PER2024-004 - Étude des calculs GP-GPU sur cartes graphiques

Maurois Quentin[1] et Soto Julien[1]

Polytech Nice Sophia, 930 Rte des Colles, 06410 Biot, France
quentin.maurois@etu.unice.fr
julien.soto@etu.unice.fr

Encadrant, Inria : Sid Touati
sid.touati@inria.fr

Abstract. L'optimisation des performances de calcul est cruciale en intelligence artificielle, calcul scientifique et rendu 3D. Cette étude compare les performances des CPU et GPU sur des calculs intensifs à l'aide de CUDA et OpenCL.
Des benchmarks sur une NVIDIA GTX 1050 Ti et un Intel Core i7-8750H ont évalué plusieurs opérations matricielles. Les résultats montrent que le GPU surpasse le CPU pour les calculs massivement parallélisables, avec une accélération atteignant un facteur 30.3 pour des matrices 1024×1024 et même 7457 pour des matrices 8192x8192. Toutefois, les transferts RAM ↔ VRAM limitent l'efficacité sur des tâches où l'accès mémoire n'est pas prédominant.
Bien que CUDA soit mieux optimisé pour les GPU NVIDIA, OpenCL offre une plus grande portabilité au prix d'une complexité accrue. Ces observations soulignent l'importance d'optimiser les transferts de mémoire et de choisir le framework en fonction des contraintes matérielles et logicielles.

1 Introduction

L'optimisation des performances de calcul est un enjeu crucial dans des domaines variés tels que l'intelligence artificielle, le calcul scientifique et le rendu 3D. Ces disciplines nécessitent l'exécution rapide et efficace d'opérations mathématiques complexes, souvent sur de grands ensembles de données. Dans ce contexte, les processeurs graphiques (GPU) se sont imposés comme une alternative performante aux processeurs traditionnels (CPU) en raison de leur capacité à exécuter un grand nombre d'opérations en parallèle.

Les architectures CPU et GPU ont été conçues avec des philosophies différentes. Le CPU, optimisé pour le traitement séquentiel et la gestion des tâches

générales d'un système, possède un nombre limité de cœurs puissants. En revanche, le GPU est composé de milliers de cœurs plus simples, spécialisés dans l'exécution parallèle massive, ce qui le rend particulièrement adapté aux calculs matriciels et aux tâches massivement parallélisables.

Dans cette étude, nous comparons les performances des CPU et GPU dans le cadre d'opérations intensives en utilisant deux frameworks majeurs : CUDA, développé par NVIDIA, et OpenCL, une alternative plus généraliste et multiplateforme. Des benchmarks ont été réalisés sur une NVIDIA GTX 1050 Ti et un Intel Core i7-8750H afin d'évaluer leurs performances respectives sur plusieurs opérations matricielles.

L'objectif de cette analyse est double :

- Quantifier les gains de performance qu'apporte le calcul sur GPU par rapport au CPU, en mesurant l'accélération obtenue sur différents types d'opérations.
- Identifier les limites liées à l'utilisation des GPU, notamment en ce qui concerne les transferts de mémoire entre la RAM et la VRAM, qui peuvent avoir un impact significatif sur les performances globales.

Les résultats obtenus mettent en évidence l'avantage des GPU sur les calculs massivement parallélisables, avec une accélération pouvant atteindre un facteur 30.3 pour des matrices de 1024×1024, et jusqu'à 7457 pour des matrices de 8192×8192. Cependant, ces performances sont conditionnées par une gestion optimisée des transferts mémoire, un facteur déterminant dans le choix entre un calcul CPU ou GPU.

Cette étude vise ainsi à fournir des recommandations sur l'utilisation optimale des GPU dans des contextes de calcul intensif et à éclairer les choix technologiques en fonction des contraintes matérielles et logicielles.

2 Étude de l'existant

2.1 CPU et GPU

Dans le domaine de l'informatique, le processeur est un élément clé qui exécute les instructions des programmes. Deux types principaux de processeurs jouent un rôle essentiel dans le traitement des données et l'exécution des calculs :

- L'unité centrale de traitement (CPU - Central Processing Unit) : C'est le cœur du calcul général d'un ordinateur. Il est conçu pour traiter une grande variété de tâches avec une gestion optimisée des instructions et des processus.
- L'unité de traitement graphique (GPU - Graphics Processing Unit) : À l'origine, elle a été développée pour l'affichage et le rendu graphique, mais elle est aujourd'hui utilisée pour des calculs parallèles intensifs dans des domaines tels que l'intelligence artificielle, la modélisation scientifique et le traitement d'images.

Les CPU et les GPU diffèrent fondamentalement dans leur architecture et leur optimisation pour différents types de calculs.

CPU Les CPU sont conçus pour le traitement séquentiel. Elles comportent généralement un nombre restreint de cœurs puissants capables de gérer une variété de tâches complexes. Cette architecture est idéale pour des opérations nécessitant des calculs séquentiels rapides et une gestion efficace des tâches multiples 1.

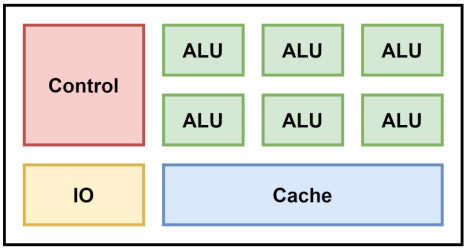

Fig. 1. Architecture d'un CPU [9]

GPU À l'inverse, les GPU sont optimisées pour le parallélisme massif. Une unité de traitement graphique possède des milliers de cœurs plus simples, capables d'exécuter simultanément la même opération sur un grand nombre de données. Cette conception est particulièrement efficace pour des tâches hautement parallélisables, comme le rendu graphique et les calculs scientifiques intensifs.

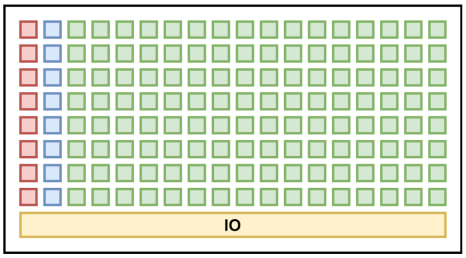

Fig. 2. Architecture d'un GPU[9]

Les GPU excellent dans les opérations matricielles, un élément central de l'apprentissage profond et du calcul scientifique, grâce à leur architecture optimisée pour l'exécution parallèle. Leur capacité à effectuer des multiplications matricielles en parallèle leur permet souvent de surpasser les CPU en termes de performances pour ces tâches spécifiques [8].

Dans le domaine de l'apprentissage profond, les GPU jouent un rôle crucial tant pour l'entraînement que pour l'inférence des modèles. Elles sont idéales

pour traiter simultanément de vastes ensembles de données, ce qui réduit considérablement les temps de calcul pour des réseaux neuronaux complexes. En revanche, les CPU conservent leur importance pour des tâches nécessitant une logique séquentielle ou une gestion complexe, comme le prétraitement des données et la coordination des processus.

Lorsqu'on compare les performances des GPU et des CPU pour l'entraînement des réseaux neuronaux, les GPU offrent un avantage significatif en termes de rapidité, particulièrement pour les réseaux profonds et complexes. Cependant, pour des réseaux plus simples ou des tâches moins intensives, l'écart de performances entre GPU et CPU peut être moins prononcé.

Malgré leurs avantages en matière de parallélisme, les GPU consomment généralement plus d'énergie que les CPU. Ce facteur rend essentielle l'évaluation du rapport performance/consommation énergétique lors du choix d'une architecture pour une application spécifique. Un équilibre entre efficacité énergétique et performances est déterminant pour optimiser les ressources dans des projets de calcul intensif [7].

2.2 Constructeurs GPU

Il existe trois grands constructeurs de GPU : Intel, AMD et NVIDIA. Chacun propose des solutions adaptées à différents besoins, allant de l'intégration dans des systèmes grand public aux solutions haut de gamme pour les professionnels et les joueurs. Parmi eux, NVIDIA se distingue particulièrement par ses innovations et ses performances, ce qui en fait un acteur majeur du marché des cartes graphiques.

NVIDIA NVIDIA Corporation a été fondée en 1993 par Jensen Huang, Chris Malachowsky et Curtis Priem, avec pour objectif de révolutionner le traitement graphique [1]. Sa première carte, la NV1, bien que novatrice, n'a pas rencontré le succès escompté, mais a jeté les bases de futures innovations.

En 1999, NVIDIA introduit le concept de GPU avec la GeForce 256, intégrant des fonctionnalités avancées qui ont redéfini l'industrie. Depuis, les GPU NVIDIA sont devenus essentiels non seulement pour le jeu, mais aussi pour la modélisation 3D, la visualisation scientifique et l'intelligence artificielle [12].

Architectures GPU Depuis sa création, NVIDIA a développé plusieurs architectures GPU marquantes [4] [13] :

- Tesla (2006) : Introduction de CUDA, ouvrant la voie aux calculs parallèles massifs [10].
- Fermi (2010) : Amélioration du parallélisme et de la précision des calculs.
- Kepler (2012) et Maxwell (2014) : Optimisation de l'efficacité énergétique et du rendu graphique [3].
- Pascal (2016) : Première architecture gravée en 16 nm, optimisée pour la VR [14].

- Volta (2017) : Introduction des Tensor Cores pour l'IA [14].
- Turing (2018) : Ray tracing en temps réel et DLSS [14].
- Ampere (2020) : Doublement des performances graphiques et IA [14].
- Ada Lovelace (2022) : Dernière avancée avec des Tensor Cores de 4e génération et un ray tracing amélioré [4].

Les recherches que nous avons effectuées en amont sur l'évolution des GPU se sont principalement appuyées sur les listes des cartes graphiques développées par NVIDIA [15]. Cette exploration a permis de mettre en lumière les différents types d'architectures ayant marqué l'histoire des GPU NVIDIA, ainsi que l'amélioration progressive des performances et des concepts technologiques associés.

Les Figures 3, 4 et 5 que nous avons réalisées à partir de l'analyse des données compilées dans ces listes, illustrent ces évolutions. Figure 3 compare les horloges mémoires et de cœurs entre différents GPU NVIDIA, tandis que Figure 4 présente la taille de mémoire moyenne et Figure 5 les performances moyennes en FP32 selon l'architecture GPU.

En analysant ces listes, une logique dans la nomenclature des modèles a également pu être déterminée. Les numéros associés aux modèles servent à introduire de nouvelles technologies ou architectures, signalant une avancée significative dans la gamme. Les lettres jointes, telles que TI, RTX, GT, GTX ou TITAN, apportent des précisions sur les performances ou les applications spécifiques.

Par exemple, les modèles RTX indiquent la prise en charge du ray tracing tandis que TITAN cible les applications professionnelles nécessitant une puissance extrême. Cette structure de dénomination reflète une hiérarchie claire, facilitant la compréhension des caractéristiques et des usages pour les consommateurs et les professionnels.

Grâce à l'évolution technologique, l'augmentation du nombre de transistors et des cœurs CUDA, NVIDIA a transformé le GPU en un puissant outil de calcul parallèle, propulsant les performances graphiques et les applications d'intelligence artificielle [2].

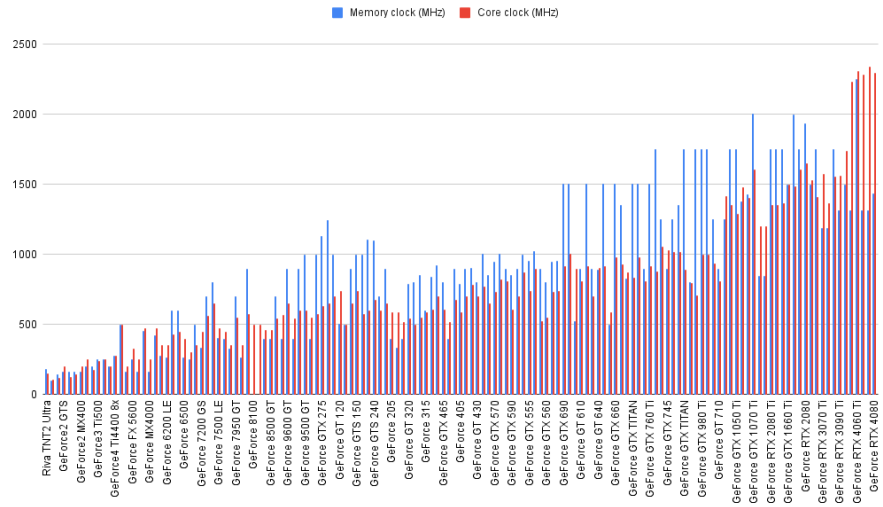

Fig. 3. Comparaison des horloges mémoires et de cœurs entre différents GPU de NVIDIA

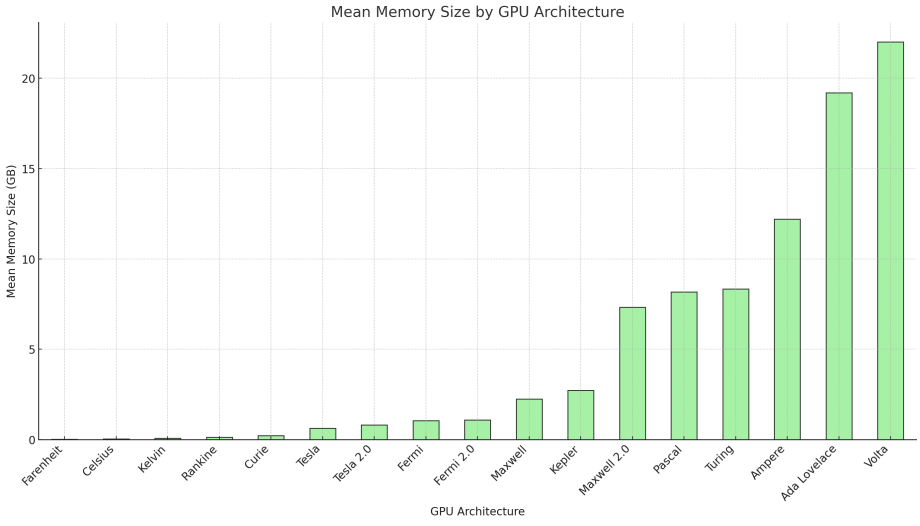

Fig. 4. Taille de mémoire moyenne selon l'architecture GPU

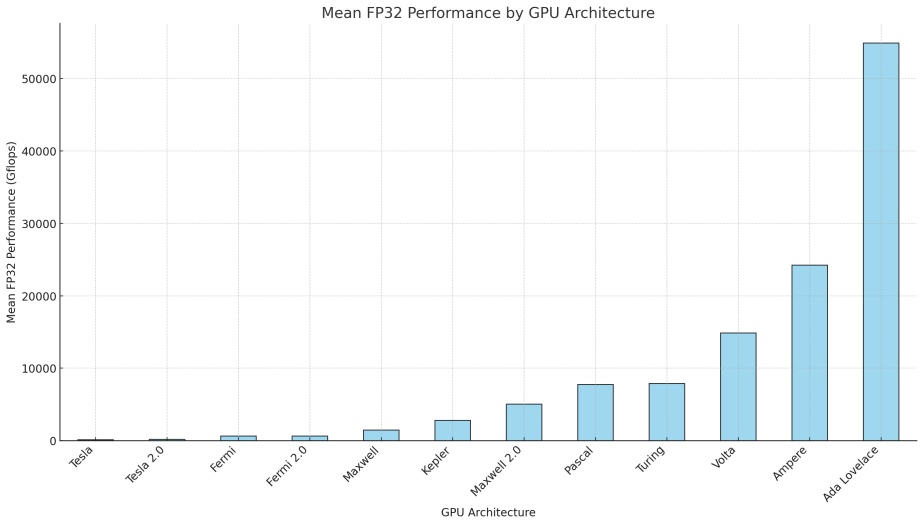

Fig. 5. Performance moyenne FP32 selon l'architecture GPU

Cette évolution a également conduit au développement de frameworks de programmation spécifiques permettant d'exploiter pleinement la puissance des GPU. Parmi eux, CUDA et OpenCL se distinguent comme deux solutions majeures.

2.3 CUDA et OpenCL

CUDA CUDA est l'API propriétaire développée par NVIDIA pour la programmation parallèle sur GPU. Elle se limite aux cartes de la marque et est conçue pour tirer parti de leur architecture au travers de différents outils et bibliothèques dédiés. Cette API est utilisée dans différents domaines tels que le calcul scientifique, le deep learning ou encore les simulations physiques [11].

Le principal inconvénient de cette API est sa dépendance à l'écosystème NVIDIA. Cette restriction limite fortement la portabilité des applications développées avec CUDA. De plus, la position de monopole de NVIDIA leur permet de maintenir des prix élevés pour leurs GPU, rendant ces technologies moins accessibles, ce qui peut représenter un obstacle significatif dans des domaines comme la recherche, souvent confrontée à des contraintes budgétaires.

D'un autre côté, CUDA présente de nombreux avantages, notamment ses hautes performances sur les GPU NVIDIA grâce à son optimisation spécifique. Son environnement de développement complet, incluant des bibliothèques optimisées (cuBLAS, cuFFT) et des outils comme Nsight, facilite le travail des développeurs et réduit le temps nécessaire à l'écriture et au débogage de code parallèle. De plus, la documentation détaillée et le large support de la communauté renforcent l'attractivité de cette API, en particulier pour les projets

exigeants tels que le deep learning, les simulations physiques et les calculs scientifiques.

OpenCL OpenCL, de son côté, est une API de calcul parallèle ouverte développée par le groupe Khronos. Elle est conçue pour fonctionner sur différentes plateformes, pas seulement des GPUs mais également des CPU, FPGA (Field Programmable Gate Arrays) ou d'autres accélérateurs. Cette flexibilité permet de créer des applications portables et performantes [5].

Bien que sa flexibilité soit un atout majeur, OpenCL ne bénéficie pas d'une optimisation spécifique pour chaque matériel supporté, ce qui peut entraîner des performances inférieures à celles de solutions propriétaires comme CUDA. De plus, le support de certains matériels, bien qu'annoncé comme compatible, peut s'avérer limité en termes de performances ou même incomplet, ce qui constitue un frein à son adoption dans certains contextes exigeants.

OpenCL peut être moins optimisé pour certains matériels, mais il reste performant sur des composants courants comme les GPU et CPU, avec des performances parfois comparables à CUDA sur les GPU NVIDIA. Son caractère multi-plateformes permet de l'utiliser sur divers dispositifs, ce qui le rend adapté aux environnements hétérogènes où l'interopérabilité est importante. Bien qu'il ne soit pas toujours aussi finement optimisé que CUDA pour des architectures spécifiques, il offre un bon compromis entre compatibilité et performances [6].

3 Contexte

Avec l'essor du calcul haute performance, l'optimisation de l'exécution des algorithmes est devenue un enjeu crucial, notamment dans des domaines comme l'intelligence artificielle, la modélisation scientifique et le traitement d'images. Les GPU (Graphics Processing Units), initialement conçus pour le rendu graphique, sont aujourd'hui largement utilisés pour le calcul parallèle intensif, surpassant souvent les CPU (Central Processing Units) dans certaines tâches spécifiques.

Deux frameworks majeurs permettent d'exploiter la puissance des GPU :

- CUDA (Compute Unified Device Architecture), développé par NVIDIA, qui offre une optimisation fine pour les cartes graphiques de la marque.
- OpenCL (Open Computing Language), une alternative plus généraliste et multiplateforme, permettant d'exécuter du code sur différents types de processeurs (CPU, GPU, FPGA, etc.).

Cependant, bien que les GPU offrent un parallélisme massif, ils ne sont pas toujours l'option la plus performante pour toutes les tâches. Le choix entre un calcul CPU et un calcul GPU dépend de plusieurs facteurs, notamment :

- La nature de l'opération (intensité de calcul, dépendance mémoire).
- La taille des données (bénéfices du parallélisme vs coût des transferts de mémoire).

– La complexité de l'implémentation (CUDA/OpenCL demandent un effort de développement supplémentaire par rapport à une simple exécution sur CPU).
– La disponibilité des ressources GPU (impact du partage entre plusieurs processus).

Dans cette étude, nous comparons les performances des implémentations CPU, CUDA et OpenCL sur plusieurs types de calculs :

– Calculs légers : addition de matrices, transposition, trace.
– Calculs lourds : multiplication matricielle, calcul du déterminant.

En plus de mesurer l'accélération obtenue par le calcul GPU, nous avons également exploré les limites du partage des ressources GPU. En effet, un GPU est souvent sollicité simultanément par plusieurs processus, par exemple pour l'affichage graphique ou d'autres tâches de calcul. Nous avons cherché à comprendre comment cette concurrence influence les performances des algorithmes, notamment en observant l'impact d'une charge GPU partielle sur l'exécution des benchmarks.

L'objectif est donc d'identifier dans quels cas il est réellement avantageux d'utiliser un GPU par rapport à un CPU, tout en prenant en compte les contraintes d'implémentation et les limites matérielles liées au partage des ressources.

4 Méthodologie

L'objectif de cette section est de détailler avec précision le matériel, les techniques et les protocoles que nous avons employés pour comparer de manière rigoureuse les performances du CPU et du GPU dans l'exécution d'opérations matricielles. Nous avons mis en œuvre CUDA et OpenCL pour l'accélération GPU et avons appliqué des optimisations sur CPU afin d'assurer une comparaison équitable et pertinente.

4.1 Matériel

Les expériences ont été réalisées sur une machine dotée d'un processeur Intel Core i7-8750H, un processeur hexacœur cadencé à 2,2 GHz avec Turbo Boost jusqu'à 4,1 GHz, et d'une carte graphique NVIDIA GTX 1050 Ti, dotée de 4 Go de mémoire GDDR5 et supportant CUDA ainsi qu'OpenCL. Le système dispose de 8 Go de mémoire vive DDR4, garantissant une exécution fluide des tests sur des matrices de grande taille. L'environnement d'exécution repose sur Ubuntu 22.04, un choix motivé par ses performances et sa compatibilité avec les outils d'accélération GPU. Le pilote installé est la version 560.35.05 des pilotes NVIDIA, compatible avec CUDA 11 et OpenCL 3.0, et configuré avec précision afin d'assurer une exploitation optimale du GPU.

Le choix d'un environnement Linux permet un contrôle accru sur l'utilisation des ressources système, avec une optimisation avancée des bibliothèques et des compilateurs pour garantir des mesures précises et répétables.

4.2 Génération des données

Afin de garantir la fiabilité et la cohérence de nos expériences, nous avons développé un système de génération automatique des matrices de test. Ces matrices, de tailles variables allant de 1x1 jusqu'à 8192x8192, sont générées aléatoirement avec des valeurs comprises entre -100 et 100. Elles sont soit de type entier (int), soit de type flottant (float), et les opérations sont effectuées uniquement entre matrices de même type. Une fois générées après la compilation du code, elles sont stockées en mémoire et réutilisées de manière identique pour toutes les implémentations, assurant ainsi une comparaison équitable des performances.

4.3 Opérations et implémentations

Nous avons sélectionné plusieurs opérations matricielles couramment utilisées en calcul scientifique et en intelligence artificielle : l'addition et la multiplication de matrices, le calcul du déterminant, le calcul de la trace et la transposition de matrices. Ces opérations ont été mises en œuvre avec différentes méthodes d'exécution afin de comparer les performances des architectures CPU et GPU.

Sur CPU, les opérations ont été implémentées en C++ natif. Nous avons également essayé d'implémenter une version optimisée en c visant à éviter tout chargement dynamique pour bénéficier des optimisations du compilateur ; nous avons compilée en utilisant les flags -O2 et -O3 pour comparer les deux optimisations. Sur GPU, l'exécution s'est faite via CUDA dans des fichiers ".cu" et via OpenCL avec des kernels en C et C++. Chaque implémentation est conçue pour exploiter au mieux les capacités de calcul parallèle des GPU et les optimisations spécifiques aux CPU modernes.

4.4 Mesure des performances

Pour obtenir des résultats fiables et reproductibles, nous avons appliqué une méthodologie de chronométrage rigoureuse. Sur CPU, le chronométrage débute juste avant l'exécution de l'opération et s'arrête immédiatement après l'obtention du résultat. Sur GPU, le chronométrage commence à la première allocation de ressource sur la carte graphique et s'arrête une fois que le résultat a été transféré vers la RAM principale.

Tous les temps d'exécution sont enregistrés sous forme de logs afin de permettre une analyse approfondie des performances et d'assurer la reproductibilité des expériences.

4.5 Automatisation et gestion des expériences

Afin d'assurer la reproductibilité et de simplifier l'exécution des expériences, nous avons développé un dépôt GitHub[1] contenant l'ensemble des scripts et du code source permettant l'automatisation complète du processus. Ce dépôt permet de :

[1] Le code source et les scripts d'automatisation sont disponibles sur GitHub : https://github.com/JulienS0t0/PER

- Compiler et exécuter automatiquement les différentes implémentations (CPU, CUDA et OpenCL) sans intervention manuelle.
- Gérer la génération des matrices d'entrée et leur stockage pour garantir une cohérence entre les différentes exécutions.
- Exécuter les benchmarks tout en collectant des traces de performance via `NVIDIA Nsight Systems`, en générant automatiquement des fichiers `.nsys-rep` pour chaque opération étudiée.
- Enregistrer systématiquement les temps d'exécution et autres métriques de performance dans un format structuré (fichiers logs ou CSV) pour faciliter l'analyse et la visualisation des résultats.

L'intégration de l'outil `nsys` a offert une visibilité détaillée sur les interactions entre le CPU et le GPU, notamment en termes de transferts mémoire et d'utilisation des unités de calcul.

Le dépôt GitHub est structuré de manière modulaire afin de faciliter son utilisation et son extension pour de futurs travaux. Une documentation détaillée accompagne les scripts afin de guider les utilisateurs souhaitant reproduire ou adapter les expériences réalisées dans cette étude.

4.6 Affichage et analyse des résultats

Les résultats sont présentés sous forme de courbes illustrant le temps d'exécution en fonction de la taille des matrices pour chaque type d'opération. Chaque courbe correspond à une implémentation particulière, permettant ainsi une comparaison visuelle des performances des différentes approches (CPU vs GPU, CUDA vs OpenCL, optimisations de compilation, etc.).

Ces graphiques permettent d'évaluer les gains de performance obtenus grâce à l'utilisation du GPU et d'identifier les scénarios où chaque méthode est la plus efficace.

5 Résultats

Nous observons que pour des matrices de petite taille ($N < 256$), le temps d'exécution sur GPU est significativement plus long que sur CPU. Cela s'explique par le fait que le temps total d'exécution inclut la transmission des données vers le GPU et leur récupération une fois les calculs terminés. Pendant ce temps, le CPU a déjà achevé son traitement.

Concernant l'addition matricielle, nous remarquons que le CPU est toujours plus rapide. En effet, cette opération nécessite principalement la lecture de deux valeurs en mémoire suivie de l'addition. L'envoi des données vers la carte graphique implique également la lecture de ces deux valeurs, ce qui signifie que la RAM constitue une limitation de performance pour les deux architectures.

En revanche, pour les opérations plus complexes comme la multiplication matricielle ou le calcul du déterminant, qui requièrent un nombre important de calculs pour chaque valeur résultante, les performances du CPU se dégradent

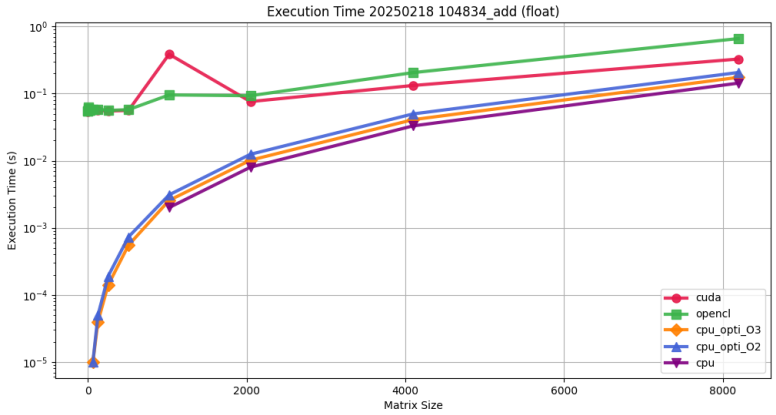

Fig. 6. Temps d'exécution de l'addition matricielle de nombres flottants en fonction de la taille de la matrice.

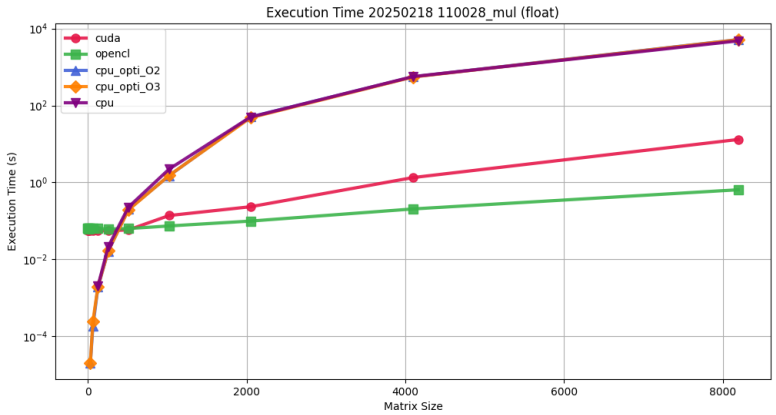

Fig. 7. Temps d'exécution de la multiplication matricielle de nombres flottants en fonction de la taille de la matrice.

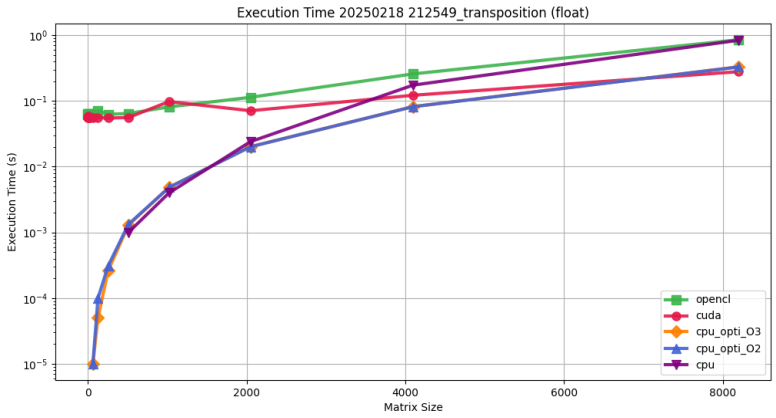

Fig. 8. Temps d'exécution de la transposition matricielle en fonction de la taille de la matrice.

rapidement au-delà de $N = 256$, atteignant plusieurs heures d'exécution pour $N = 8192$. À l'inverse, sur GPU, bien que les temps de calcul soient plus élevés pour de petites matrices, ils restent inférieurs à une seconde pour $N = 8192$.

Cette différence s'explique par le parallélisme massif du GPU : bien que les données doivent être transférées en mémoire deux fois avant le traitement, le calcul est ensuite effectué en parallèle, contrairement au CPU qui exécute les opérations de manière séquentielle. Ainsi, bien que le coût de transfert de données soit similaire dans les deux cas, l'avantage du GPU devient évident pour les calculs intensifs.

Par ailleurs, les opérations effectuées avec des nombres flottants prennent plus de place en mémoire et nécessitent un temps d'exécution légèrement plus long que les entiers. Par exemple, sous OpenCL, l'addition de matrices de taille 8192×8192 prend environ 638 ms en entier contre 659,35 ms en flottant, tandis que la multiplication prend 624 ms en entier contre 641 ms en flottant. Cette différence s'explique par la nature des opérations en virgule flottante, qui impliquent une gestion plus complexe des arrondis et des précisions. Nous avons constaté que CUDA introduit des arrondis dans certaines valeurs de calculs en virgule flottante, probablement dus à des optimisations internes ou à des conversions de type automatique. Ce point est crucial pour les applications nécessitant une précision maximale, telles que certaines simulations scientifiques ou calculs financiers où il faudra alors privilégier OpenCL.

Une question essentielle concerne la gestion de la mémoire vidéo (VRAM). Lorsque la mémoire disponible sur le GPU est insuffisante pour stocker les données nécessaires aux calculs, l'allocation mémoire est refusée par le driver, rendant l'exécution impossible. Une alternative serait d'utiliser un mécanisme de pagination mémoire, mais cela introduirait des latences importantes.

Concernant la consommation énergétique, bien que nous n'ayons pas pu effectuer de mesures directes, des études montrent généralement que, pour un même volume de calcul, le GPU consomme moins d'énergie que le CPU. Cette différence est due à l'architecture parallèle du GPU, qui permet d'effectuer un grand nombre d'opérations en une seule instruction, contrairement au CPU qui fonctionne de manière plus séquentielle.

6 Discussion

Bien que nous ayons eu accès au matériel nécessaire pour mener à bien nos expériences, la nature même de cette étude requiert impérativement un GPU performant. Or, cette exigence technique constitue une contrainte non négligeable, car tous les équipements ne disposent pas des ressources nécessaires pour exécuter efficacement les calculs intensifs que nous avons entrepris. La puissance de traitement graphique étant un élément clé dans ce type d'analyse, il nous a fallu nous adapter aux capacités du matériel à notre disposition, ce qui a parfois influencé la manière dont nous avons conçu et interprété nos expériences.

Au cours de ce travail, nous avons été confrontés à plusieurs difficultés, tant sur le plan méthodologique que technique. L'un des principaux défis a résidé dans la recherche documentaire. En effet, bien que la littérature scientifique regorge d'articles détaillant les performances des GPU dans des applications bien spécifiques, nous avons constaté qu'il existait peu d'études directement en lien avec notre sujet d'intérêt. Cette absence de références précises a complexifié l'analyse de nos résultats et nous a contraints à établir nos propres critères d'évaluation en nous basant sur des extrapolations issues de domaines connexes. Cette étape a nécessité un important travail d'interprétation et de mise en perspective afin d'assurer la pertinence et la rigueur scientifique de notre démarche.

Par ailleurs, bien que certaines notions abordées dans cette étude aient été introduites dans notre cursus pédagogique, leur traitement est resté relativement superficiel, ce qui nous a obligés à approfondir plusieurs concepts de manière autonome. La maîtrise des architectures matérielles et logicielles a donc constitué un enjeu fondamental, nous amenant à explorer de nouvelles ressources et à développer des compétences pratiques qui n'étaient pas initialement à notre portée.

Un autre obstacle majeur a été l'installation et la configuration de l'environnement de travail sous Linux, notamment en ce qui concerne la gestion des pilotes GPU. Cette étape, qui s'est révélée plus ardue que prévu, a exigé des ajustements répétés et une compréhension fine des interactions entre le système d'exploitation et le matériel. Les incompatibilités rencontrées, les erreurs de configuration et les conflits entre différentes versions de bibliothèques ont considérablement ralenti notre progression initiale et ont nécessité de nombreuses heures de recherche et d'expérimentation avant d'aboutir à un environnement stable et fonctionnel.

Enfin, la mise en place de la méthodologie expérimentale a soulevé plusieurs interrogations, en particulier concernant la pertinence des benchmarks effectués et les paramètres influençant les performances mesurées. Il nous a fallu mener

une réflexion approfondie sur les conditions de test, les métriques à privilégier et les facteurs pouvant biaiser nos résultats. Assurer la fiabilité et la cohérence de nos expériences a représenté un défi important, qui nous a contraints à ajuster notre approche au fil des observations et à développer une rigueur analytique accrue.

Malgré ces défis, cette étude nous a permis d'acquérir une compréhension plus approfondie des composants d'un ordinateur, de leur fonctionnement et de leurs limites en fonction des usages. L'analyse des performances nous a apporté des enseignements précieux sur les capacités et les faiblesses des différentes configurations testées, nous sensibilisant ainsi aux choix matériels adaptés aux besoins spécifiques des utilisateurs. Plus largement, cette expérience nous a permis de renforcer nos compétences en recherche documentaire, en résolution de problèmes techniques et en conception expérimentale, des acquis qui nous seront indéniablement utiles dans nos futurs travaux.

7 Conclusion

L'étude des calculs GP-GPU sur cartes graphiques nous a permis d'approfondir notre compréhension des architectures parallèles et des techniques d'optimisation associées. Nous avons analysé les avantages et les limites du calcul massivement parallèle en exploitant les capacités des GPU modernes, mettant en évidence les gains de performance obtenus par rapport aux approches CPU traditionnelles.

Nos expérimentations ont montré que l'utilisation efficace des ressources GPU repose sur une bonne compréhension de la gestion de la mémoire, du partitionnement des tâches et de la réduction des latences. Nous avons ainsi pu identifier les bonnes pratiques pour maximiser les performances et minimiser les limitations de performance.

Toutefois, notre approche présente certaines limites :

- Nos benchmarks sont réalisés sur une seule configuration matérielle, ce qui restreint la portée des conclusions.
- Nous n'avons pas exploité le multithreading sur CPU, ce qui pourrait influencer nos conclusions quant à l'écart de performances observé.
- Notre implémentation pourrait être améliorée pour optimiser le chargement des matrices lors de banchmarks consécutifs. Actuellement, les matrices utilisées sont déchargées une fois le programme terminé, hors cette matrice va être utilisé pour tester la prochaine implémentation.

Pour aller plus loin, plusieurs axes d'amélioration sont envisageables :

- Tester différentes architectures GPU (AMD, Intel) pour évaluer la portabilité et les performances d'OpenCL.
- Comparer les implémentation CPU d'OpenCL et utiliser des compilateurs optimisés comme ceux d'Intel.
- Analyser plus en détail l'impact des transferts mémoire et explorer des stratégies d'optimisation avancées.

– Mesurer effectivement les différences de consommation pour le même travail entre les différentes implémentations sur CPU et GPU.

En conclusion, cette étude souligne l'importance croissante du calcul GP-GPU dans les calculs matriciel bien que cela soit applicable à de nombreux autres domaines. Les avancées constantes dans ce domaine ouvrent de nouvelles perspectives et encouragent l'exploration de méthodes toujours plus efficaces pour tirer parti des architectures parallèles. L'évolution des frameworks de programmation et des architectures matérielles laisse entrevoir des opportunités encore plus vastes pour l'accélération des calculs intensifs. Cette étude constitue une base solide pour de futurs travaux explorant des optimisations plus avancées et l'intégration de ces technologies dans des applications industrielles et de recherche.

References

1. NVIDIA Corporation. About us. https://www.nvidia.com/en-us/about-nvidia. Consulté en novembre 2024.
2. NVIDIA Corporation. Geforce graphics cards. https://www.nvidia.com/en-us/geforce/graphics-cards/. Consulté en novembre 2024.
3. NVIDIA Corporation. Nvidia's next generation cudatm compute architecture: Kepler gk110/210. Technical report, NVIDIA Corporation, 2014.
4. NVIDIA Corporation. Nvidia ada gpu architecture. Technical report, NVIDIA Corporation, 2023.
5. Khronos Group. *Open Standard for Parallel Programming of Heterogeneous Systems*, 2024. Accessed: 2024-11.
6. Firas Hamze Kamran Karimi, Neil G. Dickson. A performance comparison of cuda and opencl. *arxiv*, 2010.
7. Yves Denneulin L.A. Torres, Carlos J. Barrios H. Evaluation of computational and power performance in matrix multiplication methods and libraries on cpu and gpu using mkl, cublas, and sycl. *arxiv*, 2024.
8. Shinpei Kato Karthik Lakshmanan and Ragunathan (Raj) Rajkumar† Yutaka Ishikawa. Timegraph: Gpu scheduling for real-time multi-tasking environments. *Usenix*, 2011.
9. Ming Li Ziqian Bi Tianyang Wang Yizhu Wen Qian Niu Junyu Liu Benji Peng Sen Zhang Xuanhe Pan Jiawei Xu Jinlang Wang Keyu Chen Caitlyn Heqi Yin Pohsun Feng Ming Liu. Deep learning and machine learning with gpgpu and cuda: Unlocking the power of parallel computing. *arxiv*, 2024.
10. Erik Lindholm John Nickolls Stuart Oberman John Montrym. Nvidia tesla: Aunified graphics and computing architecture. *Nvidia*, 2008.
11. NVIDIA. *CUDA Toolkit Documentation 12.6 Update 3*, 2024. Accessed: 2024-11.
12. Chris McClanahan Georgia Tech College of Computing. History and evolution of gpu architecture. *Georgia Tech College of Computing, Technical Report*, 2022.
13. TechPowerUp. Gpu specs database. https://www.techpowerup.com/gpu-specs/. Consulté en novembre 2024.
14. Marek Toma Masaryk University. Evolution of nvidia gpu from microarchitectures pascal to ampere. *Masaryk University, Technical Report*, 2022.
15. Wikipedia. List of nvidia graphics processing units. https://en.wikipedia.org/wiki/List$_o f_N vidia_g raphics_p rocessing_u nits. Consultennovembre$2024.