuelle. Néanmoins, l'outil n'applique pas toutes les optimisations identifiées lors de notre phase d'optimisation manuelle.

Les méthodes utilisées pour effectuer ces modifications sont basiques, l'utilisation d'un AST est couramment utilisée par les outils de compilation, notamment LLVM [41]. Le parcours d'un graphe d'interaction pour détecter les données pré-calculables est similaire à certaines méthodes de détection du code-mort [14]. À notre connaissance, il n'y a pas de littérature concernant l'utilisation de ces méthodes pour regrouper des tâches similaires ensemble dans le but de les optimiser. Ces regroupements permettent de tirer parti des API Batch récemment ajoutées dans certaines bibliothèques de calcul, telles que BLAS [20], les gains de performances liés à l'utilisation de telles API sont documentés [46].

Malgré les performances prometteuses du code généré, il reste de nombreuses limitations. En particulier, le code généré est limité à une exécution sur un seul GPU, alors que les architectures actuelles sont hétérogènes et disposent généralement de plusieurs GPUs et de nombreux cœurs de calcul. De plus, l'extension des tableaux conduit à une sur-allocation mémoire qui empêcherait certaines tailles de problème d'être exécutées sur un GPU. Pour lever ces limitations et supporter une exécution sur une architecture multi-GPU, nous proposons dans la suite de la thèse de permettre de réajuster la granularité de nos tâches en fonction des ressources disponibles et de leurs contraintes mémoires. Pour cela, nous nous baserons sur le concept de tâche moldable et nous travaillerons sur son intégration dans un runtime d'exécution de tâches.

Le chapitre 4 présentera une syntaxe permettant d'ajouter le concept de tâche moldable à OpenMP. Quant au chapitre 5, il présentera un runtime d'exécution de tâches moldables pour architecture hétérogène.

Chapitre 4

Exprimer la moldabilité

4.1 Introduction

Le chapitre précédent 3 a montré l'intérêt de fusionner des tâches d'algèbre linéaire pour augmenter le taux d'exploitation des GPU. Néanmoins, augmenter la granularité des tâches implique un besoin plus important en mémoire de travail qui disqualifie certains coprocesseurs. De plus, cela réduit le parallélisme exprimé par notre graphe de tâche.

Ce chapitre proposera de représenter ces tâches fusionnées comme des tâches moldables. Grâce à cette « moldabilité », il sera alors possible d'augmenter le parallélisme et d'adapter la taille des tâches en fonction des caractéristiques des unités de calcul utilisées. De plus, en limitant la granularité, nous réduisons le nombre de tâches à ordonnancer et nous espérons limiter les surcoûts d'ordonnancement en conservant le parallélisme du code.

Ce chapitre est dédié à l'expression de la « moldabilité », nous avons décidé d'inscrire cette proposition dans le cadre d'OpenMP. Le chapitre suivant sera dédié à l'exécution des tâches moldables. L'organisation du chapitre est la suivante. La prochaine section 4.2 présentera le concept de tâches moldables, la section 4.4 proposera une nouvelle directive OpenMP pour exprimer la moldabilité d'une section de code, et des exemples d'utilisation seront disponibles en section 4.6.

4.2 Tâches moldables

Présentation

La classification de Feitelson et Rudolph [24] définit une tâche moldable comme une tâche qui peut s'exécuter sur un nombre arbitraire d'unités de calcul. Le nombre d'unités de calcul utilisé est défini par le système (l'ordonnanceur) avant le début de l'exécution de la tâche. Dans le cadre de ces travaux, nous limitons les tâches considérées. Premièrement, le nombre maximum de processeurs utilisables par une tâche est fini et dépend de la taille d'un espace d'itération défini par l'utilisateur ; une sous-tâche doit toujours être exécutable sur un seul processeur. La tâche moldable est associée à une fonction qui sera exécutée par toutes les sous-tâches. Cette fonction doit donc être un homomorphisme de liste [37, 9] qui s'applique à l'espace d'itération. Un homomorphisme de liste est une fonction h sur une liste, pour laquelle il existe un opérateur associatif $\odot$ tel que $h(x + +y) = h(x) \odot h(y)$ (avec $++$ la concaténation de liste). La suite de ce chapitre sera restreinte aux cas où les données en sortie de h sont indépendantes tant que les listes x et y ne se recouvrent pas, donc l'opérateur $\odot$ ne fait rien et toutes les exécutions de h sont indépendantes. Certains exemples où $\odot$ est une opération de réduction seront discutés.

Le champ de recherche lié à l'ordonnancement de ces tâches est très fertile [66], [58], [68], [21]. Par exemple, [10] propose une $(\frac{3}{2} + \epsilon)$ approximation pour minimiser le temps total d'exécution d'un ensemble de tâches moldables. Récemment, des méthodes d'ordonnancement à la volée pour des tâches moldables avec dépendances ont été proposées [7], [55]. Des travaux permettant d'optimiser d'autres métriques telles que l'énergie consommée existent également [8].

Runtime d'ordonnancement

Malgré les nombreux résultats d'un point de vue théorique, il y a eu peu de travaux permettant d'introduire un concept de tâche moldable dans les runtimes d'ordonnancement de tâches actuelles.

StarPU [4] a récemment introduit le concept de tâches hiérarchiques [22], [44], qui à l'exécution appelent une fonction de calculs ou soumettent un sous-graphe de tâches en fonction du type de processeurs. Cela permet une granularité dynamique, plus propice aux architectures hétérogènes. OpenMP supporte l'ordonnancement de tâches, et la directive `taskloop` [61], dont les travaux récents [45] propose une extension pour supporter les dépendances, peut s'apparenter à une forme de moldabilité. Les travaux liés à Kaapi [63],[64],[56] ont proposé des algorithmes à grain adaptatif où lors de l'exécution le choix a été fait entre une exécution séquentielle ou une exécution parallèle par la génération d'un graphe de tâches.

Nous proposons d'ajouter une directive `taskmoldable`, une forme d'extension de la directive `taskloop` pour déclarer de manière plus précise la moldabilité avec OpenMP. En particulier, elle doit permettre de paralléliser des appels à des fonctions de bibliothèque sans modification de leur code.

4.3 Illustration : fonctions moldables dans les API BLAS

API batch

La fusion de tâches pour une utilisation sur GPU a montré l'intérêt d'utiliser les fonctions de type `batch` de l'API cuBLAS. De plus, des extensions à l'API BLAS ont récemment été proposées pour standardiser les fonctions de type `batch` afin de : *"providing more efficient, but portable, implementations of algorithms on high-performance manycore architectures [19]"*. Des implémentations de ces fonctions sont déjà présentes dans les principales bibliothèques d'algèbre linéaire telles que Nvidia cuBLAS, Magma [36] ou Intel MKL.

Une exécution de `batch_count` multiplications de matrices est appelée de la manière suivante [2].

```
1 gemm_batch(m,n,k,A,B,C,batch_count);
```

où chaque paramètre est un tableau de taille `batch_count` du paramètre utilisé pour appeler le kernel `gemm` de BLAS, l'appel `batch` est équivalent à :

```
1 for (int i=0; i<batch_count; ++i)
2   gemm(m[i],n[i],k[i],A[i],B[i],C[i]);
```

L'appel à la fonction `gemm_batch` exécute `batch_count` appels indépendants au kernel BLAS `gemm`, où chaque `gemm` s'exécute avec des paramètres et des données différents fournis par des tableaux de paramètres.

La parallélisation de ces fonctions avec OpenMP pourrait se faire de manière classique avec la directive `taskloop`.

```
1 #pragma omp taskloop
2 for (int i=0; i<batch_count; ++i)
3   gemm(m[i],n[i],k[i],A[i],B[i],C[i]);
```

Nous avons montré dans le chapitre 3 précédent l'intérêt des appels de fonction `batch` pour la performance à l'exécution sur les co-processeurs. Lors de l'exécution des tâches générées par la directive `taskloop`, chaque tâche exécute un sous-ensemble des itérations de la boucle et ne peut pas tirer parti des optimisations apportées par l'API `batch`.

Nous proposons de voir ces kernels `batch` comme des tâches moldables. Du fait de la boucle implicite interne à la fonction, on peut imaginer découper l'exécution en plusieurs sous-ensembles des itérations de la boucle tout en gardant des appels aux kernels `batch` pour chacune des tâches.

Pour ce faire, nous proposons d'ajouter une directive `taskmoldable` à OpenMP, qui est définie en détail dans la section 4.4. Un appel à la directive devrait être sous la forme :

2. Pour simplifier, on omet certains paramètres tels que les opérations sur les matrices (transposition...), alpha et beta sont supposés égaux à 1, les leading dimensions et les paramètres de retour d'erreurs.

```
#pragma omp taskmoldable
gemm_batch(m,n,k,A,B,C,batch_count);
```

et être équivalent à un code de ce type, avec `I` le nombre d'itérations de la boucle exécuté par chaque tâche :

```
#pragma omp taskloop grainsize(I)
for (int i=0; i<batch_count; i=i+I)
  gemm_batch(&m[i],&n[i],&k[i],&A[i],&B[i],&C[i]);
```

De ce fait, chaque tâche exécuterait un sous-ensemble des `gemm` initiaux tout en utilisant les kernels `gemm_batch`qui ont été optimisés dans les bibliothèques d'algèbre linéaire.

Cette représentation moldable permettra alors de paralléliser l'exécution de la tâche entre plusieurs threads, tout en conservant pour chacun des threads une exécution sous forme de `batch` avec les avantages présentés précédemment.

Plus loin que les API batch

Les kernels `batch` ne sont pas les seuls à pouvoir être exécutés comme des tâches moldables. D'autres fonctions peuvent être concernées si l'algorithme qu'elles exécutent implicitement est composé d'un nid de boucle régulier sans dépendance entre les itérations de ces boucles. C'est le cas des fonctions effectuant des multiplications de matrices pleines dans l'API BLAS. Par exemple `gemm` et `syrk`, ces algorithmes calculent des multiplications de matrices sous la forme $C := \alpha A * B + \beta C$, dans le cas de `syrk` : $B = A^t$ et C est symétrique.

La fonction `gemm`, exécute un code sous la forme suivante :

```
void gemm(int m, int n, int k, double* A, double* B, double* C)
{
    for( int u = 0; u < m; u++ )          # (I)
        for( int v = 0; v < n; v++ )      # (II)
            for( int w = 0; w < k; w++ )  # (III)
                C[u*n + v] = C[u*n + v] + A[u * k + w] * A[w * n + v]
}
```

Les boucles `(I)` et `(II)` sont indépendantes, en revanche les opérations effectuées au sein de la boucle `(III)` ont une dépendance entre elles. Une manière classique de paralléliser une multiplication de matrice est de découper la matrice C en sous-matrices et d'attribuer le calcul de chacune d'entre elles de manière indépendante. Notons que le calcul de chacune de ces sous-matrices peut être effectué avec un appel au kernel *gemm* qui aura vu ses paramètres adaptés. Le code suivant correspond à une découpe de la matrice C en 4 blocs égaux et la découpe associée est représentée en figure 4.1.

```
void gemm(int m, int n, int k, double* A, double* B, double* C)
{
    gemm( m/2, n/2, k, A0, B0, C00 );
    gemm( m/2, n/2, k, A0, B1, C01 );
    gemm( m/2, n/2, k, A1, B0, C10 );
    gemm( m/2, n/2, k, A1, B1, C11 );
}
```

Dans cet exemple, on voudrait que la directive `taskmoldable` soit utilisée sur un `gemm`.

```
#pragma omp taskmoldable
gemm( m, n, k, A, B, C )
```

et que le code généré soit le suivant :

```
#pragma omp task
gemm( m/2, n/2, k, A0, B0, C00 )
```

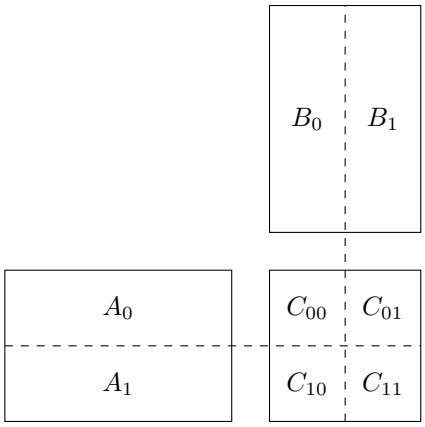


FIGURE 4.1 – Division en 4 d'un GEMM

```
3 #pragma omp task
4 gemm( m/2, n/2, k, A0, B1, C01 )
5 #pragma omp task
6 gemm( m/2, n/2, k, A1, B0, C10 )
7 #pragma omp task
8 gemm( m/2, n/2, k, A1, B1, C11 )
```

On peut alors voir ces fonctions, telles que `gemm` ou `gemm_batch`, comme des homomorphismes de liste [9, 37]. Soit deux listes $l1$ et $l2$, un opérateur de concaténation $++$ et une fonction h étant un homomorphisme de liste, on a l'équivalence :

$$h(l1 + +l2) \Leftrightarrow h(l1) \odot h(l2)$$

Avec $\odot$ un opérateur de réduction. Dans notre cas, la fonction h représente le bloc structurel qui suit la directive OpenMP. Dans l'exemple présenté, les quatre tâches sont indépendantes car elles écrivent des données à des emplacements mémoire C_{ij} distincts, mais il serait envisageable d'avoir une réduction sur une même zone mémoire, dans ce cas, il faudrait générer une dépendance d'exclusion mutuelle entre les tâches qui sont en conflit ou utiliser une méthode de réduction pour les fonctions qui s'y prêtent.

Les deux points majeurs soulevés par cette directive sont la méthode de définition de l'espace d'itération implicite et comment passer des paramètres valides aux sous-appels de fonction lorsque la tâche se répartit sur plusieurs processeurs.

4.4 La directive `taskmoldable`

La directive `taskmoldable` est, comme `taskloop`, une directive de génération de tâches (*task generation construct*). Elle améliore les fonctionnalités proposées par `taskloop` en permettant à l'utilisateur d'exploiter le parallélisme implicite de n'importe quelle structure de code grâce aux annotations de l'utilisateur.

4.4.1 structure générale

La directive `taskmoldable` est définie de la manière suivante :

```
1 #pragma omp taskmoldable batch_count(<counter-list>) \
2         access( <data-mapping-fct> [(args)] : <list-item> ) \
3         depend( <dependency-type> : <list-item> )\
4         num_tasks( <int> ) | grainsize( grain_size_list )
5 {<structured_block>}
```

La clause `batch_count` permet de définir l'espace d'itération implicite qui sera utilisé par le runtime pour définir la subdivision en sous-tâches, elle sera détaillée en sous-section 4.4.2.

La clause `access` permet à l'utilisateur de définir le schéma d'accès aux données, elle permet au runtime de définir les paramètres à utiliser dans chacune des sous-tâches. En association avec la clause `depend`, elle permet aussi de définir les dépendances entre les sous-tâches et les autres tâches générées par le runtime OpenMP. Le fonctionnement de ces clauses sera détaillé dans la sous-section 4.4.3.

Les tâches `num_tasks` et `grainsize`, déjà utilisées par `taskloop`, sont mutuellement exclusives et permettent respectivement de définir le nombre de tâches générées ou de définir la taille des tâches dans l'espace d'itération fourni par la clause `batch_count`.

Le fonctionnement attendu à la compilation et à l'exécution du runtime sera présenté dans la sous-section 4.5.

4.4.2 Espace d'itération

La directive `taskmoldable` a pour objectif de générer plusieurs tâches ; il nous faut donc définir le nombre de tâches qui peuvent être créées. La directive `taskloop` génère des tâches en fonction d'un nid de boucles et peut générer au maximum autant de tâches qu'il y a d'itérations dans le nid de boucle.

De manière similaire, la directive `taskmoldable` va se baser sur un espace d'itération pour diviser le travail. Il sera fourni par l'utilisateur à l'aide de la clause `batch_count(<counter-list>)` qui est obligatoire. Elle permet de définir un espace d'itération multi-dimensionnel où le nombre d'itérations dans chaque dimension est représenté par une valeur de l'argument `counter-list`. `counter-list` est une liste d'entiers positifs. L'espace d'itération représente un nid de boucles implicite interne au bloc d'instruction à paralléliser où le nombre d'itérations de la $i^{\text{ème}}$ boucle est égal au $i^{\text{ème}}$ élément de `counter-list`.

À l'exécution, le runtime va générer un groupe de tâches, qui vont chacune gérer un sous-ensemble de l'espace d'itération défini par la clause. Le comportement à la compilation est détaillé en sous-section 4.5.

Par exemple, le code suivant va pouvoir générer un nombre de tâches compris dans l'intervalle $[\![1, bcount]\!]$, chacune exécutant la fonction `gemm_batch`.

```
#pragma taskmoldable batchcount(bcount)
gemm_batch(m,n,k,A,B,C,bcount);
```

Code 4.1 – Exemple d'utilisation de la clause `batchcount` avec la fonction `gemm_batch`

Un exemple de découpe possible est de générer deux tâches traitant respectivement $\frac{1}{3}$ et $\frac{2}{3}$ des opérations. Dans ce cas, la première tâche travaillera sur l'intervalle $[\![1, \frac{bcount}{3}]\!]$, tandis que la seconde travaillera sur $[\![\frac{bcount}{3}+1, bcount]\!]$. Néanmoins, il manque les informations nécessaires pour que chaque fonction s'exécute avec les bons arguments, la clause data présentée dans la sous-section suivante 4.4.3 permettra de modifier les arguments en fonction de l'intervalle traité par la tâche.

4.4.3 Accès aux données

Pour générer des tâches, la directive `taskloop` peut conserver le code interne à la boucle et modifier uniquement les valeurs d'itérations initiales et de conditions d'arrêts. En revanche, la directive `taskmoldable` doit pouvoir gérer des codes plus génériques où le nid de boucle n'est pas accessible ou pas explicite. En particulier, l'objectif est de pouvoir générer des tâches à partir des fonctions `batch` telles que `gemm_batch`.

Chacune des tâches exécutera le même code avec des données d'entrées différentes. Pour définir ces données, la clause `access( <data-mapping-fct> [(args)] : <list-item> )` permet à l'utilisateur de fournir une fonction qui définit, pour chaque tâche, la valeur des arguments à utiliser. En pratique, la fonction `data-mapping-function` indique au runtime la nouvelle valeur à fournir pour chaque item de `<list-item>` en fonction des itérations exécutées.

Les fonctions doivent avoir une déclaration de cette forme :

```
1 type function( type item, int iter_start[], int iter_count[], args... );
```

La valeur de `item` dans la structure de donnée sur laquelle porte la clause sera remplacée par la valeur de retour de cette fonction. `iter_start` et `iter_count` sont des listes de la taille du nombre de dimensions de l'espace d'itération et qui représentent respectivement le premier élément à traiter et le nombre d'éléments à traiter pour chaque dimension.

Pour le code d'exemple 4.1 précédent, les tâches auront des valeurs de `iter_start` et `iter_count` définies en fonction des intervalles attribués. La première tâche qui doit travailler sur $[\![1, \frac{bcount}{3}]\!]$ aura $iter\_start = [0]$ et $iter\_count = \frac{bcount}{3}$, tandis que la seconde qui travaille sur $[\![\frac{bcount}{3} + 1, bcount]\!]$ aura $iter\_start = [\frac{bcount}{3}]$ et $iter\_stop = [2\frac{bcount}{3}]$.

Quelques fonctions basiques sont : la fonction `stride`, permet par exemple de définir comment est accédée une sous-matrice en fonction des "leading dimensions" de la matrice principale.

```
1 void* stride( void* A, int iter_start[], int iter_count[], int leading[] )
2 {
3         void* ret = A;
4         for( int i = 0; i < |iter_start|; i++ )
5         {
6                 ret += iter_start[i] * leading[i]
7         }
8         return ret;
9 }
```

Code 4.2 – Implémentation de la fonction `stride`

La fonction `size` permet de remplacer une variable par le nombre d'éléments traités pour un ensemble de dimensions.

```
1 int size( int item, int iter_start[], int iter_count[], int dims[] )
2 {
3         int ret = 1;
4         for( int i = 0; i < |dims|; i++ )
5         {
6                 ret *= iter_count[dims[i]];
7         }
8         return ret;
9 }
```

Code 4.3 – Implémentation de la fonction `size`

Ces fonctions peuvent être utilisées dans l'exemple précédent pour indiquer les variables à modifier dans l'appel de chaque tâche. Le code suivant 4.4 est un exemple d'utilisation de cette clause.

```
#pragma taskmoldable batchcount(bcount)\
        access( size(0), bcount )\
        access( stride(m*k), A )\
        access( stride(k*n), B )\
        access( stride(m*n), C )
gemm_batch(m,n,k,A,B,C,bcount);
```

Code 4.4 – Exemple d'utilisation de la clause `access` avec la fonction `gemm_batch`

Dans ce cas, les deux tâches vont remplacer les arguments `bcount`, `A`, `B` et `C` en fonction des intervalles $[\![1, \frac{bcount}{3}]\!]$, $[\![\frac{bcount}{3} + 1, bcount]\!]$ sur lesquels elles travaillent. La première tâche appellera alors la fonction `gemm_batch(m,n,k,A,B,C,bcount/3)` tandis que la deuxième appellera `gemm_batch( m, n, k, A+m*k*bcount/3, B+k*n*bcount/3, C+m*n*bcount/3, 2*bcount/3)`. De cette façon, chaque tâche va travailler sur des données différentes et l'exécution des deux tâches sera équivalente à l'appel initial à la fonction `gemm_batch`.

Un point majeur d'une gestion du parallélisme par tâches est de prendre en compte les dépendances entre les tâches pour effectuer un ordonnancement. En particulier, la gestion des dépendances sur les données. L'utilisation de la clause `depend` est possible avec la directive `taskmoldable`, de plus elle est

impactée par la clause `access` pour permettre à chaque tâche de générer des dépendances sur les données réellement accédées.

Limitation sur les dépendances : Il est envisageable que les sous-tâches d'une même tâche moldable aient des dépendances entre elles, par exemple dans le cas d'une opération de réduction. Cependant, l'ordre d'exécution des tâches générées par le runtime n'est pas explicité par les clauses et peut être différent de celui du nid de boucle implicite, en particulier lorsque l'espace d'itération comporte plusieurs dimensions. Il est donc nécessaire que les opérations effectuées soient commutatives car la directive ne permet pas d'assurer un ordre de génération des tâches.

4.5 Compilation

L'objectif de cette section est de présenter le fonctionnement de la directive `taskmoldable` à la compilation et son interaction avec le runtime. La directive `taskmoldable` va créer une tâche qui va elle-même générer des sous-tâches lors de son exécution. Pour cette section, nous allons suivre un exemple de parallélisation d'une fonction `gemm` par la directive '`taskmoldable`.

```
1  #pragma omp taskmoldable batch_count(m,n)\
2      access( size(0) : m )\
3      access( size(1) : n )\
4      access( stride(1,lda) : A )\
5      access( stride(0,ldb) : B )\
6      access( stride(1,ldc) : C )\
7      depend( in: A, B )\
8      depend( inout: C )\
9      num_tasks(4)
10 gemm( m, n, k, A, lda, B, ldb, C, ldc );
```

Code 4.5 – Code exemple pour la section compilation

On rappelle ici que la clause `batch_count(m,n)` définit un espace d'itération dans $I = [\![1, m]\!] \times [\![1, n]\!]$ où le runtime va attribuer des sous-ensembles de $I_i \subset I$ rectangulaires et disjoints à chacune des tâches. La clause `access` indique par quoi remplacer les arguments lorsque la fonction `gemm` sera appelée par chacune des tâches. Par exemple, la clause `access( size(0) : m )` va remplacer la variable m dans chaque appel de la fonction `gemm` par la taille de la première dimension de I_i. De même, la clause `access( size(1) : n )` va remplacer la variable n par de la taille de la deuxième dimension de I_i. Les clauses `depend` indiquent des dépendances de données sur les pointeurs manipulés, tandis que la clause `num_tasks` indique au runtime qu'il faut générer 4 tâches.

Espace d'itération

Une tâche moldable sans clauses `access` et `depend` peut être remplacée par le code suivant :

```
1  #pragma omp task
2  {
3      tasks = _kmpc_omp_taskmoldable( batch_count, num_tasks );
4      for( T in tasks )
5      {
6          #pragma omp task
7          {<structured block>}
8      }
9  }
```

Code 4.6 – Implémentation minimale d'une tâche moldable

où `_kmpc_omp_taskmoldable` est la fonction du support exécutif qui découpe l'espace d'itérations. Elle a pour objectif de découper l'espace d'itérations en bloc et de retourner une liste de structure de données

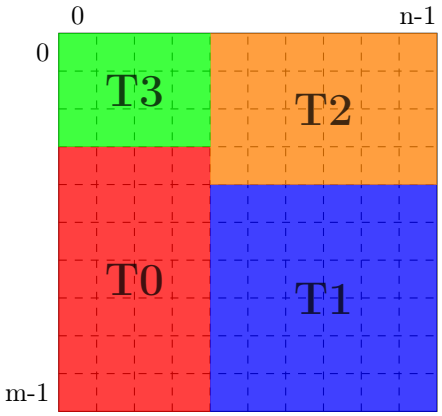


FIGURE 4.2 – Découpe possible de l'espace d'itération

qui décrira chacun des blocs (première itération sur chaque dimension et taille). Elle doit suivre si possible les indications fournies par les clauses `num_tasks` ou `grainsize` pour définir le nombre de bloc. Dans le chapitre 5, nous proposerons l'implémentation d'une fonction équivalente qui dimensionne la découpe en fonction des performances mesurées des unités de calculs pour les itérations précédentes, de manière proche de ce que fait StarPU [4].

Pour l'exemple du `gemm`, une découpe de l'espace d'itération possible en 4 est proposée en figure 4.2. Dans cet exemple, les tâches **T0**, **T1**, **T2** et **T3** générées par la fonction `_kmpc_omp_taskmoldable` auront respectivement les itérations de départ suivantes : $(\frac{3m}{10},0)$, $(\frac{4m}{10},\frac{4n}{10})$, $(0,\frac{4n}{0})$, $(0,0)$ et les tailles suivantes : $(\frac{7m}{10},\frac{4n}{10})$, $(\frac{6m}{10},\frac{6n}{10})$, $(\frac{4m}{10},\frac{6n}{10})$, $(\frac{3m}{10},\frac{4n}{10})$. On notera pour la suite de cette section que les itérations de départ sont contenues dans la structure de donnée `T.start` et la taille dans `T.size`. Ces valeurs sont listées dans le tableau 4.1.

tâche	start	size
T0	$[\frac{3m}{10}-1,0]$	$[7,4]$
T1	$[\frac{4m}{10}-1,\frac{3n}{10}-1]$	$[6,6]$
T2	$[0,\frac{4n}{10}-1]$	$[4,6]$
T3	$[0,0]$	$[3,4]$

TABLE 4.1 – Valeurs start et size pour l'exemple de parallélisation de `gemm`

Notons que pour pouvoir effectuer le bon appel `gemm` sur chacun des espaces d'itération définis par les blocs `T0,T1,T2` et `T3`, il est nécessaire de transformer les arguments d'appel au kernel `gemm` dans chacune des tâches générées. Cela est l'objet des sections suivantes.

Remplacement des variables

La clause `access` permet de remplacer des variables du bloc impacté par la directive `taskmoldable`. Dans l'exemple 4.5, la clause `access` est utilisée de deux manières différentes : la fonction `size` est utilisée pour remplacer les valeurs de `m` et `n` et définir la taille des nouvelles opérations, tandis que la fonction `stride` permet de définir les sous-matrices de travail de chacune des tâches. Ces fonctions sont définies à la section 4.4.3.

Le résultat dans le code exécuté par la tâche moldable est le suivant :

```
1 #pragma omp task
2 {
3     tasks = _kmpc_omp_taskmoldable( batch_count, num_tasks );
4     for( T in tasks )
5     {
```

```
6           #pragma omp task
7           gemm( size(m, T.start, T.size, [0]), size(n, T.start, T.size, [1]), k,
8               stride(A, T.start, T.size, [1,lda]), lda,
9               stride(B, T.start, T.size, [0,ldb]), ldb,
10              stride(C, T.start, T.size, [1,ldc]), ldc );
11      }
12  }
```

Code 4.7 – Remplacement des variables pour le code `gemm`

Les fonctions `strides` et `size` sont celles indiquées dans la clause `access` pour redéfinir les arguments, leurs implémentations sont présentées avec les codes 4.2 et 4.3.

L'exécution va alors appeler la fonction `gemm` quatre fois, une fois par chaque tâche, avec les arguments suivants :

— `gemm(7,4,k,A+3*m/10,B,C+3*m/10)`,
— `gemm(6,6,k,A+4*m/10+3*n/10*lda,B+3*n/10*ldb,C+4*m/10+3*n/10*ldc)`,
— `gemm(3,4,k,A+4*n/10*lda,B4*n/10*ldb,C+4*n/10*ldc)`,
— `gemm(4,6,k,A,B,C)`

L'exécution va donc calculer l'intégralité des valeurs de `C`.

Gestion des dépendances

La gestion des dépendances est un point sensible de définition de la directive `taskmoldable`. On rappelle que la spécification et la majorité des runtimes OpenMP ne gèrent pas les dépendances entre des régions mémoires qui ont un recouvrement partiel. Cette limitation complique la génération des dépendances pour les tâches moldables car les régions mémoires impactées par les sous-tâches peuvent être moins régulières que celles créées par un utilisateur lors d'une parallélisation explicite où le degré de parallélisme est choisi par le programmeur.

Les dépendances pour une sous-tâche `T` sont définies de la manière suivante : lorsque qu'une clause `depend` contient une variable impactée par la clause `access`, elle est remplacée par la liste des itérations gérées par `T`. C'est-à-dire que pour chaque itération le runtime crée une dépendance sur : `<data-mapping-fct>(item,iteration_start,[1,1,1,...],args)`. Par exemple, pour la tâche **T3** de l'exemple 4.2, et sachant que **T3**.start $= [0, 0]$ et **T3**.size $= [3, 4]$, la clause suivante sera générée pour la variable `C` :

```
depend(inout: C, C+1, C+2,
    C+ldc, C+ldc+1, C+ldc+2,
    C+2*ldc, C+2*ldc+1, C+2*ldc+2,
    C+3*ldc, C+3*ldc+1, C+3*ldc+2 )
```

De plus, la tâche `taskmoldable` doit aussi déclarer ces dépendances pour que les sous-tâches ainsi créées ne puissent être mal réordonnancées avec les tâches sœurs, ou leurs tâches filles, de la tâche `taskmoldable`. Cette tâche n'accédant pas directement aux données, on peut lui attribuer des dépendances de type "weak-dependencies". Ces dépendances, déjà gérées par certains runtimes de tâches, ont été proposées pour OpenMP [54]. Elles permettent à l'utilisateur d'indiquer qu'une tâche va générer des sous-tâches avec des dépendances sur certaines données, ce qui permet au runtime de générer les tâches avant la disponibilité des données tout en gardant un ordre valide. La clause *depend* de cette tâche devra couvrir l'ensemble des dépendances des tâches filles, à la seule différence qu'elles seront converties en "weak-dependency" si le runtime le supporte.

Une alternative aux dépendances "weak" est de conserver les dépendances pour la tâche génératrice, dans ce cas cela reviendrait à avoir une barrière entre les tâches précédentes et les tâches filles de la tâche moldable (avec une structure de dépendance de type fork-join). Une seconde solution est de ne pas générer de tâche génératrice et de générer directement les tâches filles, on évite donc la synchronisation, mais la découpe de la tâche moldable est alors faite en amont de la génération, ce qui ne permet pas d'avoir la même connaissance du système que dans les solutions précédentes.

Le code final généré à la place de l'exemple 4.5 :

```
1  #pragma omp task depend( in-weak : A..., B... ) depend( inout-weak : C... )
2  {
3      tasks = _kmpc_omp_taskmoldable( batch_count, num_tasks );
4      for( T in tasks )
5      {
6          #pragma omp task depend( in : C... ) depend( inout : A..., B... )
7          gemm( size(m, T.start, T.size, [0]), size(n, T.start, T.size, [1]), k,
8              stride(A, T.start, T.size, [1,lda]), lda,
9              stride(B, T.start, T.size, [0,ldb]), ldb,
10             stride(C, T.start, T.size, [1,ldc]), ldc );
11     }
12 }
```

Code 4.8 – Code complet `taskmoldable` de `gemm`

Limitations liées aux dépendances : La moldabilité des tâches induit que chaque sous-tâche travaillera sur des régions mémoires qui varieront en fonction de la découpe. Dans ce chapitre, nous avons présenté une méthode qui consiste à définir une dépendance pour chaque tableau accédé et pour chaque itération traitée par la tâche, cela permet, par exemple, à deux tâches qui travaillent sur des sous-matrices disjointes d'une même matrice A de ne pas générer une dépendance inutile entre ces deux tâches.

Néanmoins, la déclaration de nombreuses données avec la clause `depend` va impacter les performances du runtime et posera des problèmes de mise à l'échelle [53]. Une solution possible pour limiter cela serait d'avoir un support des dépendances sur les intersections de régions mémoires dans OpenMP.

4.6 Exemples d'utilisation

Cette section présente des exemples d'utilisation de la directive `taskmoldable`. Des fonctions BLAS élémentaires `gemm` et `syrk` seront présentées en sous-sections 4.6.2 et 4.6.1, une proposition d'implémentation de la factorisation de Cholesky à la section 4.6.3, enfin une implémentation de la formation de voies classique sera présentée en section 4.6.4.

4.6.1 Symetric rank-k update

La fonction `syrk` effectue le calcul `C := alpha*A*A' + C`, où `alpha` est un coefficient scalaire, `A` est une matrice de taille `n,k` et `C` une matrice triangulaire de taille `n,n`. La figure 4.3 schématise les données de l'opération.

(a) Schéma de `syrk` (b) Découpe en 4 avec le code 4.9

FIGURE 4.3 – Schéma de `syrk`

Le fait que la matrice `C` soit triangulaire permet des optimisations au niveau de l'implémentation de cette fonction, et il n'y a pas de nid de boucle régulier implicite permettant une découpe 2D comme pour la fonction `gemm`. On propose donc de faire une division selon la dimension k. Le code 4.9 présente l'implémentation, une découpe associée est présentée en figure 4.3b. On suppose que les données sont placées dans le format "column major", la variable `lda` représente la distance entre les deux premiers éléments de deux colonnes consécutives.

```
#pragma omp taskmoldable batch_count(k)\
    access( size(0) : k )\
    access( stride(lda) : A )\
    depend( in: A )\
    depend( inout: C )
syrk( n, k, alpha, A, lda, 1, C, ldc );
```

Code 4.9 – `syrk`

Les tâches générées vont travailler avec des sous-matrices A_i de A de tailles k_i en entrée et sur la matrice `C` en sortie. L'espace d'itération est $[\![1, k]\!]$ défini par la clause `batch_count(k)`. La clause `access( size(0) : k)` conduit à remplacer l'argument `k` de l'appel de fonction par la taille de l'espace d'itération attribué à la tâche. Et la clause `access( stride(lda) : A )` modifie le pointeur de travail de la tâche.

Cette découpe implique une dépendance sur la matrice `C` entre toutes les tâches générées. De ce fait, le parallélisme n'est pas augmenté. En revanche, séparer en sous-tâches permet aux tâches d'avoir des dépendances sur les sous-matrices A_0, A_1, A_2, A_3 au lieu d'une dépendance sur toute la matrice `A`. Cela permet de démarrer le calcul plus rapidement si les tâches génératrices de `A` travaillent aussi par bloc. De plus, les opérations sur `C` étant commutatives, il est possible de remplacer la dépendance `inout` par une dépendance `mutex-inout-set` pour restreindre la contrainte d'ordre d'exécution des tâches entre elles.

Dans le cas où l'on voudrait accumuler la matrice `C` en la pondérant par un scalaire `beta`, *i.e.* calculer `C := alpha*A*A' + beta*C`, le code n'est plus aussi simple au risque que chaque sous-tâche créée par la directive `taskmoldable` effectue une pondération par `beta`. Il faudrait séparer la partie pondération de la partie `syrk`. On exécuterait donc deux tâches : `geam` pour pondérer `C` par `beta` puis `syrk`.

4.6.2 Multiplication de matrice

La multiplication de matrix (`gemm`) est très proche de `syrk`, l'opération effectuée étant :
avec `A` est une matrice de taille `m,k`, `B` de taille `k,n` et `C` de taille `m,n`. Elle est plus générale au sens où `B` est différent de `A` et `C` n'est pas triangulaire, dans ce cas une découpe 2D selon les dimensions `m,n` tel que présenté au début du chapitre en figure 4.1.

La directive suivante permet une découpe en bloc 2D (ligne/colonne), identique à celle présentée en figure 4.1 (page 44) :

```
#pragma omp taskmoldable batch_count(m,n)\
    access( size(0) : m )\
    access( size(1) : n )\
    access( stride(1,lda) : A )\
    access( stride(0,1) : B )\
    access( stride(1,ldc) : C )\
    depend( in: A, B )\
    depend( inout: C )
gemm( m, n, k, alpha, A, lda, B, ldb, beta, C, ldc );
```

Code 4.10 – `gemm`

Une découpe classique en bloc, sur les dimensions `m`, `n` et `k`, introduira des dépendances entre certaines sous-tâches. Dans ce cas, une pondération de la matrice `C` par `beta` devra être séparée du `gemm` si `beta` est différent de 1.

4.6.3 Factorisation de Cholesky

La factorisation de Cholesky (`potrf`) est un algorithme permettant de trouver la valeur de L dans l'équation $A = LL^H$, L étant une matrice triangulaire inférieure et A une matrice symétrique définie positive. Cet algorithme est fréquemment utilisé dans les algorithmes de simulation numérique, il peut être utilisé en traitement du signal par exemple pour certains algorithmes de formation de voies adaptative [27].

Une méthode d'implémentation de l'algorithme de Cholesky sur de grandes matrices est de découper la matrice par bloc et d'utiliser les fonctions d'algèbre linéaire `gemm` et `syrk` et de résolution de système linéaire `trsm` et `potrf`, cette version est un exemple commun lorsque l'on travaille sur l'ordonnancement par tâche [40].

Ici, nous allons distinguer deux implémentations de l'algorithme de Cholesky.

La version par bloc classique (code 4.11) dont le principe est de calculer un bloc de données dès qu'il est disponible. L'avantage est d'exposer un haut niveau de parallélisme.

```
1  void Cholesky_right_looking( double* A, int N, size_t NB )
2  {
3      for (size_t k=0; k < N; k += NB)
4      {
5          potrf( NB, &A[k*N+k], N );
6          for (size_t m=k+ NB; m < N; m += NB)
7          {
8              trsm ( NB, NB, 1, &A[k*N+k], N, &A[m*N+k], N );
9          }
10         for (size_t m=k+ NB; m < N; m += NB)
11         {
12             syrk ( NB, NB, -1.0, &A[m*N+k], N, 1.0, &A[m*N+m], N );
13             for (size_t n=k+NB; n < m; n += NB)
14             {
15                 gemm ( CblasNoTrans, CblasTrans, NB, NB, NB,
16                     -1.0, &A[m*N+k], N, &A[n*N+k], N, 1.0, &A[m*N+n], N );
17             }
18         }
19     }
20 }
```

Code 4.11 – Bloc Cholesky

La seconde version est la Left Looking Cholesky (code 4.12), le principe est de mettre à jour une colonne de la matrice par itération en partant de la gauche. Une figure représentant les accès mémoires à chaque itération est disponible en figure 4.4. Bien que le code soit élégant et relativement simple, il n'exprime que peu de parallélisme. Cette version de l'algorithme de Cholesky est utilisée dans l'implémentation de Lapack Netlib [17].

```
1  void Cholesky_left_looking( A, N, NB, lda )
2  {
3      for( j = 0; j < N; j += NB )
4      {
5          syrk ( NB, j, -1, &A[0][j], lda, 1, &A[j][j], lda );
6
7          potrf( NB, &A[j][j], lda );
8
9          if( j+NB < N )
10         {
11             gemm( NB, N - j - NB, j, -1, A[0][j], lda, A[0][j+NB],
12                 lda, 1, A[j][j+NB], lda )
13
14             trsm( NB, N - j - NB, 1, A[j][j], lda, A[j][j+NB], lda )
15         }
16     }
17 }
```

Code 4.12 – Left Looking Cholesky

En considérant les fonctions `syrk`, `trsm` et `gemm` comme des tâches moldables, et en contrôlant la granularité de la découpe, nous pouvons obtenir un niveau de parallélisme similaire à l'implémentation

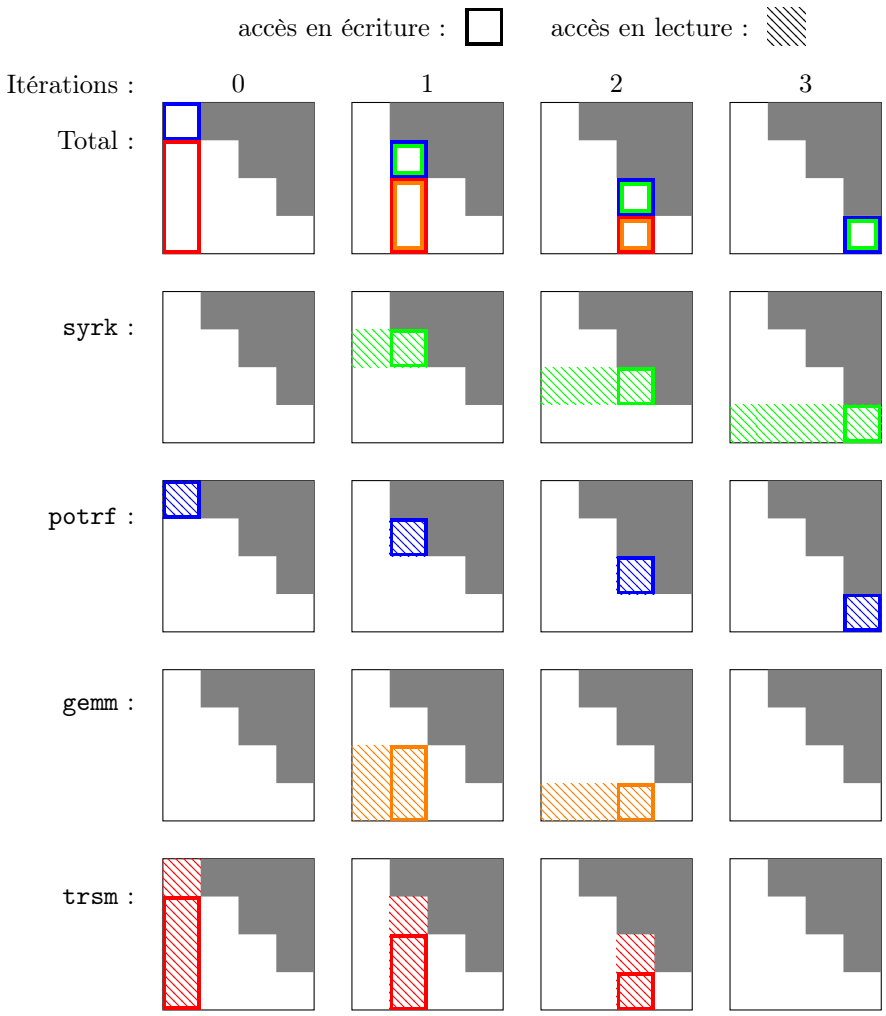
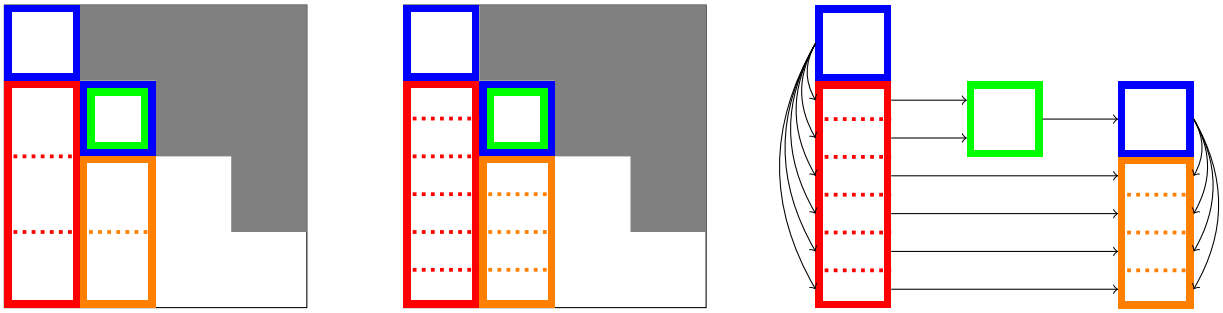


FIGURE 4.4 – Left-looking Cholesky étape par étape pour une découpe 4 x 4



(a) Découpe en bloc de taille NB

(b) Découpe en bloc de taille NB/2

(c) Dépendances

FIGURE 4.5 – Moldabilité de l'implémentation Left-looking

Right-Looking. Nous avons précédemment montré comment exprimer le parallélisme de `syrk` et `gemm`, de plus la fonction `trsm` a des accès mémoires similaires à `syrk` et peut être parallélisée de la même façon. Pour arriver au parallélisme souhaité, la granularité nécessaire est la taille du bloc `NB`. Le code annoté est fourni dans le code 4.13. Le graphe des tâches généré par ce code est équivalent à celui de la version Right-looking dans le cas où l'on ajoute la clause `grainsize` limitant à des tailles de `NB`. Il est possible de générer plus de parallélisme en diminuant la valeur de `grainsize`, des exemples de découpe possible et la génération des dépendances sont disponibles en figure 4.5.

```
1  void Cholesky_left_looking( A, N, NB, lda )
2  {
3      for( j = 0; j < N; j += NB )
4      {
5          #pragma omp taskmoldable batch_count(j)\
6              access( size(0) : j )\
7              access( stride(lda) : A[0][j] )\
8              depend( in: A[0][j] )\
9              depend( inout: A[j][j] )
10         syrk ( NB, j, -1, A[0][j], lda, 1, A[j][j], lda );
11
12         #pragma omp task depend( inout:A[j][j] )
13         potrf( NB, &A[j][j], lda );
14
15         if( j+NB < N )
16         {
17             #pragma omp taskmoldable batch_count((NB,N-j-NB,j))\
18                 access( size(0) : m )\
19                 access( size(1) : n )\
20                 access( size(2) : k )\
21                 access( stride(1,0,lda) : A[0][j] )\
22                 access( stride(0,ldb,1) : A[0][j+NB] )\
23                 access( stride(1,ldc,0) : C )\
24                 depend( in: A[0][j], A[0][j+NB] )\
25                 depend( inout: A[j][j+NB] )
26             gemm( NB, N - j - NB, j, -1, A[0][j], lda, A[0][j+NB],
27                 lda, 1, A[j][j+NB], lda )
28
29             #pragma omp taskmoldable batch_count(N - j - NB)\
30                 access( size(0) : N - j - NB )\
31                 access( stride(lda) : A[j][j+NB] )\
32                 depend( in: A[j][j] )\
33                 depend( inout: A[j][j+NB] )
34             trsm( NB, N - j - NB, 1, A[j][j], lda, A[j][j+NB], lda )
```

```
35          }
36      }
37  }
```

Code 4.13 – Left Looking Cholesky with moldability

Cette version de l'algorithme de Cholesky permet d'autoriser une granularité plus fine qu'une découpe par bloc classique sans générer un nombre excessif de tâches. Une décomposition de Cholesky « right-looking » génère $\mathcal{O}(\frac{N}{NB})^3$ tâches, où une tâche représente un appel à `potrf`, `gemm`, `trsm` ou `syrk`. Tandis qu'une décomposition « left-looking » en génère $\mathcal{O}(\frac{N}{NB})$ tâches (4 par groupe de colonnes). Or chacune de ces tâches est une tâche moldable, ce qui permet de décomposer ces $\mathcal{O}(\frac{N}{NB})$ tâches moldables en $\mathcal{O}(\frac{N}{NB})^3$ ou plus selon le nombre de cœurs à disposition.

Néanmoins, l'exploitation efficace de cette découpe adaptative n'est pas évidente. Dans le chapitre suivant, nous détaillons des expérimentations sur cet algorithme en section 5.6.4 pour lesquelles les performances ne sont hélas pas au rendez-vous.

4.6.4 Formation de voies classique

L'algorithme de formation de voies, dont le principe a été présenté en section 2.4, peut aussi profiter de la moldabilité. Rappelons que le travail présenté dans le chapitre 3 nous a permis de montrer l'intérêt de grouper les opérations des algorithmes de traitement du signal SONAR pour augmenter le taux d'utilisation du GPU. Ces modifications ont été faites, soit en modifiant des appels de fonctions BLAS, soit en utilisant les variantes `batch` de ces fonctions. Or la moldabilité d'un batch d'opérations est évidente car par définition chacune de ces opérations est indépendante, de plus, on peut se contenter d'une découpe uni-dimensionnelle.

```
1   beamforming()
2   {
3     #pragma omp taskmoldable batch_count(fft_count) \
4         depend(in:Sensor_t) depend(out:Sensor_f)\
5         access(strided{fft_stize}:Sensor_t,Sensor_f)
6     fft1DExecBatch( fft_size, Sensor_t, Sensor_f, fft_count )
7
8     #pragma omp taskmoldable batch_count( fft_count )\
9         depend(in:Sensor_f) depend(out:Sensor_f2)\
10        access(strided{fft_size}:Sensor_f)\
11        access(strided{1}:Sensor_f2)
12    transpose( Sensor_f, Sensor_f2 )
13
14    #pragma omp taskmoldable batch_count(gemm_0_count)\
15        depend(in:Sensor_f2,Dephase_x) depend(out:Pseudo_beam)\
16        access(strided{stride_S},Sensor_f2)\
17        access(strided{stride_X},Dephase_x)\
18        access(strided{stride_P},Pseudo_beam)
19    gemmStrideBatch( Sensor_f2, Dephase_x, Pseudo_beam, gemm_0_count )
20
21    #pragma omp taskmoldable batch_count(gemm_1_count)\
22        depend(in:Pseudo_beam,Dephase_y) depend(Beam)\
23        access(strided{stride_P}:Pseudo_beam)\
24        access(strided{stride_Y}:Dephase_y)\
25        access(strided{stride_B}:Beam)
26    gemmStrideBatch( Pseudo_beam, Dephase_y, Beam, gemm_1_count )
27
28    #pragma omp taskmoldable batch_count(abs_count)\
29        depend(in:Beam) depend(out:Energy)\
30        access(strided{1}:Beam,Energy)
31    abs( Beam, Energy, abs_count )
32  }
```

Code 4.14 – Parallélisation de l'algorithme de formation de voies avec des tâches moldables

La parallélisation se fait en deux phases : premièrement, le passage du domaine temporel au domaine fréquentiel se fait avec un batch de FFT, on exécute autant de FFT qu'il y a de capteurs. Ensuite, les données sont réordonnées pour être exploitables par les fonctions suivantes. Pour ces deux premières opérations, la dimension de parallélisation est la dimension des capteurs, nommée `fft_count` dans le code. La dimension `fft_size` correspond au nombre d'éléments de chaque FFT et donc au nombre maximum de fréquences observées.

La seconde phase consiste à calculer les déphasages et l'énergie sur chaque voie observée. Ces opérations sont effectuées à l'aide de deux batchs de multiplications de matrices ainsi qu'un batch de calcul des valeurs absolues. Pour ces calculs, les opérations sont indépendantes en fonction de la dimension des fréquences.

Entre ces deux phases, on a donc une dépendance all-to-all, qui peut être remplacée par un schéma de type *fork-join*.

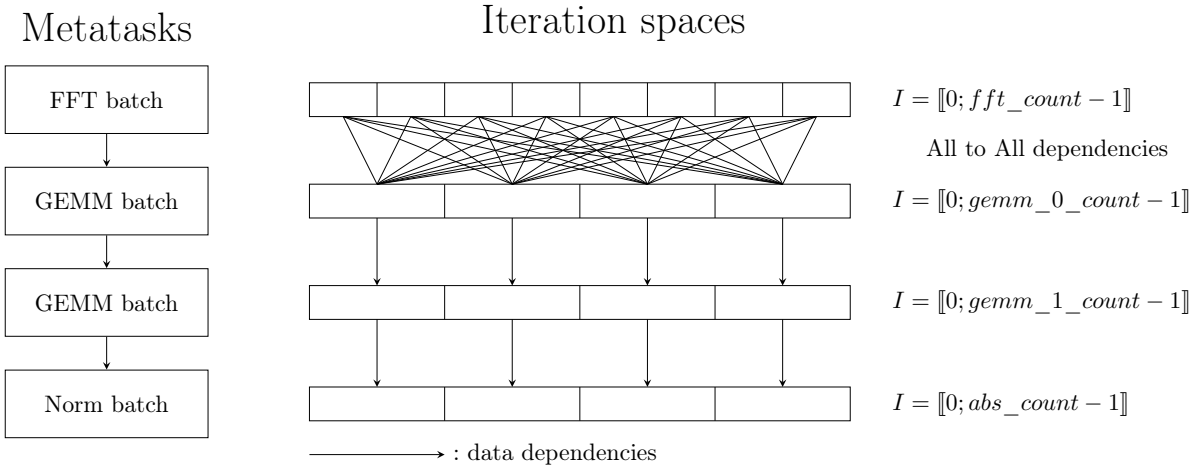


FIGURE 4.6 – Graphe de dépendances pour la formation de voies

La figure 4.6 représente les méta-tâches, l'espace d'itération associé et une découpe possible avec les dépendances associées.

4.7 Conclusion du chapitre

Dans ce chapitre, une méthode pour exprimer la moldabilité des tâches dans un contexte OpenMP a été proposée. L'intérêt principal de la directive `taskmoldable` est qu'elle permet d'exprimer un parallélisme implicite dans des fonctions non accessibles par l'utilisateur. On peut la voir comme une extension de la directive `taskloop` où le parallélisme de boucle n'est pas exposé à l'utilisateur. Cela permet, au sein d'une tâche, de conserver les optimisations internes réalisées par les développeurs des bibliothèques de calcul tout en autorisant la découpe du problème en fonction des ressources disponibles. En contrepartie, l'utilisateur doit fournir des informations détaillées sur les patterns d'accès aux données, permettant ainsi de déterminer les arguments d'entrée de chaque tâche et de définir les dépendances liées aux accès mémoires. Cependant, le standard OpenMP et les runtimes associés présentent des limitations qui peuvent affecter les performances d'un tel modèle de tâches. En particulier, la moldabilité des tâches implique que l'ensemble des accès mémoires possibles est très important. La gestion des dépendances en définissant des régions mémoires plutôt que des points précis serait probablement plus facile à appréhender et permettrait au runtime de gérer beaucoup moins de dépendances par tâche. Bien que la définition des régions mémoires soit incluse dans le standard OpenMP, la détection des recouvrements lors du calcul de dépendance entre tâches ne l'est pas. Certains runtimes expérimentaux ont intégré une gestion plus poussée des régions mémoires, tels que OmpSs [12].

Dans le chapitre 5 suivant, nous présentons la conception et l'implémentation d'un support exécutif pour des tâches moldables dont les dépendances peuvent s'exprimer sur des régions mémoires et dont l'algorithme de détection associé est rapide. Ce chapitre se terminera par un ensemble d'expérimentations sur les algorithmes que nous venons de voir.

Chapitre 5

Runtime d'ordonnancement de tâches moldables

5.1 Introduction

Les modifications des codes de traitement du signal présentées dans le chapitre 3 ont montré l'intérêt d'augmenter le grain des tâches pour exploiter efficacement les GPUs. En revanche, ces modifications limitent le parallélisme du code et augmentent les besoins en mémoire des co-processeurs qui vont exécuter ces tâches. Pour pallier ces défauts, une méthode classique serait de définir, manuellement ou automatiquement, une taille de grain qui permet plusieurs points :

1. avoir des tâches dont les données tiennent dans les mémoires des co-processeurs,
2. exprimer suffisamment de parallélisme pour exploiter la plateforme,
3. être de taille suffisante pour saturer les co-processeurs de type GPU.

Décider de la granularité des tâches est loin d'être trivial, les points 1 et 2 nécessitent de réduire la taille des tâches tandis que le point 3 nécessite de l'augmenter. De plus, dans le cas d'une plateforme hétérogène, les tailles optimales pour saturer les unités de calculs peuvent varier en fonction des ressources utilisées.

Dans le chapitre 4, nous avons proposé de résoudre ce problème en utilisant le concept de tâches moldables et nous avons introduit une syntaxe pour exprimer la moldabilité des tâches dans OpenMP. Les tâches moldables offrent une gestion dynamique de la granularité des tâches. Cependant, la syntaxe proposée restreint les fonctions qui peuvent être des homomorphismes de listes par rapport aux tâches hiérarchiques utilisées, par exemple, dans StarPU.

Dans ce chapitre, nous introduisons un environnement d'exécution pour la gestion des tâches moldables. Nous avons décidé d'implémenter un runtime spécifique afin de réaliser nos expérimentations plutôt que de nous baser sur un runtime existant. La suite de ce chapitre présente le modèle de tâche utilisé en section 5.3, les choix d'ordonnancement en section 5.4, les détails sur l'implémentation en section 5.5 et enfin des benchmarks seront présentés en section 5.6.

5.2 Positionnement

Plusieurs méthodes ont été explorées dans la littérature pour gérer la granularité : Cilk [26] propose de générer des tâches de manière récursive, qui sont réparties avec un système de vol de tâches : les tâches de plus haut niveau, a priori de plus gros grain, dans l'arbre d'appel sont volées en priorité. Ce modèle ne dispose pas de notions de dépendances et est fortement dépendant des architectures à mémoire partagée des années 90.

Athapascan-1 [28] étend le modèle de Cilk en considérant des dépendances entre tâches afin de gérer simplement une exécution sur architecture à mémoire distribuée sans considérer les architectures hétérogènes qui n'ont été considérées que par ses successeurs Kaapi [29], devenu XKaapi [31]. Les travaux liés

à Kaapi [63], [64], [56] ont proposé un modèle de programmation pour des algorithmes à grain adaptatif où, à l'exécution, le choix a été fait entre une exécution séquentielle ou une exécution parallèle par la génération d'un graphe de tâches, la génération de ce graphe ne se faisant qu'en fonction de l'inactivité des ressources lors des requêtes de vol [63]. Mais ces derniers travaux n'ont pas considéré les architectures hétérogènes.

Plus récemment, d'autres méthodes ont été introduites, en particulier adaptées aux architectures hétérogènes. StarPU a introduit des tâches parallèles [16], les tâches sont regroupées en paquets destinés aux GPU ou aux groupes de CPUs. Une autre solution a été de proposer des tâches hiérarchiques capables de générer à l'exécution un sous-graphe de tâches [65], avec le défaut de proposer une découpe statique. Des travaux récents ont ajouté ce modèle à StarPU [23], permettant une découpe dynamique des tâches hiérarchiques en fonction des ressources disponibles.

Le choix d'implémentation d'un runtime d'ordonnancement de tâches spécifique à cette thèse est justifié par plusieurs points.
Premièrement, la modification d'un runtime existant aurait nécessité de toucher à des points centraux du runtime tel que la définition des tâches ou la gestion des zones mémoires utilisées pour le calcul des dépendances. De ce fait, modifier un runtime OpenMP, tel que celui de GCC, LLVM, ou OmpSs, aurait introduit une complexité non nécessaire à la preuve de concept que nous cherchons à développer. Cela aurait été également le cas pour une intégration à StarPU ou XKaapi.

Une deuxième contrainte forte est l'aspect industriel de la thèse, qui peut imposer d'une part des limitations sur les propriétés intellectuelles du code produit et d'autre part la volonté pour l'entreprise de maîtriser les sources du développement.

La décision s'est donc portée sur une implémentation complète d'un runtime, sous forme d'un prototype, pour la gestion de l'exécution d'un programme à base de tâches moldables. Nous développerons dans les perspectives une suite souhaitable de ces travaux de thèse qui serait d'intégrer la gestion des tâches moldables dans un runtime durable, tel qu'un OpenMP-LLVM, en se basant sur les pragmas présentés au chapitre précédent.

5.3 Représentation des tâches

Une tâche moldable peut être définie par le triplet suivant : $T = (F, A, I)$ où $A = \{A^j\}_j$ est un ensemble de régions mémoires, $I = [\![a, b]\!]$ un intervalle d'itération et F la fonction exécutée par la tâche moldable. F est une fonction qui travaille sur $|A|$ régions mémoires et un intervalle d'itération : $F(I, A^0, A^1, ..., A^{|A|-1})$.

L'analogie avec la syntaxe présentée en chapitre 4 est directe, F correspond au bloc structuré impacté par la directive `taskmoldable`, A^j aux données ciblées par la clause `access` et I aux informations données par la clause `batch_count`. Dans ce chapitre, on se limitera à une implémentation manipulant un espace d'itération mono-dimensionnel. Ce choix est effectué pour simplifier l'implémentation et car les algorithmes cibles de traitement du signal peuvent se limiter à ce type d'espace d'itération pour exprimer du parallélisme.

5.3.1 Découpe de la tâche

Du fait des tâches moldables considérées, on peut découper l'intervalle d'itération I en union d'intervalles deux à deux disjoints $\{I_k\}_k$. Si F est un homomorphisme de liste sur les éléments de I alors l'exécution de la tâche T se réduit à l'exécution des sous-tâches $T_k = (F, A_k, I_k)$. On dit qu'une sous-tâche T_k est valide si I_k est un intervalle d'itération non nul, continu et $I_k \in I$. De même on considère qu'une découpe de n tâches est valide si chaque tâche travaille sur des sous-ensembles I_k distincts deux à deux et que l'union des sous-ensembles soit égale à I :

$$\forall i, j \in [\![0, n-1]\!]^2, i \neq j, I_i \cap I_j = \emptyset$$

$$\bigcup_{k=0}^{n-1} I_k = I$$

Une tâche peut être découpée en un maximum de $|I|$ sous-tâches (une itération par tâche), les méthodes de découpe seront détaillées dans la section 5.4.

5.3.2 Espace mémoire accédé par les tâches

Chaque sous-tâche doit accéder à des données différentes, pour une sous-tâche T_k, nous définissons $A_k = A_k^0, A_k^1, ..., A_k^{|A|-1}$ l'ensemble des régions mémoires accédées par la sous-tâche. Chaque sous-tâche accède à une partie de chacune des $|A|$ régions de A.

Une difficulté est de représenter des accès mémoires complexes et non continus, en particulier certaines données peuvent être stockées de manière entrelacée.

Dans le cas où nous voulons représenter des patterns mémoires non continus, plusieurs méthodes sont disponibles dans la littérature pour représenter la mémoire et en particulier détecter les dépendances qui en découlent. Par exemple, OmpSs dispose d'une vision en tableau multi-dimensionnel [12] permettant une détection des collisions en $\mathcal{O}(d)$ avec d le nombre de dimensions. Nous nous sommes inspirés des méthodes d'accès linéaire présentées par Paek [51], en restreignant les cas d'utilisation pour conserver une détection des collisions rapide. La représentation linéaire a l'élégance d'utiliser l'adressage virtuel du programme, ce qui permet des comparaisons d'accès mémoires indépendamment des manipulations faites sur les pointeurs par l'utilisateur.

Les algorithmes que nous traitons se basent principalement sur des opérations matricielles et manipulent des groupes de matrices. Pour cela, nous souhaitons pouvoir représenter les schémas de données principaux manipulés par les bibliothèques BLAS. Dans le cas d'un stockage des matrices en column-major, la fonction gemm_stride_batched a besoin de 5 arguments pour définir le placement mémoire des données : les dimensions de la matrice m et n, la dimension principale ld qui représente la distance entre les premiers éléments de deux colonnes consécutives, le *stride* qui représente la distance entre les premiers éléments de deux matrices consécutives, et le nombre de matrices. Le schéma 5.1a représente ce placement de mémoire.

Pour représenter ces aspects mémoires, la notation suivante est proposée :

— `ws` : `work segment` : Le nombre de segments mémoires consécutifs accédés par une tâche élémentaire.
— `es` : `element size` : La taille de chacun de ces éléments (en octet).
— `ej` : `element jump` : La distance entre le début de deux segments consécutifs (en octets).
— `ss` : `stride size` : La distance entre les premiers segments de deux itérations consécutives de l'espace d'itération (en octets).
— Un pointeur p qui représente le début du premier segment.

On peut constater un mapping avec les arguments présentés pour gemm_stride_batched avec : `ws` équivalent à n, `es` équivalent à $m*sizeof(type)$, `ej` à lda et `ss` à $stride$. Une comparaison avec la notation précédente est présentée en figure 5.1.

Cette notation est redondante avec les notations BLAS par design. Néanmoins, le choix a été d'introduire ces valeurs pour représenter les cas avec un nombre de dimensions supérieur à 1.

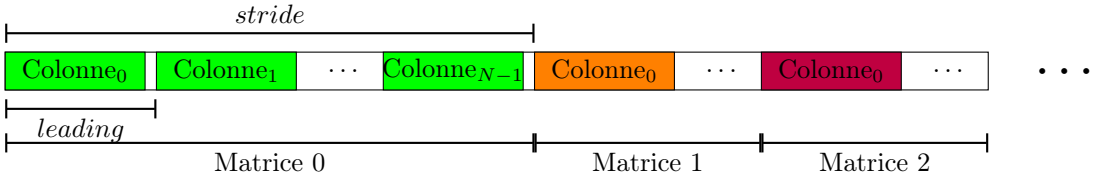
5.3.3 Intersection de deux espaces mémoires

En plus de la gestion des mouvements mémoires, connaître la région mémoire accédée par une tâche permet de définir plus finement les dépendances entre les tâches, en particulier lorsqu'elles sont liées aux accès mémoires.

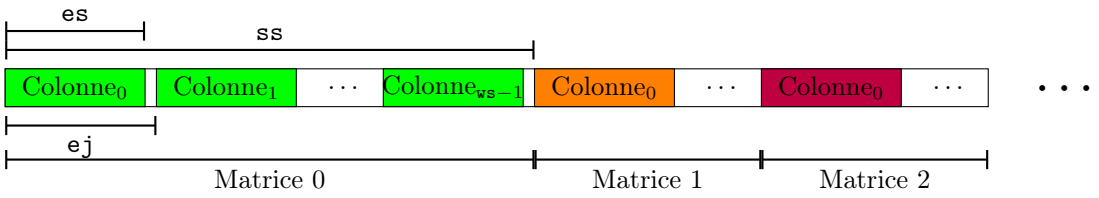
Dans le cas où une tâche calcule l'intervalle I_k, ses accès mémoire sont $A_k = A_k^0, A_k^1, ..., A_k^{|A|-1}$. Par exemple dans le cas d'un batch gemm_stride_batched ils sont représentés à la figure 5.1.

Nous allons nous restreindre au cas où la tâche accède à la mémoire sous forme d'un ensemble de segments équidistants : $\Gamma = \bigcup_{i=0}^{N-1} \Gamma_i$, avec $\Gamma_i = [a \cdot i + b; a \cdot i + b + c]$.

On peut définir Γ dans les trois cas suivants :



(a) Placement mémoire pour un GEMM batch



(b) Placement mémoire du runtime

FIGURE 5.1 – Comparaisons des représentations mémoires dans le cas d'un appel à `gemm_stride_batched`

$\texttt{es} = \texttt{ej}$	$\texttt{es} = \texttt{ss}$	$\texttt{ss} = \texttt{ws} \times \texttt{ej}$
$a = \texttt{ss}$	$a = \texttt{ej}$	$a = \texttt{ej}$
$b = p + I_k[0] \times \texttt{ss}$	$b = p + I_k[0] \times \texttt{ss}$	$b = p + I_k[0] \times \texttt{ss}$
$c = \texttt{es} \times \texttt{ws}$	$c = \texttt{es} \times \lvert I_k \rvert$	$c = \texttt{es}$
$N = \lvert I_k \rvert$	$N = \texttt{ws}$	$N = \lvert I_k \rvert \times \texttt{ws}$

Les autres cas peuvent être représentés par une union d'ensemble de segments $\bigcup_{j=0}^{M-1} \bigcup_{i=0}^{N-1} \Gamma_{i,j}$ avec $\Gamma_{i,j} = [a \cdot i + (b + d \cdot j); a \cdot i + (b + d \cdot j) + c]$ avec $a = \texttt{ej}$, $b = p + I_k[0] \times \texttt{ss}$, $c = \texttt{es}$, $N = \texttt{ws}$, $d = \texttt{ss}$, $M = |I_k|$.

Ces cas particuliers peuvent arriver, par exemple quand on manipule des matrices disposant d'un padding. Ces cas ne sont pas présents dans notre base de code.

La dépendance entre deux tâches peut alors être détectée en calculant s'il y a un recouvrement entre deux ensembles de segments $\Gamma = \bigcup_{i=0}^{N-1} \Gamma_i$ et $\Gamma' = \bigcup_{i=0}^{N'-1} \Gamma'_i$ associés aux deux tâches. Il y a une intersection entre deux régions s'il existe deux sous-régions Γ_i et Γ'_j qui se recouvrent. L'algorithme 5.1 détecte une intersection en temps constant entre deux segments.

```
1  func detect_collision(Γ0,Γ1):
2          if( Γ1.b < Γ0.b ):
3                  return detect_collision( Γ1, Γ0 );
4          a0 = Γ0.a;
5          b0 = Γ0.b;
6          c0 = Γ0.c;
7          N0 = Γ0.N;
8
9          a1 = Γ1.a;
10         b1 = Γ1.b;
11         c1 = Γ1.c;
12         N1 = Γ1.N;
13
14         b = b1 - b0;
15         delta = a1 - floor(((float) a1)/a0)*a0;
16         gamma = delta - a0;
17         x0 = b%a0;
18
19         if( delta == 0 ):
20                 j0 = b/a0;
21                 if( (x0 < c0 && j0 < N0)  (x0 > a0 - c1 && j0 + 1 < N0) ):
```

```
22                          return 1;
23                  else:
24                          return 0;
25          else if( x0 < a0 - delta ):
26          { // x1 = x0 + delta, assume xi = x0 + i*delta
27                  ic = ceil( ((float) (a0 - c1 - x0))/delta );
28          }
29          else:
30          { // x1 = x0 + gamma, assume xi = x0 + i*gamma
31                  ic = ceil( ((float) c0-x0)/gamma );
32          }
33
34          jc = (a1*ic+b)/a0;
35          if( ic < N1 && jc < N0 ):
36                  return 1;
37          else:
38                  return 0;
```

Code 5.1 – Algorithme de détection de collisions

La suite de cette section va donner l'intuition derrière cet algorithme : on commence par expliquer une méthode pour détecter les dépendances. On propose et on justifie une approximation qui évite d'effectuer des calculs dans $\mathbb{Z}/n\mathbb{Z}$ sans générer de faux-négatifs dans les détections de dépendances. Finalement, on définit le lien entre cette méthode et le code présenté en 5.1.

Détection des dépendances. Pour illustrer le fonctionnement de l'algorithme, on suppose que Γ^0 commence avant Γ^1, donc $b^0 < b^1$. On définit $b = b^1 - b^0$ et

$$x_i = (a^1 \times i + b) \mod a^0$$

x_i est la position du $i^{\text{ème}}$ segment de Γ^1 dans l'espace $\mathbb{Z}/a^0\mathbb{Z}$ dont la position 0 coïncide avec le début du premier segment de Γ^0. L'indice du segment de Γ^0 est noté $j_i = \lceil \frac{a^1 \times i + b^1}{a^0} \rceil$.

On sait donc que le $i^{\text{ème}}$ segment de Γ^1 commence après le $j_i^{\text{ème}}$ segment de Γ^0 et avant le $(j_i + 1)^{\text{ème}}$ segment.

On a une collision entre Γ^0 et Γ^1 si et seulement si :

$$\exists i \geq 0, x_i < c^0 \text{ et } j_i < N^0 \text{ et } i < N^1$$

ou

$$\exists i \geq 0, x_i > a^0 - c^1 \text{ et } j_i + 1 < N^0 \text{ et } i < N^1$$

Dans le premier cas, le segment i est en collision avec le segment j_i car j_i termine avant i ($x_i < c^0$). Dans le second cas, le segment $j_i + 1$ commence avant la fin du segment i ($x_i > a^0 - c^1$). Les conditions sur N^0 et N^1 vérifient que les segments font bien partie de Γ^0 et Γ^1.

Trouver la première valeur de i remplissant une de ces conditions revient à résoudre une équation dans $\mathbb{Z}/a^0\mathbb{Z}$.

Approximation de la détection. Pour éviter d'avoir à effectuer cette opération, on décide d'approximer le résultat et d'accepter des faux positifs.

Premièrement, on peut noter que $x_{i+1} = (x_i + a^1) \mod a^0$. Si $x_{i+1} \geq x_i$, on peut écrire :

$$x_{i+1} = x_i + \delta \text{ avec } \delta = a^1 - \lceil \frac{a^1}{a^0} \rceil a^0 \geq 0$$

sinon on peut écrire :

$$x_{i+1} = x_i + \gamma \text{ avec } \gamma = a^1 - \lceil \frac{a^1}{a^0} \rceil a^0 - a^0 = \delta - a^0 < 0$$

La suite des x_i est alors générée par l'ajout de δ ou de γ à $x_0 = b \mod a^0$.

Notre approximation consiste à supposer que la suite est composée uniquement de δ ou de γ. De plus, $x_1 > 0$ si $x_0 + \delta < a$, on supposera donc :

Si $x_0 < a - \delta$:

$$x_i = x_0 + \delta \times i$$

Si $x_0 \geq a - \delta$:

$$x_i = x_0 + \gamma \times i$$

L'objectif est alors de trouver i_c, la première valeur de i pour laquelle on a une collision (en supposant le nombre de segment de Γ^0 et Γ^1 infinis).

Comme la suite x_i est croissante, la collision aura lieu lorsque $x_{i_c} \geq a^0 - c^1$. Donc :

$$\begin{aligned} x_0 + \delta i_c &\geq a^0 - c^1 \\ i_c &\geq \frac{a^0 - c^1 - x_0}{\delta} = \frac{a^0 - c^1 - (b \mod a^0)}{\delta} \\ i_c &= \lceil \frac{a^0 - c^1 - (b \mod a^0)}{\delta} \rceil \end{aligned}$$

Comme la suite x_i est décroissante, la collision aura lieu lorsque $x_{i_c} \leq c^0$. Donc :

$$\begin{aligned} x_0 + \gamma i_c &\leq c^0 \\ i_c &\geq \frac{c^0 - x_0}{\gamma} = \frac{c^0 - (b \mod a^0)}{\gamma} \\ i_c &= \lceil \frac{c^0 - (b \mod a^0)}{\gamma} \rceil \end{aligned}$$

Cas particulier : lorsque $\delta = 0$, dans ce cas, $a^0 = a^1$. Il y a une collision si et seulement si x_0 est en collision avec j_0 ou $j_0 + 1$.

Justification. Pour justifier que cette méthode évite les faux-négatifs, on va montrer que $\forall i < i_c$, x_i utilisé par notre approximation est valide.

Premièrement, on rappelle que :

$$\begin{aligned} x_i &= (a^1 \times i + b) \mod a^0 \\ &= (((a^1 \times i) \mod a^0) + (b \mod a^0)) \mod a^0 \\ &= (\delta \times i + x_0) \mod a^0 \end{aligned}$$

Si $x_0 < a^0 - \delta$, alors par définition de i_c, $x_i < a^0 - c^1 < a^0, \forall i < i_c$, donc on ne manque pas de collisions.

De même, si $x_0 \geq a^0 - \delta$, les x_i seront supérieurs à 0 tant que $i < i_c$.

Sur-détection. On peut montrer, par exemple, que cet algorithme peut générer des dépendances superflues.

Prennons :

— $\Gamma^0 = \bigcup_{i=0}^{3} [10i; 10i + 1]$ avec (N = 3, a = 10, b = 0, c = 1).

— $\Gamma^1 = \bigcup_{i=0}^{4} [4i + 4; 4i + 4 + 1]$ avec (N = 4, a = 4, b = 4, c = 1).

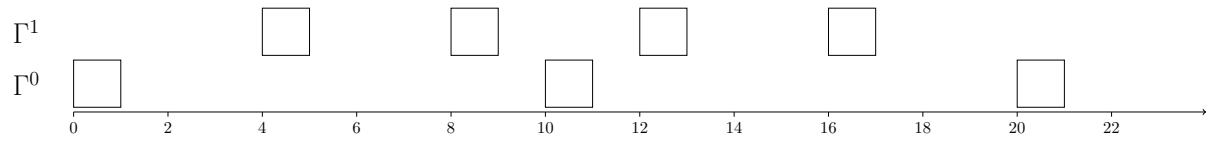


FIGURE 5.2 – Exemple de sur-détection

On obtient donc une valeur de $\delta = 4$ et une détection pour $x_i = 3$ alors qu'aucun segment de Γ^0 n'intersecte Γ^1.

Lien avec le code. La fonction 5.1 est le pseudo-code implémentant cette méthode, les lignes 2 à 12 initialisent les variables et assurent que Γ^0 commence avant Γ^1. Les valeurs de δ, γ et $x0$ sont ensuite calculées (lignes 15 à 17).

Ensuite, la condition, ligne 19, gère le cas où δ est nul.

La valeur de i_c est calculée en fonction de la position de x_0 en ligne 25 ou 29.

Enfin, j_c est calculé en ligne 34 et on vérifie que les segments concernés soient inclus dans Γ^0 et Γ^1.

5.4 Ordonnancement

5.4.1 Découpe des tâches

L'objectif de la découpe des tâches est double pour le runtime. Premièrement, il s'agit de répartir la charge de travail entre les workers de manière à équilibrer le temps d'exécution de chacun des workers, pour cela nous utilisons une estimation des performances détaillées en sous-section 5.4.2. Deuxièmement, les tâches générées doivent tenir dans la mémoire du worker qui doit les exécuter.

Lors de la création des tâches, le runtime a connaissance de la granularité maximale de découpe via la taille de l'espace d'itération $|I|$. Sont aussi à disposition des informations sur le pattern d'accès des données, elles permettent entre autre le calcul de l'espace mémoire nécessaire pour exécuter une tâche en fonction du nombre d'itérations traitées de l'espace I. En pratique, cette taille est de la forme d'une fonction affine $a \times |I_k| + b$, avec b l'espace mémoire qui n'évolue pas en fonction du nombre d'itérations traitées et a l'espace mémoire supplémentaire nécessaire pour chaque itération. Connaissant l'espace mémoire disponible sur un worker, on peut alors décider de la taille maximum de la tâche exécutable sur ce même worker.

Pour un runtime dynamique, il y a plusieurs méthodes pour définir quel thread va exécuter une tâche. La première est de prendre la décision lorsque la tâche est soumise par l'utilisateur, auquel cas l'ordonnanceur aurait un comportement proche d'un ordonnanceur statique basé sur un algorithme glouton. Une seconde méthode est de calculer seulement les dépendances à la soumission des tâches et d'attribuer une tâche à un thread lorsque celle-ci est prête à être exécutée. Une troisième méthode est de laisser les threads sélectionner la prochaine tâche prête à être exécutée lorsqu'ils n'ont plus de tâches en exécution [11].

Le choix de la taille de chacune des tâches est liée aux workers, la découpe est donc équivalente à définir quel worker va exécuter quelle tâche. De plus, le calcul des dépendances nécessite de connaître la découpe. Nous avons donc choisi de découper les tâches à la soumission par l'utilisateur.

Les tâches sont découpées en 2 étapes : premièrement, chaque worker se voit attribuer une sous-tâche. La taille de ces sous-tâches est spécifiée dans la section 5.4.2. Une seconde passe permet de re-découper les tâches attribuées aux co-processeurs afin que chacune d'entre elles puisse respecter les contraintes mémoires.

5.4.2 Estimation des performances

La première étape de la découpe d'une tâche permet de répartir la charge de travail entre les workers. Le runtime utilise un modèle de performance pour estimer le temps d'exécution d'une tâche par un worker. Les performances sont mesurées à l'exécution et utilisées aux itérations suivantes pour affiner la répartition du travail. Cette méthode est inspirée de l'approche de StarPU [3], néanmoins, notre runtime ne conserve pas les informations de performances d'une exécution à l'autre.

Les algorithmes applicatifs ciblés travaillent sur des flux de données et répètent donc de nombreuses fois les mêmes chaînes de traitement. On applique donc une heuristique basique : le principe de persistance [39], qui suppose que le comportement, donc les performances, d'une itération seront proches de ceux des dernières itérations.

L'objectif de la découpe est que tous les workers travaillent le même temps pour chacune des tâches. En supposant que la relation entre le temps de travail et la taille de l'espace d'itération I_k traité par un worker est linéaire, on peut définir le temps de travail du worker w : $t_{w,k} = P_w \times |I_k|$ avec P_w la performance estimée du worker pour cette tâche.

À la découpe, l'ordonnanceur va fonctionner en 2 phases : premièrement estimer la valeur de P_w en fonction des temps d'exécution précédents. Ensuite définir I_k pour chaque worker pour minimiser le plus long temps d'exécution des workers.

Pour la suite, on va noter : $t^i_{w,k}$ le temps d'exécution du worker w pour la $k^{\text{ème}}$ sous-tâche de l'itération i, p^i_w la performance estimée du worker w en fonction des itérations $[\![0, i-1]\!]$. La première fois que le runtime rencontre une tâche, il définit un poids équivalent pour chaque worker :

$$p^0_w = \frac{1}{n}$$

Avec n désignant le nombre de workers.

Il va utiliser le temps d'exécution global pour calculer q^{i+1}_w, le poids qu'aurait dû avoir le worker pour que sa durée de travail soit égale à la moyenne des temps de travail de chacun des workers.

$$q^{i+1}_w = \frac{p^i_w}{t^i_w} \times \frac{\sum_{k=0}^{n-1} t^i_k}{n}$$

Et finalement, on calcule le poids pour l'itération suivante en effectuant une normalisation :

$$p^{i+1}_w = \frac{q^{i+1}_w}{\sum_{k=0}^{n-1} q^{i+1}_k}$$

De ce fait, chaque worker va diminuer son temps de travail s'il est plus long que la moyenne et l'augmenter sinon. Les temps d'exécutions $t^i_{w,k}$ convergent tous vers la même valeur.

Le choix de pousser tous les workers à avoir le même temps d'exécution est justifié par le théorème suivant :

Théorème 1. *Le temps d'exécution minimal d'une tâche moldable est atteint si et seulement si tous les workers ont le même temps d'exécution.*

Démonstration. Par l'absurde : soit un ordonnacement avec $n > 1$ workers et une découpe optimale $R = (r_0, r_1, ..., r_{n-1})$, avec :

$$r_i \in [0, 1]$$
$$\sum_{i=0}^{n-1} r_i = 1$$

où r_i est la fraction de travail effectuée par le worker i.

Supposons que les temps d'exécution ne soient pas égaux. Alors il existe un ensemble de $a > 0$ workers $[x_0, x_1, ...x_{a-1}]$ tel que :

$$t_{x_i} = t_{min} = min(t_0, t_1, ..., t_n)$$

et un ensemble de $b > 0$ workers $[y_0, y_1, ..., y_{b-1}]$ tel que :

$$t_{y_i} = t_{max} = max(t_0, t_1, ..., t_n)$$

Soit R', une nouvelle découpe telle que :

$$r'_{x_0} = r_{x_0} + dr$$
$$r'_{y_i} = r_{y_i} - \frac{dr}{b}, \forall i \in [\![0, b-1]\!]$$
$$r'_w = r_w, \forall w \neq x_0, w \notin [\![0, b-1]\!]$$

avec $dr = \frac{1}{2 * P_{x_0}}(t_{max} - t_{min})$.

La découpe R' est valide car :

$$\begin{aligned}\sum_{i=0}^{n-1} r'_i &= \sum_{i=0}^{n-1} r_i + (\sum_{j=0}^{b-1} r'_{y_j} - \sum_{j=0}^{b-1} r_{y_j}) + (r'_{x_0} - r_{x_0}) \\ &= 1 + \sum_{j=0}^{b-1} r'_{y_j} - r_{y_j} + (r'_{x_0} - r_{x_0}) \\ &= 1 + \sum_{j=0}^{b-1} (-\frac{dr}{b}) + dr = 1\end{aligned}$$

Le temps d'exécution de chacun des worker est :

$$\begin{aligned}t'_{x_0} &= P_{x_0} \times r'_{x_0} = P_{x_0} \times r_{x_0} + P_{x_0} \times dr \\ &= t_{min} + \frac{1}{2}(t_{max} - t_{min}) < t_{max} \\ t'_{y_i} &= P_{y_i} \times r'_{y_i} = P_{y_i} \times r_{y_i} - P_{y_i} \times \frac{dr}{b} \\ &= t_{y_i} - P_{y_i} \times \frac{dr}{b} < t_{max} \\ t'_w &= t_w < t_{max}\end{aligned}$$

Donc $t'_{max} = max(t'_0, t'_1, ..., t'_{n-1}) < t_{max}$, donc la découpe R' est meilleurs que la découpe R.

Or R était supposée optimale, donc c'est absurde.

□

Converger vers un temps d'exécution identique pour chacun des workers est optimal lorsque le runtime exécute des tâches moldables de manière séquentielle. On notera que ce n'est plus le cas dès que l'on exécute des tâches moldables en parallèle.

5.5 Implémentation

Cette section présente les détails d'implémentation du runtime, le choix a été de l'implémenter sous forme d'une bibliothèque C. L'organisation générale des threads sera détaillée en sous-section 5.5.1, l'API utilisateur en sous-section 5.5.2, les principales structures de données internes en sous-section 5.5.3 et finalement les détails des threads workers et du monitoring en sous-sections 5.5.4 et 5.5.6.

5.5.1 Vue d'ensemble du runtime

À l'exécution du runtime, le code utilisateur génère séquentiellement des tâches moldables appelées `metatasks`. Elles sont découpées en tâches et les dépendances sont calculées par le thread utilisateur. Les tâches ayant toutes leurs dépendances satisfaites sont empilées dans les piles d'exécution des workers puis exécutées par les threads workers. Un schéma de ce fonctionnement est disponible en figure 5.3.

Le monitoring est effectué directement par les threads workers et les données sont compilées pour mettre à jour les valeurs de p_w^i lorsque toutes les tâches d'une `metatask` sont terminées.

5.5.2 API

L'API utilisateur comporte un ensemble limité de fonctions. Les fonctions basiques de management sont : une fonction d'initialisation `init_scheduler` permet de définir le nombre de workers total, et le nombre de workers associés à un GPU. Une fonction pour nettoyer la mémoire et les threads alloués par le runtime `destroy_scheduler`. `register_function` qui permet de préparer les fonctions qui seront utilisées par les tâches.

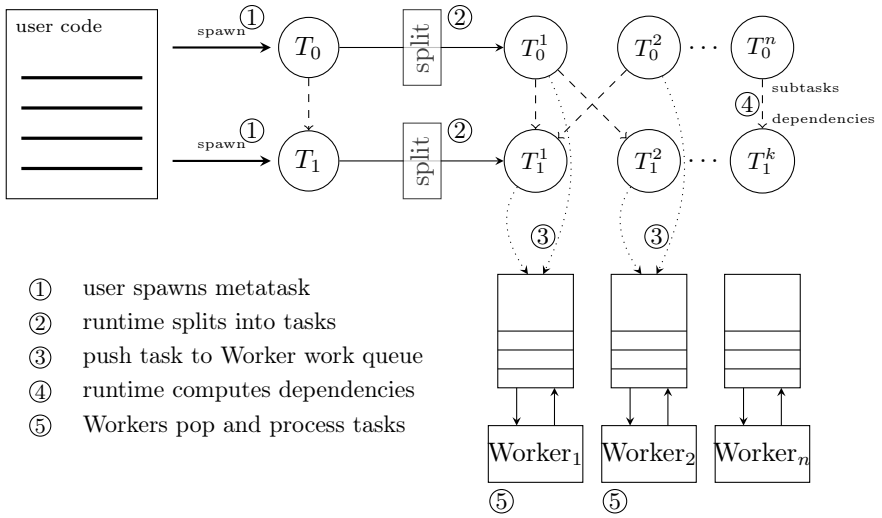


FIGURE 5.3 – Exemple d'exécution du runtime

Les fonctions `sched_malloc` et `sched_free` permettent d'allouer des tableaux mémoires utilisables par le runtime. En pratique, ils sont alloués en mémoire page-lock avec `cudaMallocHost` pour permettre des transferts asynchrones avec les GPUs. La gestion de la mémoire est également effectuée par les fonctions `ignore_mem` et `clear_mem`, qui indiquent que le tableau ne sera plus utilisé ultérieurement. La fonction `clear_mem` rapatrie toutes les données présentes sur des coprocesseurs en mémoire CPU.

La fonction `spawn_function` permet de générer une tâche moldable `metatask` de manière asynchrone, une fonction utilitaire `create_mem_info` permet d'allouer la structure de données qui permet à l'utilisateur de préciser le pattern d'accès aux données détaillé en sous-section 5.3.2.

Deux fonctions de synchronisation `scheduler_sync` et `scheduler_sync_on_array` permettent respectivement de synchroniser le thread utilisateur avec l'ensemble des tâches en cours ou avec l'ensemble des tâches travaillant sur un tableau mémoire donné.

Lors d'un appel à la fonction `spawn_function`, l'utilisateur doit fournir un identifiant de fonction obtenu via `register_function`. Cet identifiant permet de lier les tâches générées à une fonction pour chacun des types de matériel disponibles (cœur CPU et GPU Nvidia). Ces fonctions ont le type défini en code 5.2.

```
void function(void* ctx, void* common_args, struct sched_mem_info_t memory_args);
```

Code 5.2 – Déclarations des fonctions des workers

Le premier argument, `ctx`, est un contexte qui sera fourni par le thread exécutant la fonction. En fonction du matériel cible, il aura l'un des deux types suivants :

```
struct cpu_ctx_t {
    uint32_t worker_id;
    uint32_t hardware_id;
    uint32_t core_id;
};
```

```
struct cuda_ctx_t{
        uint32_t worker_id;
        uint32_t hardware_id;
        uint32_t device_id;
        cudaStream_t   wstream;
        cublasHandle_t cublas_handle;
        cudaStream_t   htdstream;
        cudaStream_t   dthstream;
};
```

L'argument `common_args` est un pointeur fourni par l'utilisateur lors de l'appel à `spawn_function` qui permet d'envoyer des arguments communs à toutes les tâches générées par la tâche moldable. Finalement, l'argument `memory_args` contient les pointeurs d'accès aux données et le schéma de stockage.

La fonction `spawn_function` est déclarée de la manière suivante :

```
sched_error_t spawn_function(
        unsigned int function,
        void* common_args, size_t args_size,
        struct sched_mem_info_t* memory_args, sched_perf_info_t* perf_tracker,
        int should_record );
```

Le premier argument `function` est un identifiant obtenu par `register_function`. L'argument `common_args` est le pointeur qui va être transmis aux tâches sous le même nom. `args_size` est sa taille en octets. `memory_args` est la structure qui permet de définir le placement des données.

```
struct sched_mem_info_t {
    uint32_t array_count;
    uint32_t dim_count; // = 1
    uint32_t allocated_array_count;
    uint32_t allocated_dim_count;

    struct sched_array_t** arrays;
    void** local_data_ptr;

    size_t*     es; // [a]
    uint64_t*   ws; // [a*d]
    size_t*     ej; // [a*d]
    size_t*     ss; // [a*d]
    enum sched_access_policy_t* flags; // [a] Access policy read/write
    uint64_t*   s;      // [a*d] First subproblem to deal with
    size_t*     shifts; // [a] Shift to first pointer
    uint64_t*   dims;   // [a*d] Number of subproblem to deal with in each dim
};
```

L'utilisateur doit remplir les valeurs des tableaux‘ `es`‘, ‘`ws`‘, ‘`ej`‘, ‘`ss`‘ comme défini en sous-section 5.3.2. De plus, la valeur ‘`arrays`‘ doit être renseignée avec des pointeurs alloués par la fonction ‘`sched_malloc`‘, la valeur ‘`flags`‘ indique si la fonction accède en lecture ou en écriture aux données. Les autres données sont pré-remplies à l'allocation de la structure avec la fonction ‘`create_mem_info`‘.

Enfin, l'argument `perf_tracker` permet de conserver les informations du monitoring. Si la même variable est utilisée pour plusieurs appels à la fonction `spawn_function`, le runtime utilisera le même modèle de performance pour toutes ces tâches.

5.5.3 Implémentation des fonctions

La sous-section précédente a présenté les fonctions et les structures de données nécessaires pour que l'utilisateur interagisse avec le runtime. Dans cette sous-section, nous allons voir les principales structures de données utilisées en interne et l'interaction entre les fonctions du runtime et ces structures. En section 5.4, nous avons précisé que les opérations de découpe des tâches et de génération des dépendances étaient faites de manière synchrone avec le thread utilisateur, il faut donc une implémentation minutieuse de ces fonctions pour limiter les surcoûts liés au runtime.

La règle principale de l'implémentation est de limiter les appels systèmes, en particulier les allocations mémoires. Pour ce faire, la majorité des structures fréquemment utilisées ne sont pas désallouées après utilisation mais enregistrées dans une liste de structures disponibles pour une future utilisation.

Un exemple avec la structure struct metatask_t :

```
struct metatask_t {
    uint32_t metatask_id;
    uint32_t action_id;

    uint32_t worker_count;

    uint32_t function_id;
```

```
    void* common_args;
    size_t common_args_size;
    size_t common_args_allocated_size;
    struct sched_mem_info_t memory_args;

    uint32_t task_count;
    uint32_t allocated_task_count;
    struct task_t* tasks;
    uint32_t completed_task_count;

    struct perf_info_t* profiling_info;
    uint32_t* worker_workload;
    uint32_t* worker_task_count;
    int       should_record;

    pthread_mutex_t metatask_mutex;
    sem_t* sync_sem;
};
```

Code 5.3 – Déclaration de la structure `struct metatask_t`

Lors d'un appel à la fonction `spawn_function`, une variable de type `struct metatask_t` est récupérée grâce à la fonction `get_free_metatask` : si aucune méta-tâche n'est allouée et inutilisée, elle en alloue une nouvelle. Lorsque toutes les tâches de la méta-tâche sont terminées, elle est replacée dans la liste des méta-tâches non utilisées au lieu d'être désallouée. De même, tous les tableaux de tailles variables internes ne sont jamais désalloués mais seulement réalloués lorsque qu'une nouvelle méta-tâche a besoin de plus de place.

La génération d'une tâche consiste principalement à découper la tâche moldable en sous-tâches. Le runtime doit savoir combien de mémoire est nécessaire pour exécuter une tâche. Pour cela, il calcule deux valeurs : une taille constante allouée quel que soit le nombre d'itérations traitées par la tâche cst_size et une taille supplémentaire pour chaque itération traitée var_size. La mémoire nécessaire pour exécuter une tâche est cst_size + var_size $\times$ $|I_k|$. Ces tailles sont la somme des tailles nécessaires pour chaque tableau manipulé. cst_size est incrémenté de `es` $\times$ `ws` lorsque `ss` $= 0$, sinon var_size est incrémenté. Ensuite, pour chaque worker, on définit le nombre d'itérations à traiter workload $= p_w \times |I|$ et le nombre maximum d'itérations allouables en mémoire it_count $= \lfloor \frac{\text{mem\_limit} - \text{cst\_size}}{\text{var\_size}} \rfloor$. On peut alors définir le nombre de tâches exécutées par le worker : $\frac{\text{workload}}{\text{it\_count}}$.

Le runtime va ensuite calculer les dépendances des nouvelles tâches à générer. Pour chaque région mémoire accédée, on vérifie les dépendances entre chacune des tâches nouvellement créées et chacune des tâches en cours liées à cette région mémoire. La détection est effectuée avec l'algorithme présenté en sous-section 5.3.3.

Les tâches sans dépendance sont ensuite ajoutées à la file de tâches, prête à l'exécution des workers associés.

5.5.4 Implémentation des threads worker

Les threads workers ont une boucle principale décomposée en quatre étapes (voir code 5.4). Premièrement, on récupère une tâche prête à être exécutée dans la pile. La pile de tâches fonctionne en mode Last In First Out car on suppose que si une dépendance a été satisfaite récemment, les données ont plus de chances d'être disponibles dans la mémoire du worker.

La fonction `prepare_task` consiste à migrer les données vers la mémoire utilisée par le worker et, pour les coprocesseurs, à allouer l'espace mémoire nécessaire si besoin.

Ensuite, la fonction enregistrée par l'utilisateur est liée à la tâche et exécutée par le thread. La version sélectionnée dépend du matériel associé au worker.

Enfin, la fonction end_task gère les dépendances et les informations de monitoring.

```
void worker_loop( struct worker_data_t* sched_data, struct worker_func_t* funcs,
    void* worker_data, void* worker_ctx  )
{
    while( running == 1 )
    {
        struct task_t* task = pop_last_task( sched_data->task_list );

        funcs->prepare_task( task, worker_data, worker_ctx );

        if( task->metatask->function_id != FUNC_SKIP_RESERVER_ID )
        {
            uint32_t func_pos = task->metatask->function_id *
                sched_data->hardware_type_count + sched_data->hardware_type_id;
            sched_data->function_list[func_pos]( worker_ctx,
                task->metatask->common_args,
                task->memory_args );
        }
        funcs->finish_task( task, worker_data, worker_ctx );
    }
}
```

Code 5.4 – Boucle principale d'un thread worker

5.5.5 Spécificités des workers GPU

Les workers GPUs effectuent les fonctions de manière asynchrone. L'objectif est de permettre du parallélisme entre les communications et les calculs de plusieurs tâches différentes. Pour cela un worker GPU va exécuter jusqu'à 3 tâches en parallèle.

Trois streams CUDA sont créés par le worker pour permettre le parallélisme : prepare_stream, work_stream et finish_stream, qui gèrent respectivement l'allocation et le chargement des données, le lancement des kernels et la fin de la tâche. Les synchronisations entre ces streams sont faites à l'aide d'événements CUDA.

Pour obtenir des allocations de mémoire performantes, on utilise l'API : `Stream Ordered Memory Allocator` de CUDA qui permet d'allouer et de désallouer la mémoire de manière asynchrone. Cette API permet d'allouer de la mémoire dans une `memory pool`, la mémoire allouée liée à la `memory pool` peut être réutilisée pour plusieurs allocations successives sans avoir besoin de passer par un appel système. Désallouer la mémoire de manière asynchrone ne désalloue pas la mémoire de la `memory pool` tant que la quantité de mémoire est inférieure à un seuil. Pour limiter l'impact des allocations/désallocations, on utilise le code 5.5 à l'utilisation, qui permet de mettre le seuil à la valeur maximale de la mémoire du GPU.

Ce faisant, les allocations mémoires sont uniquement gérées dans l'espace utilisateur une fois que la `memory pool` a alloué l'intégralité de la mémoire du GPU.

En pratique, le seuil d'allocation mémoire `memory pool` est fixé à 75%, cela permet de pouvoir utiliser des kernels utilisant l'allocation dynamique. De plus, nous avons observé que la `memory pool` pouvait surallouer, induisant des erreurs du type `cudaErrorMemoryAllocation` lorsque le seuil était placé à des valeurs supérieures à 0.9 fois la taille totale de la mémoire GPU.

```
cudaMemPoolCreate( &memory_pool, &pool_props )
cudaMemPoolSetAttribute( memory_pool, AttrReleaseThreshold, &total_size )
cudaDeviceSetMemPool( device_id, memory_pool )
```

Code 5.5 – Gestion de la pool d'allocation mémoire

Un suivi de la mémoire allouée/disponible est effectué par le thread worker, la désallocation des pointeurs qui ne sont plus utilisés est effectuée uniquement lorsque l'on veut allouer un nouvel espace mémoire et que la `memory pool` ne dispose plus de suffisamment de mémoire. On classe les pointeurs en

différentes catégories : les pointeurs en cours d'utilisation, les pointeurs valides, les pointeurs sauvegardés, les pointeurs invalides.

Le thread désalloue en priorité les pointeurs invalides qui contiennent des données qui ont été explicitement ignorées par l'utilisateur ou invalidées par la sortie d'une tâche. Ensuite, les pointeurs sauvegardés sont désalloués. Ils comportent des données qui ont été rapatriées sur le CPU et qui ne seront pas perdues à la désallocation. Les pointeurs valides contiennent des données qui ne sont présentes que sur le GPU. Avant d'être désalloués, un transfert sera effectué pour enregistrer ces données en mémoire CPU. Enfin, les données en cours d'utilisation ne sont jamais désallouées.

Lors d'un transfert de données, le thread initiant le transfert cherche la source qui permettra le transfert avec la plus grande bande passante [30]. On choisit par ordre de priorité : données locales, GPU connecté via Nvlink, RAM CPU, autre GPU. Chaque segment mémoire est stocké dans un tableau ordonné pour faciliter cette recherche. Dans le cas où le réseau Nvlink est non complet, les transferts entre deux GPU non connectés passent toujours par la RAM CPU, car nous n'avons pas implémenté de méthodes de routage permettant d'utiliser le Nvlink de manière indirecte.

5.5.6 Monitoring

Deux méthodes de monitoring sont mises en place : une pour l'estimation des performances utilisées par le runtime pour la répartition des tâches, et une pour générer des traces d'exécutions.

Les performances sont mesurées en utilisant des mesures de temps classiques `clock_gettime` au début et à la fin de l'exécution de chaque tâche, les temps de communication sont pris en compte mais une option est disponible pour les ignorer dans cette mesure.

Les traces utilisateurs sont implémentées avec l'API NVTX de Nvidia, ce qui permet d'avoir les traces visibles lorsque l'application est profilée avec les outils de Nvidia (*nvvp*, *nsight-system*). Ce monitoring peut être désactivé avec une variable d'environnement ou à la compilation pour limiter les impacts sur les performances.

5.5.7 Gestion Mémoire

La gestion des accès mémoire est primordiale pour détecter les dépendances entre les tâches et effectuer les mouvements de données. Dans certains cas, des limitations de performances découlent de cette implémentation et seront présentées en sous-section 5.6.4.

Définition des dépendances Pour chaque argument d'accès à un tableau, les tâches contiennent une structure de type $\Gamma = \bigcup_{i=0}^{N-1}[a \cdot i + b; a \cdot i + b + c]$. La structure en mémoire nécessaire pour stocker cette information est le n-uplet (a,b,c,N). Le tableau accédé contient alors une liste des tâches qui sera parcourue en $\mathcal{O}(A)$ avec A le nombre de tâches qui accèdent au tableau pour vérifier les dépendances.

Pour accélérer la méthode, nous parcourons en particulier la liste des méta-tâches avant de parcourir la liste des sous-tâches de cette méta-tâche lorsque qu'une collision est détectée.

Gestion de l'état de la mémoire Le statut de la mémoire est sauvegardé sous forme d'une liste de segments ordonnés. L'ordre est le suivant. Pour des segments $S_i = [a_i, b_i]$, on a $S_i > S_j$ si et seulement si $a_i > a_j$ ou $a_i = a_j$ et $b_i < b_j$. De plus, chaque segment contient des informations sur sa position en mémoire, sa correspondance sur un device si le segment n'est pas hébergé en RAM CPU, son état associé à un événement CUDA pour savoir si la donnée est prête.

Une liste chaînée est maintenue pour chaque tableau alloué par l'utilisateur. L'ajout d'un segment se fait en temps linéaire $\mathcal{O}(M)$ avec M le nombre de segments déjà présents. De même, l'ajout d'un ensemble de segments déjà ordonné (cas typique d'un Γ) est également effectué en temps linéaire $\mathcal{O}(M + V)$ avec V le nombre de segments à ajouter. La suppression d'un segment est réalisée en temps constant $\mathcal{O}(1)$.

Lorsqu'un segment est généré par une écriture, il doit invalider les données déjà présentes. Pour cette opération, on recherche tous les segments qui ont une intersection avec le segment ajouté et on les divise pour retirer la section de l'intersection. Cette opération est aussi en $\mathcal{O}(M)$ par segments ajoutés.

Finalement, une dernière opération est utilisée pour fusionner les segments CPU entre eux : les segments présents sur la mémoire CPU sont fusionnés s'ils ont une intersection pour limiter l'expansion de

la liste chaînée. Cette fusion est effectuée uniquement lorsque les données sont valides et en place, de plus elle est effectuée de manière opportuniste lors des opérations d'ajout d'un segment.

Le cycle de vie des segments est le suivant :

- Création d'une tâche :
  - Les segments sont créés à la création des sous-tâches.
  - Si la tâche doit écrire sur ces segments, elle invalidera des segments de la liste chaînée.
  - Les segments sont ajoutés à la liste chaînée.
- Lorsqu'un segment sur CPU devient valide (un transfert de données se termine de manière asynchrone), un flag est ajouté pour qu'il soit fusionné avec ses potentiels voisins.
- Lorsqu'un segment (a) sur GPU englobe un ancien segment (b) présent sur le même GPU : le segment (b) est retiré de la liste.
- Lorsque le GPU désalloue de la mémoire, les segments associés sont retirés de la liste et les données sont sauvegardées sur le CPU si nécessaire.

5.6 Benchmarks

Les benchmarks suivants ont été effectués pour évaluer les performances de notre runtime et d'un modèle de tâches moldables. La sous-section 5.6.1 présentera une exécution sur des batchs de multiplication de matrices, la deuxième sous-section 5.6.2 présentera les performances pour l'algorithme de formation de voies classique, la troisième sous-section 5.6.4 sera portée sur la factorisation de cholesky présentée dans la sous-section 4.6.3 du chapitre 4, et la dernière sous-section 5.6.3 présentera une mesure du temps d'ordonnancement des tâches et de calcul des dépendances lors de l'ordonnancement de l'algorithme de formation de voies.

5.6.1 Batch de multiplication de matrices

Les premières expériences permettent de tester le runtime avec une application basique pour mettre en avant l'évolution de la découpe des tâches au cours du temps. Pour ce test, un serveur hétérogène, nommé Ramses, a été utilisé avec 1 CPU et 2 GPU différents, les caractéristiques sont détaillées dans le tableau 5.1.

Fréquence CPU : La fréquence du CPU est fixée à 1,6 GHz, car les applications testées utilisent intensivement les unités AVX512 du processeur. Lorsque ces unités de calculs sont utilisées de manière prolongée, le processeur réduit automatiquement la vitesse du cœur à 1,6 GHz. Cette baisse de fréquence ne tient pas compte des réglages de l'utilisateur. Pour éviter toute modulation de fréquence lors des expérimentations, la fréquence a donc été fixée à 1,6 GHz et les C-states ont été réglés à la valeur C0. Les commandes suivantes ont été utilisées pour régler la fréquence des CPU :

```
cpupower frequency-set -g userspace
cpupower frequency-set -f 1600000
```

Fréquence GPU : La fréquence des GPU a été fixée à 1.8 GHz. Cette fréquence permet de faire tourner des applications intensives sans subir de baisse de fréquence due aux contraintes thermiques (thermal throttling). Cette limite a été mesurée de manière expérimentale et peut varier pour chaque carte et chaque serveur, car la capacité de la carte à évacuer sa chaleur dépend de paramètres externes tels que le système de refroidissement ou la température de l'air ambiant. Les commandes suivantes ont été utilisées pour régler les fréquences des GPU et les GPU utilisés :

```
nvidia-smi -pm 1
nvidia-smi -ac 6501,1800 -i 0
nvidia-smi -ac 6501,1800 -i 2
CUDA_VISIBLE_DEVICES=0,2
CUDA_DEVICE_ORDER=PCI_BUS_ID
```

Processeur	Coeurs	Fréquence (GHz)	perf FP32 (GFlop/s)	perf FP64 (GFlop/s)
Xeon Platinum 8253	16	1.6	1638	819
RTX 8000	4608	1.8	16500	512
RTX 4000	2304	1.8	8300	256

TABLE 5.1 – Caractéristiques de la workstation Ramses

Performances théoriques : Sur CPU, les performances théoriques en FP32 sont calculées avec la formule suivante :

$$fp_{32} = 2 \times (f \times c \times (a \times 512)) \times \frac{1}{32}$$

Avec f la fréquence, c le nombre de cœurs, a le nombre d'unités AVX512 par cœur (2 dans le cas du Xeon), la performance 64 bits $fp_{64} = \frac{fp_{32}}{2}$.

Pour les GPUs, on utilise la formule :

$$fp_{32} = 2 \times f \times c$$

Le coefficient 2 est présent car chaque opérateur (CUDA core sur GPU, et unité AVX sur CPU) effectue des opérations de type Fused Multiply-Addition ($d = a * b + c$) à chaque cycle, donc 2 opérations par cycle. Ce calcul permet de rester cohérent avec les notations utilisées par les fabricants de matériel. La performance en double précision est $fp_{64} = \frac{fp_{32}}{32}$ pour les cartes RTX.

L'algorithme mesuré dans cette section est un batch de multiplications de matrices, chaque tâche étant exécutée une à une, ce qui permet d'observer l'évolution de la découpe du runtime au cours du temps. Les fonctions cblas_cgemm et cblas_zgemm de la bibliothèque Intel MKL sont utilisées sur CPU et les fonctions cublasCgemmStridedBatched et cublasZgemmStridedBatched de la bibliothèque Nvidia cuBLAS sur GPU.

Le code exécuté 5.6 est une boucle avec un appel à la fonction GEMM et une synchronisation. Les mesures sont faites à l'aide des traces.

```
for( int i = 0; i < it_count; i++ )
{
    spawn_function( gemm_strided_id, &c_args, sizeof(c_args), &infos, &perf, 1 );
    scheduler_sync_on_array( &C )
}
```

Code 5.6 – Appel aux fonctions GEMM

La première observation porte sur la vitesse de stabilisation de la découpe. L'exécution se fait avec un ensemble de 16 workers composé de 2 threads GPU et 14 threads CPU. En suivant les performances théoriques du tableau 5.1, on obtient une puissance disponible de 26 Tflop/s en FP32 et 1.1 Tflop/s en FP64. On s'attend donc à avoir la répartition suivante du travail en FP32 : 64% de la charge pour la RTX8000, 32% pour la RTX4000, et 0.2% pour chacun des cœurs. En FP64, on aurait respectivement 34%, 17% et 3.5%.

Les figures 5.4 et 5.5 représentent respectivement la répartition du travail pour chaque itération pour des calculs en FP32 et en FP64. Premièrement, on observe une convergence rapide de la découpe : les deux exécutions convergent en 3 itérations. En revanche, la répartition est éloignée des attentes théoriques. En FP32, la RTX8000 effectue 50% des opérations contre 64% attendus, la RTX 4000 effectue 38% des calculs, et les processeurs 0.8% chacun.

Ce décalage peut être expliqué car les temps de transfert ne sont pas pris en compte dans l'estimation de performance théorique, or on peut observer dans la timeline fournie en figure 5.6a que les transferts de données PCIe ne sont pas masqués lorsque l'on exécute les opérations sur la RTX8000 car le ratio entre la puissance de calcul et la bande passante PCIe est trop important. Les performances de la RTX4000 sont moins impactées par ce problème car la bande passante du bus PCIe est la même alors que la puissance de calcul de la carte est divisée par deux.

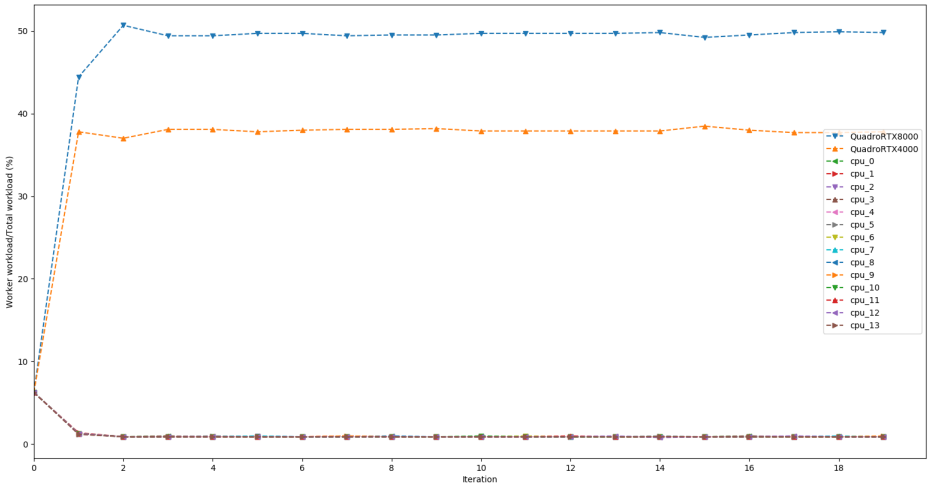


FIGURE 5.4 – Découpe des tâches pour la fonction gemmBatch en simple précision

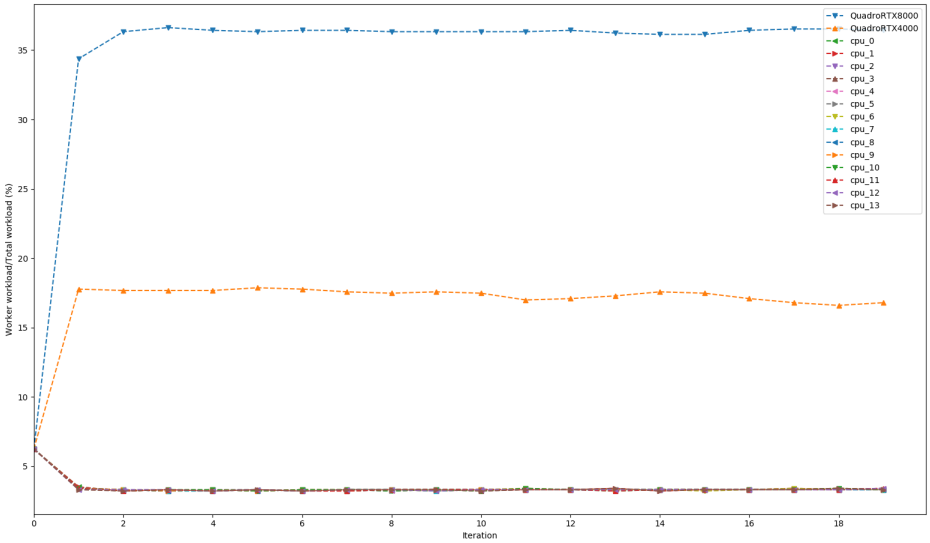
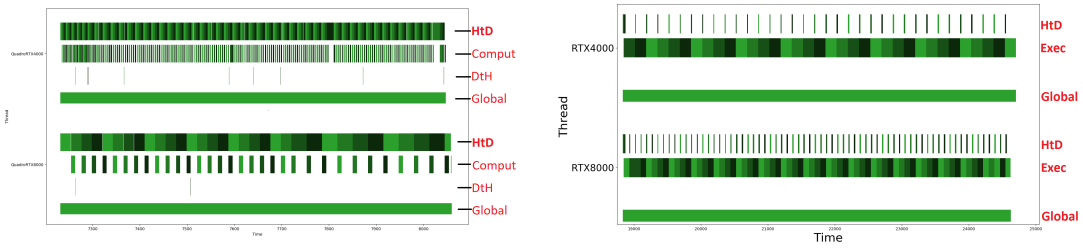


FIGURE 5.5 – Découpe des tâches pour la fonction gemmBatch en double précision

Ressource workload	Itération number							
	1	2	3	4	8	12	16	20
RTX8000 (%)	6.25	35	36	36	36	36	36	36
RTX4000 (%)	6.25	18	18	18	18	18	18	18
CPU core (%)	6.25±0.00	3.32±0.20	3.31±0.08	3.30±0.12	3.31±0.11	3.31±0.08	3.31±0.11	3.31±0.11
CPU Times (s) (*avvg*)	10.63	6.20	6.17	6.16	6.18	6.18	6.18	6.18
(*max*/*min*)	12.08/1.12	6.52/5.84	6.28/5.96	6.32/6.01	6.30/6.06	6.28/6.11	6.33/6.06	6.30/6.06

TABLE 5.2 – ZGEMM workload repartition against iteration number



(a) Timeline GEMM FP32

(b) Timeline GEMM FP64

FIGURE 5.6 – Timeline d'exécution des GPUs pour un appel gemmBatch

Pour les opérations en FP64, la répartition effective est de 36%, 18% et 3.3%, beaucoup plus proche des valeurs théoriques. Le détail de la répartition est fourni dans le tableau 5.2. La timeline en figure 5.6b montre que dans le cas d'une exécution en double précision, les GPUs sont limités par leur puissance opérative et non plus par la bande passante du bus PCIe.

De plus, les timelines permettent d'observer le comportement de la découpe des tâches pour les GPUs. Premièrement, on voit sur la timeline 5.6b que la RTX8000 a deux fois plus de tâches que la RTX 4000 (69 vs 35), et que ces tâches sont exécutées deux fois plus rapidement. Ces résultats découlent des particularités de la découpe GPU, présentées en section 5.5.5. Dans ce cas, la taille des tâches est fixée par la cible mémoire de 200 Mo allouée par tâche, ce qui permet d'augmenter le nombre de tâches et de masquer plus facilement les transferts mémoires. On voit aussi que les GPUs n'ont pas besoin d'évincer, donc aucun transfert device-to-host n'est effectué.

En revanche, sur la timeline du calcul en simple précision (figure 5.6a), les GPU gèrent une plus grosse part du travail. La taille des tâches allouées à la RTX4000 ne rentre pas en mémoire, ce qui impose des évictions et des transferts de données.

5.6.2 Formation de voies classique

L'algorithme de formation de voies classique a été présenté en section 2.4, le chapitre 3 a montré comment regrouper les tâches pour augmenter les performances lors d'une exécution mono-GPU et une version moldable a été proposée en section 4.6.4. Cette sous-section présentera l'implémentation de la formation de voies classique en utilisant le runtime de tâche moldable précédemment présenté et analysera les performances sur deux serveurs différents.

Une partie des expérimentations a été exécutée sur le serveur Ramses présenté précédemment en tableau 5.1. Un second serveur, Sirius, a été utilisé. C'est un serveur de type DGX A100 mis à disposition par grid5000 [5]. Les performances théoriques du serveur sont présentées en table 5.3, la topologie mémoire en figure 5.7.

L'implémentation de la boucle principale avec l'ordonnanceur est présentée en code 5.7. Les réglages ont été effectués avec les mêmes commandes qu'en sous-sections 5.6.1. Les dimensions suivantes ont été utilisées pour les mesures : 1024 colonnes, 1024 lignes, 1024 voies verticales, 1024 voies horizontales, 8 spectres, 256 échantillons par spectre, 256 fréquences observées, 20 récurrences. Les dimensions utilisées sont volontairement élevées par rapport aux antennes SONAR réelles, cela permet d'augmenter l'espace mémoire nécessaire pour qu'elles ne tiennent pas en mémoire sur les GPUs.

Processeur	Coeurs	Nombre	Fréquence (GHz)	perf FP32 (GFlop/s)	perf FP64 (GFlop/s)
AMD EPYC 7742	64	2	2.0	2×4096	2×2048
Nvidia A100	6912	8	1.4	8×19300	8×9600

TABLE 5.3 – Caractéristiques du serveur DGX A100 Sirius

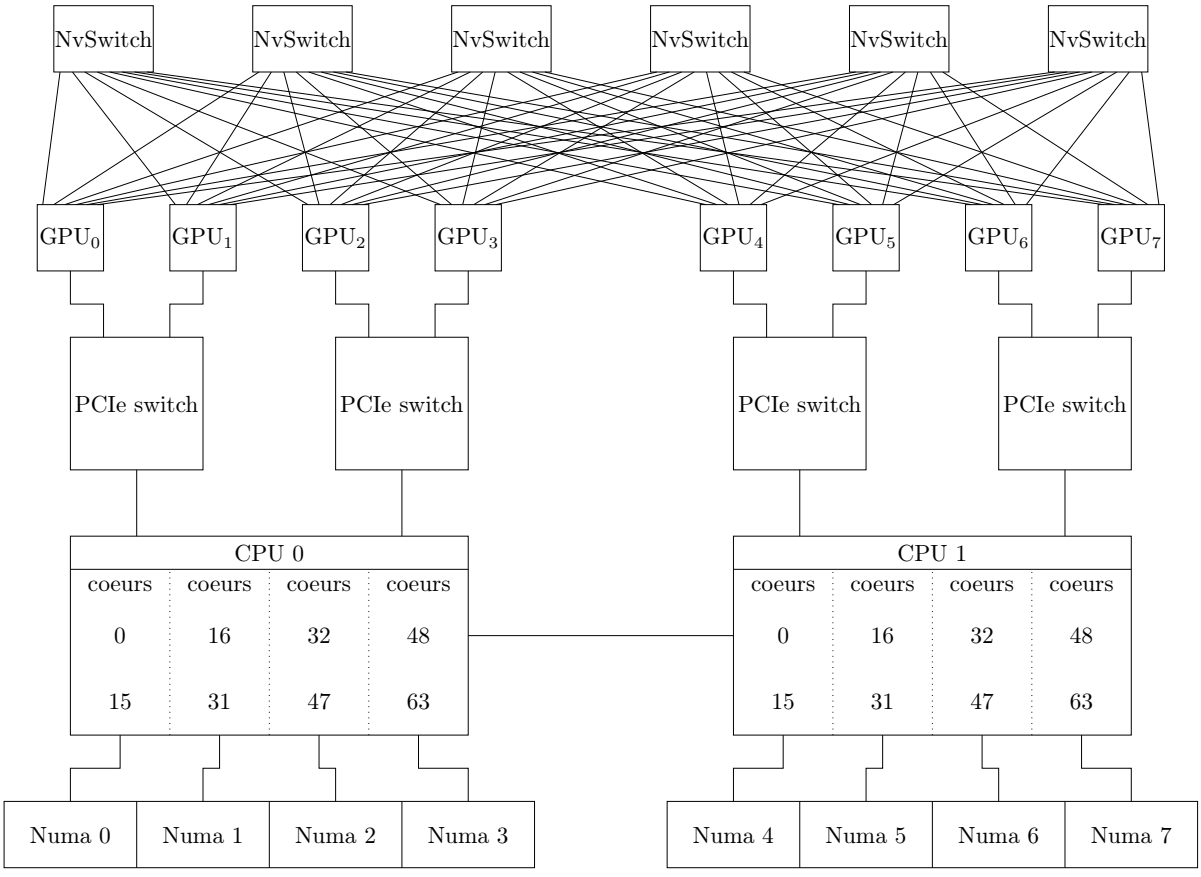
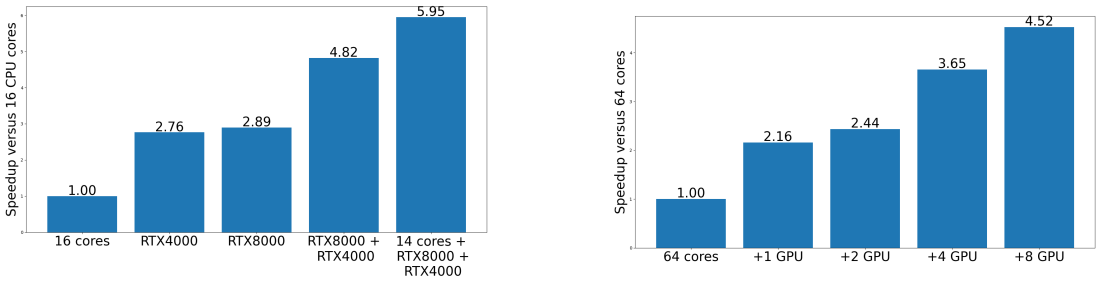


FIGURE 5.7 – Topologie mémoire du serveur DGX A100 Sirius

Pour l'implémentation des tâches, les bibliothèques utilisées côté GPU sont Nvidia Cublas et CuFFT. Côté CPU, Ramses utilise une implémentation avec Intel MKL, tandis que l'implémentation Sirius utilise OpenBLAS et Intel MKL pour les FFT. Cela permet d'utiliser les opérations vectorielles sur les processeurs AMD pour les opérations BLAS.

Exécution CPU. Les premières mesures ont été effectuées uniquement sur CPU, l'objectif est d'observer si le runtime arrive à ordonnancer les tâches correctement sur plusieurs cœurs. Le code a été exécuté sur la plateforme Ramses en utilisant $n = 1, 2, 4, 8, 12$ et 16 cœurs. Les allocations mémoire ont été faites sur un seul nœud numa et les cœurs utilisés sont sur le même nœud numa que la mémoire allouée, 16 cœurs physiques sont laissés disponibles sur un second nœud numa, ce qui doit limiter le nombre de threads que l'OS va exécuter sur les cœurs utilisés par le runtime.

On observe un temps d'exécution homogène entre les différents cœurs, chaque tâche est découpée en n sous-tâches de tailles égales par le runtime. Les speedup observés sont respectivement de 1, 1.98, 3.95, 7.80, 11.50, 14.70, proches d'un speed-up linéaire. On observe donc que le runtime arrive à ordonnancer de manière équilibrée des tâches moldables sur une architecture homogène sans transferts de données, tout en ayant un surcoût d'ordonnancement limité pour ce problème. Des mesures plus précises du surcoût seront présentées en sous-section 5.6.3.



(a) Ramses (b) Sirius

FIGURE 5.8 – Beam-forming speedups

Exécution sur Ramses. Les mesures suivantes sont faites avec une architecture hybride, avec jusqu'à 16 coeurs et 2 GPU différents. Le temps de référence est l'exécution sur 16 threads CPU qui a été mesuré à 28 secondes. Comme précisé précédemment, il est 14.7 fois plus rapide que l'exécution séquentielle. L'exécution avec la RTX4000 uniquement (un seul worker) est 2.76 fois plus rapide que l'exécution de référence, tandis que la RTX8000 a un speed-up de 2.89. La troisième exécution avec les deux GPUs (2 workers) a un speed-up de 4.82. Finalement, une exécution avec 14 threads CPU et 2 threads GPU (16 workers) a un speed-up de 5.95. Les résultats sont disponibles sur la figure 5.8a.

Nous avons vu en sous-section 5.6.1 que les multiplications de matrices en 32 bits étaient plus limitées par la bande passante du PCIe sur les GPUs à disposition. De plus, la formation de voies est composée de fonctions qui ont une complexité arithmétique plus faible que la multiplication de matrice par rapport au nombre de données traitées (FFT, valeurs absolues, gemm). Il est donc attendu d'avoir des limitations de performances dues au PCIe. Cela explique que les deux GPUs aient un temps d'exécution proche alors que leurs performances théoriques sont différentes.

L'exécution utilisant 2 GPU seulement a un speed-up de 4.82 seulement, alors qu'on aurait pu s'attendre à un speed-up de $2.76 + 2.89 = 5.65$. Le speed-up est bon car chaque GPU est sur un nœud PCIe distinct. En revanche, le partage des tâches induit des communications supplémentaires entre les GPUs, ce qui conduit à un speed-up sous-linéaire.

La dernière exécution proposant d'utiliser des coeurs CPUs (14) et les GPUs, deux résultats théoriques sont attendus : $\frac{14}{16} + 2.76 + 2.89 = 6.52$ dans le cas où les speed-up des GPUs et des CPU se somment linéairement, ou $\frac{14}{16} + 4.82 = 5.69$ dans le cas où on prend en compte le résultat d'exécution précédent avec seulement les deux GPUs. Le speed-up mesuré de 5.95 est intermédiaire, cela s'explique par le fait

que donner moins de travail aux GPUs permet de limiter les mouvements de données. On voit donc un intérêt à faire une exécution hétérogène.

Exécution sur DGX A100. Les dernières mesures ont été effectuées sur la plateforme Sirius (table 5.3), les résultats sont disponibles en figure 5.8b. La référence est une exécution sur 64 cœurs mesurée à 9 secondes. Toutes les mesures sur Sirius ont été effectuées avec 64 workers dont $n = 0$, 1, 2, 4 et 8 GPUs (donc $64 - n$ cœurs CPUs), les cœurs CPU utilisés sont tous sur le CPU 0 et les données sont allouées sur les nœuds NUMA 0, 1, 2 et 3 reliés au CPU 0 (voir 5.7). Les GPUs utilisés sont pour un GPU : 0, pour deux GPUs : 0 et 1, pour quatre GPUs : 0, 1, 2, 3, pour 8 GPUs : 0, 1, 2, 3, 4, 5, 6, 7, 8.

On observe des speed-up limités : 2.16 avec un GPU, 2.44 avec deux, 3.65 avec quatre et 4.52 avec 8 GPUs. En particulier, l'augmentation du speed-up en passant de 1 à 2 GPUs est limitée, comme on peut le voir sur le schéma de la topologie les GPUs 0 et 1 partagent le même switch PCIe, les deux GPUs utilisent donc un bus de communication commun qui limite leurs performances. Une exécution supplémentaire avec les GPUs 0 et 2 permet de s'affranchir de cette limitation et atteint un speed-up de 3.5, ce qui est proche du speed-up observé avec 4 GPUs partageant 2 switchs.

Les transferts de mémoire ont été monitorés sur une exécution avec 8 GPUs, ce qui nous a permis de constater que seulement 16% des données sont réutilisées avant d'être évincées de la mémoire du GPU.

5.6.3 Surcoût d'ordonnancement

Des mesures ont été effectuées sur l'exécution du code de formation de voies avec les mêmes paramètres que dans le chapitre 5.6.2, à l'exception du nombre de récurrences qui est passé de 20 à 10. Les performances ont été mesurées sur la plateforme Ramses avec le même code utilisateur. En revanche, le runtime a été modifié pour ajouter des points de mesure de temps et des compteurs : le nombre de tâches créées, le nombre de sous-tâches créées, le nombre de dépendances créées et le temps passé dans les fonctions de création de tâches et de calculs de dépendances.

Pour rappel, le thread utilisateur est chargé de créer les tâches et de calculer les dépendances, il n'y a donc pas de problème d'exécution concurrente possible. Pour limiter l'impact des mesures sur les performances, les temps et les compteurs sont incrémentés à l'exécution en utilisant une variable unique et les valeurs obtenues sont des moyennes sur ces mesures. Lorsque plusieurs tâches ou plusieurs dépendances sont créées en même temps, une seule mesure est faite pour limiter les appels à la fonction clock_gettime et les accès aux compteurs.

Lors de l'exécution, le code a généré 40 tâches moldables, qui ont été décomposées en 4062 sous-tâches avec 103086 dépendances générées entre ces sous-tâches. Le temps de génération des tâches moldables a été mesuré à 360µs, moins de 4µs par sous-tâches et 140ns par dépendances. Ces temps sont possibles car l'implémentation présentée précédemment limite les appels système et la structure des tâches moldables permet de grouper des calculs lors de la création des sous-tâches et des dépendances.

5.6.4 Factorisation de cholesky

Dans la section 4.6.3, nous avons vu une méthode permettant d'implémenter une factorisation de Cholesky en utilisant des tâches moldables. Une implémentation a été réalisée avec ce runtime pour concrétiser ce concept.

Ces expérimentations nous ont permis de mettre en avant certaines limitations de l'implémentation du runtime en particulier au niveau de la gestion des données. Nous avons observé une baisse de performances importante corrélée avec le nombre de workers utilisés pour traiter le problème, les causes de ces mauvaises performances ont été identifiées et sont liées à certains choix d'implémentation du runtime initiés avec les développements des applications et benchmarks précédents.

La suite de cette section détaille les expérimentations menées et les causes des mauvaises performances observées.

Implémentation. L'implémentation de la factorisation est fournie en code 5.9. L'algorithme est la version "left-looking" décrite dans la section 4.6.3 du chapitre précédent, avec une découpe de taille `NB` et une taille de matrice de `N*N`.

```
// Initialisation des objets, setmeminfo indique les valeurs ws, ss, es, ...
struct sched_mem_info_t fft_mem_infos;
create_mem_info( &fft_mem_infos, 2, 1 );
setmeminfo( &fft_mem_infos, 0, 0, &a_hydro_t, sizeof(float) , n_ech,
        sizeof(float), sizeof(float)*n_ech,
        n_row*n_col*n_spe, SCHED_READ  )
setmeminfo( &fft_mem_infos, 1, 0, &a_hydro_f, 2*sizeof(float) , n_ech,
        2*sizeof(float), 2*sizeof(float)*n_ech,
        n_row*n_col*n_spe, SCHED_WRITE )

// ....

for( int i_rec = 0; i_rec < n_rec; i_rec++ )
{
    scheduler_sync();
    clock_gettime( CLOCK_MONOTONIC, &t_start );
    spawn_function( fn_fft_id,
        &fft_common_args, sizeof(struct fft_common_args_t),
        &fft_mem_infos, &fft_perf, 1 );

    spawn_function( fn_gemm_strided_id_0,
        &gemm_0_common_args, sizeof(struct gemm_common_args_t),
        &gemm_0_mem_infos, &gemm_0_perf, 1 );

    spawn_function( fn_gemm_strided_id_1,
        &gemm_1_common_args, sizeof(struct gemm_common_args_t),
        &gemm_1_mem_infos, &gemm_1_perf, 1 );

    spawn_function( fn_abs_stab_id,
        &abs_stab_common_args, sizeof(struct abs_common_args_t),
        &abs_stab_mem_infos, &abs_stab_perf, 1 );

    clear_mem(&a_energy);
    clear_mem(&a_hydro_f);
    clear_mem(&a_pseudo);
    clear_mem(&a_voies);
    scheduler_sync();

    clock_gettime( CLOCK_MONOTONIC, &t_stop );
}
```

Code 5.7 – Boucle principale de la formation de voies classique

Il est basé sur quatre fonctions BLAS : `syrk`, `potrf`, `gemm` et `trsm`. L'algorithme met à jour les valeurs de la matrice par blocks de `NB` colonnes, l'identifiant du block est $j \in [\![0, N/NB - 1]\!]$.

À chaque itération, on applique les fonctions BLAS dans l'ordre suivant :

— `syrk` : mise à jour de la sous-matrice (j, j) de taille (NB, NB) à partir de la sous-matrice $(j, 0)$ de taille $(NB, j \times NB)$. La tâche peut être décomposée en $j \times NB$ sous-tâches, elles sont mutuellement exclusives car les résultats de chacune des sous-tâches sont accumulés dans la sous-matrice (j, j).

— `potrf` : mise à jour de la sous-matrice (j, j) de taille (NB, NB), cette tâche n'est pas moldable.

— `gemm` : mise à jour de la sous-matrice $(j + 1, j)$ de taille $(N - j \times NB, NB)$ à partir de la sous-matrice $(j - 1, 0)$ de taille $((j + 1) \times NB, (j - 1) \times NB)$. La tâche est décomposée en $N - j \times NB$ sous-tâches indépendantes.

— `trsm` : mise à jour de la sous-matrice $(j+1, j)$ de taille $(N-j \times NB, NB)$ à partir de la sous-matrice (j, j) de taille (NB, NB). Elle peut être décomposée en $N - j \times NB$ sous-tâches indépendantes.

La matrice d'entrée, nommée A, est stockée sous un format « column-major » (les éléments d'une même colonne sont consécutifs en mémoire).

Génération des tâches et des dépendances. Le code a été exécuté sur un problème de taille $N = 5120$ et $NB = 1024$ ($\frac{N}{NB} = 5$). Deux exécutions différentes ont été proposées. La première consiste à limiter la moldabilité des tâches pour que les découpes soient liées aux blocs de tailles `NB`. Les exécutions sont effectuées par 4 workers.

Limiter la granularité des tâches moldables à une taille de `NB` nous permet d'obtenir une découpe équivalente à celle d'un algorithme de type "right-looking". Le graphe 5.9a représente le graphe des tâches de cette exécution, il comporte 31 nœuds (5 `potrf`, 10 `trsm`, 10 `syrk`, 6 `gemm`) et 48 dépendances (moyenne de 1.5 dépendances par tâche).

Dans le cas où l'on ne limite plus la granularité, les tâches moldables (`syrk`, `trsm`, `gemm`) seront décomposées en 4 tâches. Le graphe 5.9b comporte 49 tâches (5 `potrf`, 16 `trsm`, 16 `syrk`, 12 `gemm`) en particulier : 6 tâches `gemm` et 6 tâches `trsm` supplémentaires que l'exécution avec la granularité limitée. En revanche, les données utilisées par les tâches ont des recouvrements, on a donc plus de dépendances que précédemment, 98 en moyenne (2 dépendances par tâche).

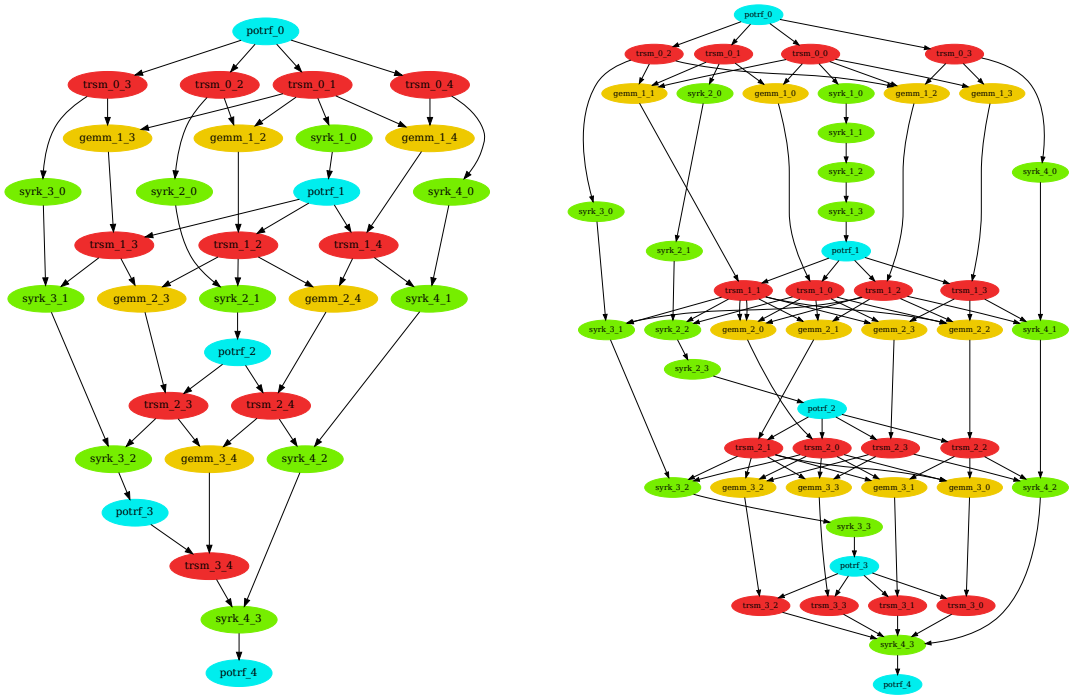
L'ajout de ces nouvelles tâches permet d'augmenter le parallélisme ; cependant, les dépendances sur des zones mémoires qui se recouvrent partiellement impliquent que certaines tâches vont attendre des données dont elles n'ont pas l'utilité.

Limitations dues à la gestion des segments mémoire. Le monitoring de l'algorithme de Cholesky a mis en évidence des défauts liés à certaines décisions d'implémentation, en particulier des effets qui ne se manifestaient pas ou qui étaient négligeables dans le cas de l'exécution de la formation de voies.

Lors de l'allocation d'un tableau mémoire par l'utilisateur, le runtime ajoute des structures de données pour gérer les dépendances liées à ce tableau. Dans l'implémentation de la formation de voies, on manipule différents tableaux et chaque tableau est écrit par une seule tâche moldable à chaque itération. A l'opposé, l'algorithme de Cholesky travaille sur un tableau commun, représentant la matrice de travail, pour tout l'algorithme, chaque sous-élément du tableau est modifié plusieurs fois au cours d'une même itération. De nombreuses étapes du cycle de vie des tâches ont besoin d'accéder à ce tableau, en particulier à la liste des segments mémoires associés, or ces structures sont protégées par des mutex ce qui limite le parallélisme des étapes d'initialisation et de destruction des tâches.

Les trois étapes principalement impactées sont : la détection et l'ajout de dépendances, la définition des nouveaux segments mémoires et le chargement des données, le nettoyage des segments mémoires.

La gestion des dépendances est suffisamment rapide pour ne pas être notable sur le temps d'exécution, en revanche, la gestion des segments mémoires est un point bloquant. Le fait que l'on travaille sur des sous-matrices implique que chaque tâche manipule un nombre de segments de l'ordre de `NB` ou `N`, en effet l'accès à une sous-matrice génère un nombre de segments égal au nombre de colonnes. Et la manipulation de ces segments est verrouillée par un mutex, ce qui rend séquentielle cette étape pour l'algorithme de Cholesky.



(a) Découpe limitée à des blocs de taille NB.

(b) Découpe en 4 sous-tâches par tâches moldable.

FIGURE 5.9 – Graphe des tâches pour l'algorithme de cholesky $\frac{N}{NB} = 5$

Limitations dues à l'utilisation intensive des streams CUDA. Les tâches s'exécutant sur le GPU sont ralenties par le nombre important de segments à gérer. En effet, les communications asynchrones entre le CPU et le GPU nécessitent l'utilisation de streams CUDA. Or, pour chaque segment, plusieurs actions sont insérées dans une stream CUDA, telles que l'attente d'un événement, le transfert des données, l'enregistrement d'un événement, et l'appel à une fonction asynchrone côté CPU pour mettre à jour certains segments (CUDA Callback). Cependant, ces actions sont effectuées pendant le parcours de la liste des segments du tableau associé à la matrice, tout en verrouillant le mutex afin d'éviter qu'un autre thread invalide un segment en cours d'utilisation. Si les actions sur la stream CUDA étaient purement asynchrones, les impacts sur les performances seraient semblables à ceux observés pour une exécution purement GPU. Cependant, les streams ont un nombre d'actions en attente limité, au-delà duquel toute nouvelle insertion d'une action asynchrone dans la stream devient bloquante jusqu'à ce que le runtime CUDA consomme une action de la stream. Par exemple, l'insertion d'une action de copie mémoire peut devenir synchrone et attendre que le transfert réel soit effectué.

De plus, pour garantir la cohérence des modifications des données du runtime, un mutex peut être pris avant l'insertion d'une action dans une stream, par exemple avant l'initialisation d'un transfert. Les temps d'attente augmentent, l'exécution devient plus séquentielle et le temps de gestion des tâches devient lié au temps d'exécution des tâches. En effet, l'exécution des tâches sur GPU est asynchrone avec le code CPU, et la stream d'exécution est synchronisée avec les streams de transfert de données via des event CUDA. La vitesse à laquelle le runtime complète les actions d'une stream est donc liée à la plus lente des streams.

À ma connaissance, la limite sur la taille des streams CUDA à partir de laquelle les actions d'insertion deviennent bloquantes n'est pas spécifiée dans la documentation Nvidia. Pour estimer grossièrement cette valeur, nous avons utilisé le simple code 5.8, où chaque tour de boucle exécute `kernel_sleep`, un kernel qui attend une seconde. Une exécution de ce code va effectuer rapidement les premiers tours de boucle car chaque lancement de `kernel_sleep` correspond simplement à l'insertion d'une action dans la stream, mais une fois que la limite sur la stream sera atteinte, le lancement de `kernel_sleep` durera environ une

seconde ! En détectant cette vitesse d'évolution, nous pouvons en déduire cette limite des streams CUDA.

Nous avons observé une taille de stream de 2048 éléments avec CUDA 11.2 et un driver CUDA 460.91.03. Aucune documentation n'a pu être trouvée concernant le fait si cette valeur peut être modifiée par l'utilisateur ou si celle-ci a évolué entre les différentes versions de CUDA et du driver associé.

```
int main()
{
        int N = 16384;
        cudaStream_t stream;
        cudaStreamCreateWithFlags( &stream, cudaStreamNonBlocking );
        for( int exec_id = 0; exec_id < N; exec_id++ )
        {
                kernel_sleep<<<1,1,0,stream>>>();
                printf("execute␣%5d/%d\r", exec_id, N);
        }
        printf("\n");
        return 0;
}
```

Code 5.8 – Estimation de la taille des streams CUDA

```
for ( int j = 0; j < N/NB; j += 1  )
{
  if( j > 0 )
  {
    uint64_t dim = j*NB;

    size_t   es_a = NB*sizeof(double);
    size_t   ej_a = 0;
    uint64_t ws_a = 1;
    size_t   ss_a = N*sizeof(double);
    size_t   pshift_a = j*NB*sizeof(double);

    size_t   es_b = NB*sizeof(double);
    size_t   ej_b = N*sizeof(double);
    uint64_t ws_b = NB;
    size_t   ss_b = 0;
    size_t   pshift_b = sizeof(double) * (j*NB + j*NB*N);

    setmeminfo( &syrk_mem_infos, 0, 0, &A, es_a, ws_a, ej_a, ss_a, dim,
                SCHED_READ, pshift_a )
    setmeminfo( &syrk_mem_infos, 1, 0, &A, es_b, ws_b, ej_b, ss_b, dim,
                SCHED_READ_WRITE, pshift_b )
    spawn_function( fn_syrk_id, &syrk_common_args,
                sizeof(struct syrk_common_args_t),
                &syrk_mem_infos, &syrk_perf, update_perf );
  }

  {
    size_t es = sizeof(double) * NB;
    size_t ej = sizeof(double) * N;
    uint64_t ws = NB;
    size_t ss = 0;
    size_t pshift = sizeof(double) * (j*NB + j*NB*N);

    setmeminfo( &potrf_mem_infos, 0, 0, &A, es, ws, ej, ss, 1,
                SCHED_READ_WRITE, pshift )
    spawn_function( fn_potrf_id, &potrf_common_args,
                sizeof(struct potrf_common_args_t),
```

```
39                &potrf_mem_infos, &potrf_perf, update_perf );
40  }
41
42  if( j*NB + NB < N )
43  {
44    if( j > 0 )
45    {
46      uint64_t dim = N-(j+1)*NB;
47
48      size_t      es_a = sizeof(double);
49      size_t      ej_a = N*sizeof(double);
50      uint64_t    ws_a = j*NB;
51      size_t      ss_a = sizeof(double);
52      size_t ptshift_a = (j+1)*NB*sizeof(double);
53
54      size_t      es_b = NB*sizeof(double);
55      size_t      ej_b = N*sizeof(double);
56      uint64_t    ws_b = j*NB;
57      size_t      ss_b = 0;
58      size_t ptshift_b = j*NB*sizeof(double);
59
60      size_t      es_c = sizeof(double);
61      size_t      ej_c = N*sizeof(double);
62      uint64_t    ws_c = NB;
63      size_t      ss_c = sizeof(double);
64      size_t ptshift_c = ((j+1)*NB + j*NB*N)*sizeof(double);
65
66      setmeminfo( &gemm_mem_infos, 0, 0, &A, es_a, ws_a, ej_a, ss_a, dim,
67                SCHED_READ, ptshift_a ) // A
68      setmeminfo( &gemm_mem_infos, 1, 0, &A, es_b, ws_b, ej_b, ss_b, dim,
69                SCHED_READ, ptshift_b ) // B
70      setmeminfo( &gemm_mem_infos, 2, 0, &A, es_c, ws_c, ej_c, ss_c, dim,
71                SCHED_READ_WRITE, ptshift_c ) // C
72      spawn_function( fn_gemm_id, &gemm_common_args,
73                sizeof(struct gemm_common_args_t), &gemm_mem_infos,
74                &gemm_perf, update_perf );
75    }
76
77    {
78      uint64_t dim = N - (j+1)*NB;
79
80      size_t   es_a = NB*sizeof(double);
81      size_t   ej_a = N*sizeof(double);
82      uint64_t ws_a = NB;
83      size_t   ss_a = 0;
84      size_t   pshift_a = j*(NB+NB*N)*sizeof(double);
85
86      size_t   es_b = sizeof(double);
87      size_t   ej_b = sizeof(double)*N;
88      uint64_t ws_b = NB;
89      size_t   ss_b = sizeof(double);
90      size_t   pshift_b = ((j+1)*NB+j*NB*N)*sizeof(double);
91
92      setmeminfo( &trsm_mem_infos, 0, 0, &A, es_a, ws_a, ej_a, ss_a, dim,
93                SCHED_READ, pshift_a )
94      setmeminfo( &trsm_mem_infos, 1, 0, &A, es_b, ws_b, ej_b, ss_b, dim,
95                SCHED_READ_WRITE, pshift_b )
96      spawn_function( fn_trsm_id, &trsm_common_args,
```

```
97                  sizeof(struct trsm_common_args_t),
98                  &trsm_mem_infos, &trsm_perf, update_perf );
99      }
100   }
101 }
```

Code 5.9 – Factorisation de cholesky

5.7 Conclusion du chapitre

Ce chapitre nous a permis de montrer notre prototype d'implémentation d'un runtime de gestion des tâches moldables, et en particulier d'une méthode spécifique pour détecter les dépendances entre tâches et sous-tâches générées du fait de la modabilité. Des mesures ont été effectuées sur différents algorithmes, un (basique) algorithme de batch de multiplication de matrices 5.6.1 afin de caractériser la répartition de la charge de travail et les mécanismes de découpe. Cette répartition est proche de celle attendue vis-à-vis des performances théoriques du matériel dès que l'intensité arithmétique s'accroît afin de masquer les communications. De plus, l'algorithme converge rapidement vers un état d'équilibre.

Une implémentation de l'algorithme de formation de voies 5.6.2 a permis d'expérimenter un cas d'application pratique où plusieurs tâches moldables s'enchaînent. Nous avons pu observer une adaptation de l'exécution sur différentes configurations hétérogènes et un gain de performance notable lors de l'augmentation du nombre de ressources à disposition. Cet algorithme a également permis de mesurer des surcoûts d'ordonnancements très faibles de notre implémentation. Les mesures montrent un speedup linéaire des performances en ajoutant plusieurs GPU sans conflit d'accès sur le bus PCIe, permettant sur un serveur bi-GPU d'approcher un speedup de 6 par rapport à une exécution sur 16 cœurs CPU. Sur le serveur DGX-A100 nous avons pu observer un speedup de 4.5 en utilisant 8 GPU, en comparaison avec une exécution sur 64 cœurs.

Finalement, notre code de la factorisation de Cholesky avec tâches moldables a mis en avant certaines faiblesses de notre implémentation du runtime qui n'apparaissaient pas avec les autres codes ou algorithmes que nous avons expérimentés. Nous avons observé des baisses de performances lorsque les tâches accèdent à des données fortement fragmentées et avons identifié un comportement non attendu des streams CUDA. Ces résultats, bien que négatifs du point de vue des performances obtenues, vont nous permettre de rebondir sur quelques perspectives.

Chapitre 6

Conclusion

6.1 Résumé des travaux

Ces travaux ont été motivés par le besoin de porter des algorithmes de traitement SONAR sur des plateformes hétérogènes. En particulier, pour les mettre en œuvre sur des GPUs. L'implémentation de tels algorithmes n'est pas évidente pour plusieurs raisons : les aspects mathématiques et physiques sous-jacents ne sont pas triviaux, ce qui implique que la conception des algorithmes est réalisée par des « numériciens ». De plus, exploiter efficacement un GPU peut demander une maîtrise approfondie du comportement du matériel et des optimisations spécifiques à un GPU, qui ne seront peut-être pas portables et pourraient nécessiter une algorithmique particulière du domaine de l' « informaticien » du HPC.

Pour soulager la complexité des portages, nous avons opté pour la méthodologie suivante : exploiter au maximum les opérations élémentaires disponibles dans les bibliothèques (par exemples BLAS), identifier des optimisations applicables aux chaînes de traitement SONAR, appliquer ces optimisations de manière semi-automatique.

Les travaux de thèse ont commencé par la sélection d'un algorithme cible avec les équipes de numériciens, le choix s'est porté sur l'algorithme de formation de voies classique du fait de ses besoins en puissance de calcul et de sa place centrale dans la majorité des chaînes de traitement SONAR. Une étape d'optimisation manuelle nous a permis d'identifier plusieurs optimisations à appliquer à partir d'une implémentation naïve de l'algorithme et d'identifier des difficultés lors de l'implémentation. Ce code a servi de version de référence pour la suite des travaux. En particulier, le choix du placement des données en mémoire pour éviter les opérations de réordonnancement et la difficulté à implémenter des kernels performants effectuant des opérations non triviales.

Le chapitre 3 a introduit un outil d'analyse statique d'un code C, associé à un jeu d'annotations. Il a pour objectif d'automatiser les optimisations identifiées précédemment afin de les appliquer à d'autres algorithmes écrits par les « numériciens ». Cet outil génère un code C valide qui respecte les dépendances entre les calculs effectués sur le code original. Il se base sur la construction d'un graphe de dépendances entre les différentes fonctions du code en rapport avec les données lues ou écrites par ces fonctions. La génération est faite en parcourant l'arbre syntaxique abstrait du code C initial, les informations sur les nids de boucles sont conservées dans un graphe hiérarchique permettant de retrouver la position de chaque appel de fonction dans la hiérarchie. Ce graphe hiérarchique permet de connaître la structure des nids de boucle et le nombre d'appels de chaque fonction, ces informations étant utiles pour regrouper les appels ensemble et sortir certaines fonctions du nid de boucle. Par modification et annotation de ces graphes, l'outil est capable d'appliquer des modifications basiques telles que la détection de données pré-calculables et le regroupement de fonctions similaires sous forme de batch afin d'exploiter plus efficacement un GPU sur un grand ensemble de petits calculs.

Ces optimisations nous ont permis d'atteindre des performances correctes sur l'algorithme de formation de voies, mais en augmentant l'utilisation de l'espace mémoire sur GPU. Nous atteignons, avec la version générée par l'outil un speedup de 5 par rapport à la version non optimisée. Néanmoins, une optimisation manuelle consistant à fusionner des multiplications de matrice n'a pas été intégrée à l'outil

car spécifique au produit matriciel uniquement. Combinée avec les autres optimisations, elle permet un speedup de 10 par rapport à la version non optimisée. Ces différentes modifications nécessitent d'augmenter les allocations mémoire de 15 à 75%, par rapport à la version non optimisée, selon les optimisations appliquées. En raison de cette consommation mémoire excessive, il était impossible de traiter les instances de formation de voies.

Dans un second temps, nous avons voulu étendre les limites de cet outil afin de lever ces contraintes mémoires pour traiter des instances plus importantes, mais aussi de pouvoir exploiter les architectures multi-GPU afin de gagner en temps d'exécution tout en ayant davantage d'espace mémoire sur les GPUs. L'objectif est d'adapter finement l'exécution aux ressources disponibles (GPU et mémoire disponible). La méthodologie a été, d'une part, de se baser sur une implémentation du concept de tâches moldables tout en conservant les bonnes propriétés de la connaissance des dépendances et, d'autre part, de concevoir un support exécutif ou runtime adapté.

Le chapitre 4 propose d'utiliser le concept de tâches moldables pour permettre d'adapter la granularité des tâches effectuant les calculs aux ressources, possiblement hybrides et hétérogènes en puissance, puis, grâce à cette moldabilité, de sur-décomposer une tâche afin de limiter l'utilisation mémoire nécessaire au calcul. Nous avons décliné ce concept dans le monde OpenMP où nous avons proposé une directive et un ensemble de clauses pour étendre simplement le modèle de tâche OpenMP afin de programmer avec des tâches moldables. Cette directive « OpenMP » `taskmoldable`, fonctionne avec des clauses permettant de définir les données accédées régulièrement par chacune des sous-tâches dans l'objectif de supporter les dépendances de données directement entre les sous-tâches générées. Le principal intérêt pratique de cette directive par rapport à une directive de génération de tâches classique tels que `taskloop` est qu'elle permet de générer des tâches à partir de n'importe quel bloc d'instructions, en particulier nous avons montré comment paralléliser des fonctions BLAS tels que `gemm` ou `gemmBatched` sans avoir accès à l'implémentation des fonctions ! Nous avons aussi illustré cette directive par les exemples d'un algorithme de formation de voies et un algorithme de décomposition de Cholesky.

Le dernier chapitre de cette thèse 5 a consisté à concevoir l'implémentation d'un runtime d'ordonnancement de tâches moldables supportant une exécution hétérogène. Le modèle de découpe de ces tâches est associé à un modèle d'accès aux données qui permet de détecter des potentielles collisions entre deux tâches en temps constant. Cet algorithme de détection des dépendances effectue des approximations qui permettent un calcul rapide des dépendances sans avoir de faux-négatifs et sans introduire, en pratique, plus de dépendances que nécessaires. Ce support exécutif permet d'adapter la répartition de la charge de calcul sur des ressources hybrides (CPU ou GPU) et/ou hétérogènes (puissance de calcul différente et non homogène). La méthode, itérative, se base sur une distribution initiale de la charge d'une tâche qui est ensuite affinée en mesurant les performances des itérations de calcul précédentes pour estimer la répartition des itérations suivantes. Cette méthode tire parti du fait que les algorithmes de traitement SONAR sont appliqués de manière itérative sur un flux de données.

Le runtime a été utilisé pour implémenter un algorithme de formation de voies sur cible multi-CPU/multi-GPU. Cela a permis de valider le fonctionnement global du runtime et de la méthode d'estimation des performances qui dirige la découpe. Les mesures effectuées sur différentes architectures ont permis de montrer que la répartition du travail évoluait bien pour exploiter l'ensemble des co-processeurs à disposition, malgré des limitations dues à une sur-utilisation du bus PCIe.

Les expérimentations de l'algorithme de décomposition de Cholesky, présenté au chapitre précédent, ont mis en avant de nombreuses limitations du runtime au niveau des performances. La principale étant l'implémentation de la gestion des données qui manipule des structures de données trop coûteuses lorsque les zones mémoires accédées par une tâche sont fortement fragmentées.

6.2 Perspectives

Les résultats de ces travaux mènent à de nombreuses perspectives.

Intégration dans un autre runtime

Sur l'ensemble des travaux présentés dans les chapitres précédents, deux points importants seraient à intégrer dans un autre runtime, comme par exemple le runtime OpenMP, pour assurer une certaine pérennité dans le temps. Le premier point concerne le modèle de tâche moldable qui pourrait être intégré afin d'unifier, voire de simplifier, des approches proposées précédemment telles que les constructions `parallel for` ou `taskloop`, tout en étendant l'application de ces constructions à des appels de fonction de bibliothèque dont le code n'est pas accessible (par exemple le `gemm` du chapitre 4).

Le second point concernerait l'intégration du calcul de dépendances tel que présenté dans ce manuscrit qui permet in fine de calculer simplement des dépendances en sous-tâches de tâches moldables mais qui pourrait aussi directement être exploité dans le modèle actuel des tâches OpenMP pour étendre la capacité à décrire des dépendances entre des régions mémoires. Ce travail nécessitera des efforts importants d'implémentations pour adapter le système de gestion mémoire et des dépendances du runtime cible.

Comparaison avec d'autres approches

L'utilisation d'un modèle de tâche moldable nécessite d'être comparé à d'autres runtimes. En particulier, les approches qui visent les mêmes objectifs de réduction de granularité, tels que les tâches parallèles [15] et les tâches hiérarchiques [44] de StarPU. Mais aussi des implémentations plus standards à base de tâches avec OpenMP ou XKaapi.

D'autres méthodes visent à limiter les surcoûts du runtime CUDA tels que cuda-graphs [34], dans le cas d'algorithmes itératifs.

Exploration des stratégies de découpe

L'implémentation actuelle du runtime utilise une méthode basique pour découper le travail d'une tâche moldable. Notons que cet aspect a été laissé de côté dans le chapitre 4 relatif à OpenMP, en revanche l'implémentation du runtime présenté au chapitre 5 découpe les tâches en fonction du nombre de workers utilisés et de la puissance estimée de chacun des workers.

Les nombreux travaux théoriques sur les tâches moldables proposent des algorithmes de répartition de complexités différentes pour différents cas d'utilisation : optimisation énergétique, minimisation du temps d'exécution. Une implémentation de ces heuristiques dans un runtime de tâche moldable permettrait des expérimentations réelles afin de valider ou infirmer expérimentalement certaines hypothèses sous-jacentes ou les algorithmes proposés.

De même, les travaux de cette thèse ont insisté sur la définition des zones mémoires impactées par la tâche moldable et par les sous-tâches générées. L'exploration d'algorithmes de découpe visant à limiter les transferts mémoires serait pertinent, en particulier ils pourraient éviter certains problèmes soulevés par l'implémentation de la décomposition de Cholesky.

Equilibrage de la charge et vol de tâches

La méthode d'équilibrage utilisée dans le runtime repose sur le fait que les algorithmes de traitement SONAR sont itératifs et répètent la même suite d'instructions en boucle. Cette méthode n'est pas générique et risque de conduire à des mauvais équilibrages dans d'autres cas.

Une solution abondamment utilisée dans Cilk ou XKaapi est de laisser les workers inactifs voler des tâches en attente des autres workers. Cette méthode est appelée le vol de tâches. Une transposition directe serait possible dans le cas des tâches moldables, menant potentiellement à des problèmes si, par exemple, un worker CPU vole une tâche d'un worker GPU d'une granularité trop importante. Il serait alors possible d'explorer des heuristiques originales où l'algorithme de vol de tâche prend avantage de la moldabilité pour voler qu'une fraction de tâches en fonction des caractéristiques du worker récupérant la tâche.

Revisiter la représentation mémoire

L'implémentation de la factorisation de Cholesky a mis en avant des problèmes de performances liés à la gestion des segments mémoires. Bien sûr, une implémentation plus minutieuse, en particulier des transferts de données vers le GPU, pourrait permettre de limiter ces problèmes. Néanmoins, les pertes de performances découlent principalement de la représentation de la mémoire en segments continus, ce qui induit une explosion du nombre d'objets à manipuler dans certains cas.

Explorer d'autres représentations, par exemple basées sur des tableaux multi-dimensionnels, pourrait limiter fortement le nombre d'objets nécessaires et permettre de réduire les surcoûts du runtime.

Liaisons des outils et simplifications

La thèse se déroulant dans un contexte appliqué, industriel, je me dois d'évoquer une perspective purement technique. L'outil d'optimisation de code statique, présenté en chapitre 3, a pour vocation de générer une entrée directement compatible avec une exécution de tâche moldable. Soit en générant un code annoté avec les pragma OpenMP, soit un code utilisant un runtime de tâche moldable. Cette liaison entre les deux outils permettrait à l'utilisateur initial de ne pas avoir à manipuler les syntaxes des pragma OpenMP ou des `memory_layout` qui sont difficiles à prendre en mains.

Réflexion sur l'utilisation des modèles par tâches

L'utilisation de tâches à granularité variable nécessite un travail important de la part du développeur pour définir les zones mémoires utilisées et les dépendances. Or, une grande partie des utilisateurs de nœuds HPC, qui pourraient profiter de la mise en œuvre de ces modèles, ne sont pas spécialistes des sciences informatiques mais plutôt du domaine scientifique lié à l'application.

Démocratiser l'utilisation de tels modèles pourrait passer par l'implémentation moldable ou hiérarchique de bibliothèques reprenant les fonctions standard (BLAS, LAPACK, ...) par-dessus un runtime commun. D'une manière similaire au support des FFT de StarPU [25], ou à l'implémentation XKBLAS [30] de fonctions BLAS utilisant XKaapi. Les utilisateurs non spécialistes seraient alors en mesure d'utiliser les API de manière similaire à une implémentation séquentielle tout en ayant la scalabilité apportée par le runtime d'ordonnancement de tâches.

Néanmoins, l'implémentation et le maintien de telles bibliothèques nécessitent d'importants efforts, en particulier pour prendre en charge une grande variété de matériels.

Références

[1] *AMD CDNA3 Architecture*. https://www.amd.com/content/dam/amd/en/documents/instinct-tech-docs/white-papers/amd-cdna-3-white-paper.pdf. Accessed : 7 December, 2023. 2023.

[2] Julien Demouth Andrew Kerr Duane Merrill et John Tran. *CUTLASS : Fast Linear Algebra in CUDA C++*. https://developer.nvidia.com/blog/cutlass-linear-algebra-cuda/. Accessed : 7 December, 2023. 2018.

[3] Cédric Augonnet, Samuel Thibault et Raymond Namyst. "Automatic calibration of performance models on heterogeneous multicore architectures". In : *Euro-Par 2009–Parallel Processing Workshops : HPPC, HeteroPar, PROPER, ROIA, UNICORE, VHPC, Delft, The Netherlands, August 25-28, 2009, Revised Selected Papers 15*. Springer. 2010, p. 56-65.

[4] Cédric Augonnet et al. "StarPU : a unified platform for task scheduling on heterogeneous multicore architectures". In : *European Conference on Parallel Processing*. Springer. 2009, p. 863-874.

[5] Daniel Balouek et al. "Adding Virtualization Capabilities to the Grid'5000 Testbed". In : *Cloud Computing and Services Science*. Sous la dir. d'Ivan I. Ivanov et al. T. 367. Communications in Computer and Information Science. Springer International Publishing, 2013, p. 3-20. isbn : 978-3-319-04518-4. doi : 10.1007/978-3-319-04519-1_1.

[6] Francesco Baralli et al. "GPU-based real-time synthetic aperture sonar processing on-board autonomous underwater vehicles". In : *2013 MTS/IEEE OCEANS-Bergen*. IEEE. 2013, p. 1-8.

[7] Anne Benoit et al. "Online scheduling of moldable task graphs under common speedup models". In : *Proceedings of the 51st International Conference on Parallel Processing*. 2022, p. 1-11.

[8] Anne Benoit et al. "Shelf schedules for independent moldable tasks to minimize the energy consumption". In : *2021 IEEE 33rd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE. 2021, p. 126-136.

[9] Richard S. Bird. "An Introduction to the Theory of Lists". In : *Logic of Programming and Calculi of Discrete Design*. Sous la dir. de Manfred Broy. Berlin, Heidelberg : Springer Berlin Heidelberg, 1987, p. 5-42. isbn : 978-3-642-87374-4.

[10] Raphaël Bleuse et al. "Scheduling independent moldable tasks on multi-cores with GPUs". In : *IEEE Transactions on Parallel and Distributed Systems* 28.9 (2017), p. 2689-2702.

[11] Robert D Blumofe et Charles E Leiserson. "Scheduling multithreaded computations by work stealing". In : *Journal of the ACM (JACM)* 46.5 (1999), p. 720-748.

[12] Javier Bueno et al. "Implementing OmpSs Support for Regions of Data in Architectures with Multiple Address Spaces". In : *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*. ICS '13. Eugene, Oregon, USA : Association for Computing Machinery, 2013, p. 359-368. isbn : 9781450321303.

[13] Jo Inge Buskenes et al. "An optimized GPU implementation of the MVDR beamformer for active sonar imaging". In : *IEEE Journal of Oceanic Engineering* 40.2 (2014), p. 441-451.

[14] Yih-Fam Chen, Emden R Gansner et Eleftherios Koutsofios. "A C++ data model supporting reachability analysis and dead code detection". In : *IEEE Transactions on Software Engineering* 24.9 (1998), p. 682-694.

[15] Terry Cojean. "Programmation des architectures hétérogènes à l'aide de tâches divisibles ou modulables". Thèse de doct. Bordeaux, 2018.

[16] Terry Cojean et al. "Resource aggregation for task-based cholesky factorization on top of modern architectures". In : *Parallel Computing* 83 (2019), p. 73-92.

[17] LAPACK Contributors. *potrf : triangular factor*. https://netlib.org/lapack/explore-html/d2/d09/group__potrf.html. Accessed : 7 December, 2023. 2023.

[18] *CUDA Toolkit*. https://developer.nvidia.com/cuda-toolkit. Accessed : 7 December, 2023. 2009.

[19] Jack Dongarra et al. "A proposed API for batched basic linear algebra subprograms". In : (2016).

[20] Jack Dongarra et al. "Batched BLAS (basic linear algebra subprograms) 2018 specification". In : (2018).

[21] Pierre-François Dutot. "Hierarchical scheduling for moldable tasks". In : *Euro-Par 2005 Parallel Processing : 11th International Euro-Par Conference, Lisbon, Portugal, August 30-September 2, 2005. Proceedings 11*. Springer. 2005, p. 302-311.

[22] Mathieu Faverge et al. "Programming heterogeneous architectures using hierarchical tasks". In : *European Conference on Parallel Processing*. Springer. 2022, p. 97-108.

[23] Mathieu Faverge et al. "Programming heterogeneous architectures using hierarchical tasks". In : *Concurrency and Computation : Practice and Experience* 35.25 (2023), e7811.

[24] Dror G Feitelson et Larry Rudolph. "Toward convergence in job schedulers for parallel supercomputers". In : *Workshop on job scheduling strategies for parallel processing*. Springer. 1996, p. 1-26.

[25] *FFT Support in starPU*. `https://files.inria.fr/starpu/doc/html/FFTSupport.html`. Accessed : 7 December, 2023.

[26] Matteo Frigo, Charles E Leiserson et Keith H Randall. "The implementation of the Cilk-5 multithreaded language". In : *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*. 1998, p. 212-223.

[27] Daniel R Fuhrmann et Geoffrey San Antonio. "Transmit beamforming for MIMO radar systems using partial signal correlation". In : *Conference Record of the Thirty-Eighth Asilomar Conference on Signals, Systems and Computers, 2004.* T. 1. IEEE. 2004, p. 295-299.

[28] François Galilée et al. "Athapascan-1 : On-line building data flow graph in a parallel language". In : *Proceedings. 1998 International Conference on Parallel Architectures and Compilation Techniques (Cat. No. 98EX192)*. IEEE. 1998, p. 88-95.

[29] Thierry Gautier, Xavier Besseron et Laurent Pigeon. "KAAPI : A Thread Scheduling Runtime System for Data Flow Computations on Cluster of Multi-Processors". In : *Proceedings of the 2007 International Workshop on Parallel Symbolic Computation*. PASCO '07. London, Ontario, Canada : Association for Computing Machinery, 2007, p. 15-23. isbn : 9781595937414.

[30] Thierry Gautier et Joao VF Lima. "Xkblas : a high performance implementation of blas-3 kernels on multi-gpu server". In : *2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. IEEE. 2020, p. 1-8.

[31] Thierry Gautier et al. "Xkaapi : A runtime system for data-flow task programming on heterogeneous architectures". In : *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IEEE. 2013, p. 1299-1308.

[32] Isaac D Gerg et al. "GPU acceleration for synthetic aperture sonar image reconstruction". In : *Global Oceans 2020 : Singapore–US Gulf Coast*. IEEE. 2020, p. 1-9.

[33] Paul Graham et Brent Nelson. "FPGA-based sonar processing". In : *proceedings of the 1998 ACM/SIGDA sixth international symposium on field programmable gate arrays*. 1998, p. 201-208.

[34] Alan Gray. *Getting Started with CUDA Graphs*. `https://developer.nvidia.com/blog/cuda-graphs/`. Accessed : 7 December, 2023. 2019.

[35] Ana Maria G Guerreiro, Adriio Duarte D Neto et FA Lisboa. "Beamforming applied to an adaptive planar array". In : *Proceedings RAWCON 98. 1998 IEEE Radio and Wireless Conference (Cat. No. 98EX194)*. IEEE. 1998, p. 209-212.

[36] Azzam Haidar et al. "Framework for Batched and GPU-resident Factorization Algorithms to Block Householder Transformations". In : *ISC High Performance*. Springer. Frankfurt, Germany : Springer, juil. 2015.

[37] Troels Henriksen. *List Homomorphisms and Parallelism*. `https://sigkill.dk/writings/par/lhomo.html`.

[38] *HIP : C++ Heterogeneous-Compute Interface for Portability*. `https://github.com/ROCm/HIP`. Accessed : 7 December, 2023. 2016.

[39] Laxmikant V Kale et Gengbin Zheng. "Charm++ and AMPI : Adaptive runtime strategies via migratable objects". In : *Advanced Computational Infrastructures for Parallel and Distributed Applications* (2009), p. 265-282.

[40] Jakub Kurzak et al. "Scheduling dense linear algebra operations on multicore processors". In : *Concurrency and Computation : Practice and Experience* 22.1 (2010), p. 15-44.

[41] Chris Lattner et Vikram Adve. "LLVM : A compilation framework for lifelong program analysis & transformation". In : *International symposium on code generation and optimization, 2004. CGO 2004.* IEEE. 2004, p. 75-86.

[42] Ang Li et al. "Evaluating modern gpu interconnect : Pcie, nvlink, nv-sli, nvswitch and gpudirect". In : *IEEE Transactions on Parallel and Distributed Systems* 31.1 (2019), p. 94-110.

[43] Wei Liu et Stephan Weiss. *Wideband beamforming : concepts and techniques*. John Wiley & Sons, 2010.

[44] Gwenolé Lucas. "On the use of hierarchical task for heterogeneous architectures". Thèse de doct. Université de Bordeaux, 2023.

[45] Marcos Maroñas, Xavier Teruel et Vicenç Beltran. "OpenMP taskloop dependences". In : *OpenMP : Enabling Massive Node-Level Parallelism : 17th International Workshop on OpenMP, IWOMP 2021, Bristol, UK, September 14–16, 2021, Proceedings 17*. Springer. 2021, p. 50-64.

[46] Nenad Mijić et Davor Davidović. "Batched matrix operations on distributed GPUs with application in theoretical physics". In : *2022 45th Jubilee International Convention on Information, Communication and Electronic Technology (MIPRO)*. IEEE. 2022, p. 293-299.

[47] Carl-Inge Colombo Nilsen et Ines Hafizovic. "Digital beamforming using a GPU". In : *2009 IEEE International Conference on Acoustics, Speech and Signal Processing*. IEEE. 2009, p. 609-612.

[48] *NVIDIA A100 Tensor Core GPU Architecture*. `https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf`. Accessed : 7 December, 2023. 2021.

[49] *NVIDIA Grace Hopper Superchip Architecture Whitepaper*. `https://resources.nvidia.com/en-us-grace-cpu/nvidia-grace-hopper`. Accessed : 7 December, 2023. 2023.

[50] *Open Standard for Parallel Programming of Heterogeneous Systems*. `https://www.khronos.org/opencl/`. Accessed : 7 December, 2023. 2009.

[51] Yunheung Paek, Jay Hoeflinger et David Padua. "Efficient and Precise Array Access Analysis". In : *ACM Trans. Program. Lang. Syst.* 24.1 (jan. 2002), p. 65-109. issn : 0164-0925.

[52] Carl Pearson. "Interconnect Bandwidth Heterogeneity on AMD MI250x and Infinity Fabric". In : *arXiv preprint arXiv :2302.14827* (2023).

[53] Romain Pereira et al. "Investigating Dependency Graph Discovery Impact on Task-based MPI+ OpenMP Applications Performances". In : *Proceedings of the 52nd International Conference on Parallel Processing*. 2023, p. 163-172.

[54] Josep M. Perez et al. "Improving the Integration of Task Nesting and Dependencies in OpenMP". In : *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2017, p. 809-818.

[55] Lucas Perotin et Hongyang Sun. "Improved Online Scheduling of Moldable Task Graphs under Common Speedup Models". In : *arXiv preprint arXiv :2304.14127* (2023).

[56] Jean-Louis Roch, Daouda Traoré et Julien Bernard. "On-Line Adaptive Parallel Prefix Computation". In : *Euro-Par 2006, Parallel Processing, 12th International Euro-Par Conference, Dresden, Germany, August 28 - September 1, 2006, Proceedings*. Sous la dir. de Wolfgang E. Nagel, Wolfgang V. Walter et Wolfgang Lehner. T. 4128. Lecture Notes in Computer Science. Springer, 2006, p. 841-850. url : `https://doi.org/10.1007/11823285%5C_88`.

[57] Tarek Sallam et Ahmed M Attiya. "Convolutional neural network for 2D adaptive beamforming of phased array antennas with robustness to array imperfections". In : *International Journal of Microwave and Wireless Technologies* 13.10 (2021), p. 1096-1102.

[58] Erik Saule, Doruk Bozdağ et Ümit V Çatalyürek. "Optimizing the stretch of independent tasks on a cluster : From sequential tasks to moldable tasks". In : *Journal of Parallel and Distributed Computing* 72.4 (2012), p. 489-503.

[59] Samuel D Somasundaram. "Wideband robust capon beamforming for passive sonar". In : *IEEE Journal of Oceanic Engineering* 38.2 (2012), p. 308-322.

[60] Matthias Bo Stuart, Mikkel Schou et Jørgen Arendt Jensen. "Row-column beamforming with dynamic apodizations on a GPU". In : *Medical Imaging 2019 : Ultrasonic Imaging and Tomography*. T. 10955. SPIE. 2019, p. 169-175.

[61] Xavier Teruel et al. "A proposal for task-generating loops in OpenMP". In : *OpenMP in the Era of Low Power Devices and Accelerators : 9th International Workshop on OpenMP, IWOMP 2013, Canberra, ACT, Australia, September 16-18, 2013. Proceedings 9*. Springer. 2013, p. 1-14.

[62] Haowen Tian et al. "Design and implementation of a real-time multi-beam sonar system based on fpga and dsp". In : *Sensors* 21.4 (2021), p. 1425.

[63] Daouda Traoré. "Algorithmes parallèles auto-adaptatifs et applications. (Self-adaptive parallel algorithms and applications)". Thèse de doct. Grenoble Institute of Technology, France, 2008. URL : https://tel.archives-ouvertes.fr/tel-00353274.

[64] Daouda Traoré et al. "Deque-Free Work-Optimal Parallel STL Algorithms". In : *Euro-Par 2008 - Parallel Processing, 14th International Euro-Par Conference, Las Palmas de Gran Canaria, Spain, August 26-29, 2008, Proceedings*. Sous la dir. d'Emilio Luque, Tomàs Margalef et Domingo Benitez. T. 5168. Lecture Notes in Computer Science. Springer, 2008, p. 887-897. DOI : 10.1007/978-3-540-85451-7_95. URL : https://doi.org/10.1007/978-3-540-85451-7%5C_95.

[65] Wei Wu et al. "Hierarchical dag scheduling for hybrid distributed systems". In : *2015 IEEE International Parallel and Distributed Processing Symposium*. IEEE. 2015, p. 156-165.

[66] Xiaohu Wu et Patrick Loiseau. "Efficient approximation algorithms for scheduling moldable tasks". In : *European Journal of Operational Research* 310.1 (2023), p. 71-83.

[67] Xun Wu et al. "Fast Wideband Beamforming Using Convolutional Neural Network". In : *Remote Sensing* 15.3 (2023), p. 712.

[68] Deshi Ye, Danny Z Chen et Guochuan Zhang. "Online scheduling of moldable parallel tasks". In : *Journal of Scheduling* 21.6 (2018), p. 647-654.