

Université d'Ottawa  
Faculté de génie

École de science informatique  
et de génie électrique



University of Ottawa  
Faculty of Engineering

School of Electrical Engineering  
and Computer Science

## **CEG 4536 Architecture des ordinateurs III Automne 2024**

### ***Lab3 : Optimisation de la hiérarchie de la mémoire CUDA***

Groupe #1

[Chergui, Hakim](#) - 300173357

[Krayem, Christopher](#) - 300212035

[Simon Marchildon](#) - 300242291

[Adam Taktek](#) - 300110268

[Maro Mohamed Sherine Zaher Abdine](#) - 300222317

[Bensalah, Reda](#) - 300217542

Dates de l'expérimentation: 11, 18 Novembre 2024

Date de la soumission: 23 Novembre 2024

## Table of Contents

<b>Table of Contents.....</b>	<b>2</b>
<b>Table of Figures.....</b>	<b>2</b>
<b>Table of Tables.....</b>	<b>2</b>
<b>Introduction.....</b>	<b>3</b>
<b>Objectif.....</b>	<b>3</b>
<b>Analyse du problème.....</b>	<b>3</b>
<b>Conception de la solution.....</b>	<b>5</b>
<b>Équipements &amp; Composants Utilisés.....</b>	<b>6</b>
<b>Procédure de résolution et algorithme.....</b>	<b>7</b>
<b>Évaluation d'implémentation.....</b>	<b>10</b>
<b>Résultats et Validation.....</b>	<b>11</b>
<b>Problèmes Rencontrés.....</b>	<b>14</b>
<b>Conclusion.....</b>	<b>14</b>
<b>Distribution de tâches.....</b>	<b>16</b>
<b>Références.....</b>	<b>16</b>
<b>Appendice.....</b>	<b>17</b>

## Table of Figures

<b>Figure 1.....</b>	<b>9</b>
<b>Figure 2.....</b>	<b>9</b>
<b>Figure 3.....</b>	<b>10</b>
<b>Figure 4.....</b>	<b>11</b>
<b>Figure 5.....</b>	<b>12</b>
<b>Code du Prof en ajoutant le temps d'exécution.....</b>	<b>14-15</b>
<b>Code en CUDA en ajoutant le temps d'exécution.....</b>	<b>16-17</b>

## Table of Tables

<b>Table 1.....</b>	<b>13</b>
---------------------	-----------

# Introduction

Dans le cadre de ce laboratoire, nous explorons le rôle crucial de la gestion efficace de la mémoire dans le domaine du calcul haute performance, en particulier avec les accélérateurs GPU. CUDA, une plateforme de calcul parallèle, propose un modèle de mémoire sophistiqué permettant d'optimiser la latence et la bande passante entre la mémoire de l'hôte et celle du périphérique. Ce projet met en lumière les différents niveaux de mémoire CUDA (tels que les registres, la mémoire partagée et la mémoire globale) et vise à optimiser les schémas d'accès à la mémoire pour garantir une programmation GPU efficace.

## Objectif

Ce projet a pour but de développer une application utilisant CUDA pour tirer pleinement parti de la hiérarchie de mémoire et ainsi maximiser la performance. L'objectif est d'exploiter des techniques avancées pour réduire la latence et optimiser l'utilisation de la bande passante. Par exemple, on travaillera sur la coalescence de la mémoire pour regrouper les accès à la mémoire globale de manière efficace, la gestion des conflits entre les banques de mémoire partagée pour éviter les ralentissements, et la réduction des transferts de données entre l'hôte (CPU) et le GPU afin de minimiser les coûts de communication. À travers ces approches, on pourra concevoir et mettre en œuvre des solutions capables de rendre les applications CUDA plus rapides et performantes, tout en optimisant l'utilisation des ressources matérielles du GPU.

## Analyse du problème

### Tâche 1: Définir le problème

Dans cette partie, il nous faut solutionner le problème initial. Nous devons réaliser la multiplication matricielle. Faire ce calcul de manière séquentielle est lent et exhaustif. Il consiste en la multiplication de la première rangée d'une matrice avec la colonne de l'autre matrice, et ainsi de suite pour toutes les paires de rangé et colonne. On réalise que chaque multiplication de rangé et colonne pourrait être réalisée en même temps. Ceci démontre l'intérêt d'utiliser le parallélisme CUDA, sur lequel toutes ces opérations parallèles pourraient être effectuées. Pour garder la solution simple, la mémoire globale sera utilisée dans cette partie.

### Tâche 2: Optimisation de l'accès à la mémoire globale :

Dans le processus de multiplication matricielle traditionnel sur les GPU, les méthodes d'accès à la mémoire globale peuvent provoquer une inefficacité notable, particulièrement du fait des accès non coalescés. Ils surviennent lorsque les threads d'un bloc se rendent dans des colonnes distinctes de la matrice B, contraignant ainsi la mémoire à effectuer des transactions

additionnelles non optimisées. Ceci accroît le temps de latence des lectures en mémoire globale, une ressource qui se distingue d'une latence importante en comparaison avec la mémoire commune ou les registres. Par conséquent, chaque élément de la matrice résultante  $C$  est lu à plusieurs reprises sans interruption, ce qui diminue le rendement de la bande passante et prolonge le temps d'exécution total. Cette question se pose de manière particulièrement importante lorsque les dimensionnements des matrices augmentent.

### Tâche 3: Utilisation de la mémoire partagée :

Pour cette tâche, l'objectif est de multiplier deux matrices en utilisant efficacement la mémoire partagée pour minimiser les accès à la mémoire globale, qui est plus lente. La mémoire partagée, accessible à tous les threads d'un bloc, offre une latence bien inférieure et permet de réduire considérablement les transactions en mémoire globale si elle est utilisée de manière stratégique. L'approche consiste à diviser les matrices  $A$  et  $B$  en sous-matrices appelées tuiles, qui seront traitées bloc par bloc. Chaque bloc de threads copie les données des tuiles de  $A$  et  $B$  dans la mémoire partagée, effectue les calculs nécessaires localement pour les éléments de la tuile correspondante dans  $C$ , puis répète le processus jusqu'à ce que toutes les tuiles pertinentes soient multipliées.

### Tâche 4: Optimiser les schémas d'accès à la mémoire partagée :

Le problème consiste à optimiser la multiplication matricielle sur GPU en évitant les conflits de banques dans la mémoire partagée, ce qui ralentit les performances. Ces conflits surviennent lorsque plusieurs threads accèdent simultanément à la même banque mémoire. Une gestion inefficace des accès mémoire entraîne également des goulots d'étranglement, limitant l'efficacité du calcul parallèle. L'objectif est d'organiser et d'aligner les données en mémoire pour minimiser ces conflits tout en garantissant une synchronisation correcte des threads.

### Tâche 5: Profilage des performances :

Le problème abordé est l'optimisation de la multiplication matricielle, une opération fondamentale utilisée dans des domaines tels que l'apprentissage automatique et la simulation scientifique. Elle est gourmande en calculs et en mémoire, particulièrement pour de grandes matrices. La lenteur des implémentations séquentielles sur CPU, combinée à l'inefficacité potentielle des accès mémoire, justifie l'exploration de solutions parallèles. L'objectif est donc de maximiser l'utilisation des ressources GPU, réduire les temps d'exécution et garantir des résultats précis.

### Tâche 6: Optimisation itérative

Le problème consiste à effectuer une multiplication matricielle  $C=A \times B$  en utilisant les capacités de parallélisme offertes par les GPU via CUDA. La multiplication matricielle est une opération

essentielle dans de nombreux domaines tels que l'apprentissage automatique, la simulation numérique et le traitement d'images. Cependant, lorsqu'elle est implémentée naïvement sur un CPU, elle peut être lente pour de grandes matrices en raison de sa complexité algorithmique  $O(n^3)$ . L'objectif est donc de concevoir une solution optimisée qui exploite les threads GPU pour distribuer les calculs entre des milliers de cœurs, tout en minimisant les latences dues aux accès mémoire non optimaux.

## Conception de la solution

### Tâche 1: Définir le problème

Pour commencer, il nous faut créer une fonction main qui fait appel à notre kernel d'intérêt produisant la multiplication matricielle. Dans le main, on crée des matrices, on les initialise, puis on appelle notre fonction qui va réaliser notre multiplication matricielle. On mesure aussi le temps d'exécution. Finalement, on affiche la matrice résultante et on compare avec les résultats du CPU.

Dans la fonction de multiplication matricielle, on alloue des matrices dans le device en copiant les valeurs du host. On définit la dimension des blocs et de la grille. On appelle le kernel qui va réaliser la multiplication matricielle dans CUDA.

Dans le kernel, on calcule les indices de rangée et colonne du thread présent. Si ceux-ci sont compris dans les limites de la matrice, alors on multiplie la rangée de A avec la colonne de B.

### Tâche 2: Optimisation de l'accès à la mémoire globale :

On conserve la même structure que dans la tâche 1, seulement on modifie le contenu du kernel.

Afin de dépasser ces contraintes, on suggère de transposer la matrice B en mémoire hôte avant de procéder aux calculs sur le processeur numérique. Cette transposition transforme les colonnes de B, habituellement lues en lignes non contiguës, en lignes contiguës dans la matrice transposée BT. Cela donne aux threads la possibilité d'accéder ensemble aux éléments de BT lors des calculs, alignant les lectures en fonction des besoins spécifiques des threads d'un bloc. Par la suite, le kernel est ajusté en utilisant BT au lieu de B, ce qui permet d'exploiter au mieux les lectures parallèles et de diminuer les transactions superflues dans la mémoire totale. Cette méthode optimise la capacité de mémoire et diminue le délai d'accès, tout en préservant la précision des calculs.

### Tâche 3: Utilisation de la mémoire partagée :

Pour implémenter cette solution, on commence par allouer des tableaux en mémoire partagée pour stocker temporairement les tuiles de A et B. Chaque thread d'un bloc charge un élément de la tuile de A et de B correspondant à ses indices locaux (donnés par `threadIdx.x` et `threadIdx.y`) depuis la mémoire globale. Une fois que toutes les données des tuiles sont dans la mémoire

partagée, les threads synchronisent leurs actions pour s'assurer que toutes les données nécessaires sont disponibles avant de commencer les calculs. Chaque thread calcule ensuite une partie du produit scalaire entre une ligne de  $A$  et une colonne de  $B$  en itérant sur les éléments des tuiles. La somme intermédiaire est stockée dans un registre local pour minimiser les lectures/écritures en mémoire. Après avoir traité toutes les tuiles, le résultat final du produit scalaire est écrit dans la mémoire globale à l'indice correspondant dans  $C$ . Cette approche permet de maximiser le parallélisme tout en minimisant la latence liée aux accès à la mémoire, ce qui rend la multiplication matricielle beaucoup plus efficace pour des matrices de grande taille comme celles utilisées ici.

#### Tâche 4: Optimiser les schémas d'accès à la mémoire partagée :

La solution proposée repose sur l'utilisation de mémoires partagées avec alignement optimisé pour éviter les conflits de banques. Chaque bloc de threads GPU travaille sur une sous-matrice de taille  $TILE\_SIZE \times TILE\_SIZE$ , qui est copiée dans des matrices partagées  $tileA$  et  $tileB$ . Pour aligner correctement les accès, un padding a été ajouté à ces matrices partagées, augmentant leur taille à  $(TILE\_SIZE+1) \times TILE\_SIZE$ . Ce décalage garantit que chaque thread accède à une banque différente, évitant ainsi les conflits de banques. En plus de cette optimisation, la conception du kernel inclut une synchronisation explicite des threads à l'aide de `__syncthreads()` pour garantir que tous les éléments nécessaires d'une sous-matrice sont chargés avant le début des calculs. Le kernel est structuré de manière à parcourir les matrices  $A$  et  $B$  en sous-blocs, multipliant les parties correspondantes et accumulant les résultats dans des variables locales pour réduire les accès fréquents à la mémoire globale. Cette conception tire parti des capacités de parallélisme massif du GPU tout en maximisant l'utilisation de la mémoire partagée pour améliorer les performances globales.

#### Tâche 5: Profilage des performances :

La solution repose sur l'utilisation de CUDA pour paralléliser la multiplication matricielle en décomposant les matrices en blocs manipulés indépendamment par les threads GPU. Pour améliorer les performances, la mémoire partagée est utilisée afin de limiter les accès à la mémoire globale, et un padding est ajouté pour éliminer les conflits bancaires. Les blocs de threads sont configurés pour équilibrer la charge de travail, et des techniques comme la synchronisation garantissent la cohérence des calculs.

#### Tâche 6: Optimisation itérative

Pour résoudre cette tâche, nous avons développé et optimisé un algorithme de multiplication matricielle en utilisant CUDA. L'objectif principal était d'exploiter la parallélisation GPU pour améliorer les performances par rapport à une implémentation séquentielle sur CPU. L'algorithme repose sur la décomposition des matrices en sous-blocs (tiling) afin de maximiser l'utilisation de la mémoire partagée, tout en minimisant les accès coûteux à la mémoire globale. Chaque thread

calcule une partie de la matrice résultante en chargeant des blocs de la matrice source dans des matrices partagées, avec un rembourrage (padding) pour éviter les conflits bancaires.

## Équipements & Composants Utilisés

- Windows PC  
Le système d'exploitation Windows est essentiel pour ce laboratoire car il est compatible avec les outils utilisés pour le développement, comme Visual Studio 2022 et le CUDA Toolkit. De plus, nos machines de travail qui contiennent les cartes GPU Nvidia fonctionnent sous Windows, ce qui permet de compiler et exécuter le programme CUDA de manière optimale.
- Visual Studio 2022  
Cet IDE (environnement de développement intégré) est utilisé pour écrire, déboguer et compiler le code C++ et CUDA. Visual Studio offre une intégration fluide avec CUDA Toolkit et debugger, permettant de compiler des programmes CUDA directement et de gérer des projets complexes grâce à son interface utilisateur riche et ses fonctionnalités avancées de gestion de code.
- Carte GPU Nvidia  
La carte GPU est l'élément clé qui permet d'exécuter les calculs massivement parallèles en utilisant CUDA. Nvidia est le principal fabricant de GPU prenant en charge CUDA, ce qui nous permet l'exécution simultanée de milliers de threads, accélérant les calculs massivement parallèles comme les algorithmes de réduction.
- CUDA Toolkit  
Cet ensemble d'outils est indispensable pour développer et optimiser des applications CUDA. Il comprend un compilateur, des bibliothèques et des outils de développement pour nous permettre d'exploiter la puissance du GPU. Le CUDA Toolkit permet de paralléliser les calculs sur le GPU, ce qui est au cœur de l'optimisation demandée dans ce laboratoire.
- NVIDIA Profiler (nvprof)  
L'outil de profilage est utilisé pour analyser les performances des applications CUDA. Il permet d'identifier les goulets d'étranglement et d'optimiser l'occupation des warps et l'utilisation de la mémoire.
- CUDA Debugger  
Le débogueur CUDA est essentiel pour tester et profiler les kernels CUDA. Il permet de diagnostiquer les erreurs dans le code CUDA en exécutant les threads GPU pas à pas.

Cela aide à identifier les erreurs de synchronisation ou de gestion de mémoire, ce qui est crucial.

## Procédure de résolution et algorithme

### Tâche 1: Définir le problème

Dans main, on définit les tailles de la matrice carré, la taille pour stocker une matrice de cette taille avec des valeurs integer. On alloue les matrices d'entrée a et b et de sortie c dans la mémoire du host. On initialise leur valeur avec une boucle for. Puis, on crée les événement de “start” et de “stop” pour mesurer le temps d’exécution, on appelle la multiplication matricielle et affiche le temps d’exécution. Finalement, on affiche la matrice résultante de la multiplication. On vérifie que le résultat de la multiplication est juste avec une fonction CPU. Pour terminer, on libère la mémoire des matrices hôtes.

Dans la fonction de multiplication matricielle, on alloue des matrices A et B d'entrée, et une matrice de sortie C. Puis, on copie les matrices hôtes dans le device. On choisit les tailles de bloc et de grille. On appelle le kernel, synchronise, copie le résultat du device au host, et on libère la mémoire du device.

Dans le kernel, on calcul les indices de rangé et colonne du thread présent. Si ceux-ci sont compris dans les limites de la matrice, alors on multiplie la rangée de A avec la colonne de B. Pour ce faire, on parcourt ceux-ci avec une boucle for qui accumule dans tmp les résultats de multiplication de cellule de A et B. tmp est affecté à la cellule C à rangé “row” et colonne “col”.

### Tâche 2: Optimisation de l'accès à la mémoire globale :

La méthode choisie pour optimiser la multiplication matricielle sur GPU repose sur une amélioration des schémas d'accès mémoire. Initialement, les accès à la mémoire globale pour lire les colonnes de la matrice B étaient non coalescés, ce qui diminuait l'efficacité des calculs. Pour résoudre ce problème, la matrice B a été transposée avant d'être envoyée au GPU. Cette transposition permet d'aligner les lectures avec les threads du bloc de calcul, rendant les accès mémoire plus efficaces. Une fois la matrice transposée prête, un nouveau kernel a été conçu pour travailler avec cette nouvelle structure. Ce kernel calcule les valeurs de la matrice résultante C en utilisant les données transposées, tout en respectant les principes d'accès coalescés pour maximiser la bande passante mémoire. Enfin, les données sont copiées du GPU vers le CPU pour validation.

### Tâche 3: Utilisation de la mémoire partagée :

Dans cette implémentation, le kernel CUDA est configuré pour gérer la multiplication matricielle en divisant les matrices A et B en tuiles carrées, dont la taille correspond à celle des blocs de



threads. Ces tuiles sont stockées temporairement en mémoire partagée pour exploiter la faible latence de cette dernière. La taille du bloc est fixée, ce qui détermine combien d'éléments de  $A$  et  $B$  chaque bloc peut traiter simultanément.

Pour gérer efficacement les données, deux tableaux sont déclarés dans la mémoire partagée, représentant les tuiles locales de  $A$  et  $B$ . Chaque bloc de threads charge dans ces tableaux une sous-matrice respective depuis la mémoire globale. Ces tuiles locales permettent de réduire le nombre d'accès à la mémoire globale, qui est plus lente, lors des calculs.

Les indices globaux des threads, représentant les lignes et colonnes de la matrice résultat  $C$ , sont calculés. Ces indices permettent à chaque thread de savoir quelles parties des matrices  $A$  et  $B$  il doit traiter. Une variable locale est initialisée dans chaque thread pour accumuler le résultat partiel du produit scalaire entre une ligne de  $A$  et une colonne de  $B$ . Cette variable est stockée dans des registres pour un accès rapide.

Les threads chargent ensuite les tuiles dans la mémoire partagée. Les conditions vérifient que les indices d'accès restent dans les limites des matrices pour éviter des erreurs. Une synchronisation est effectuée après le chargement des données pour s'assurer que toutes les tuiles sont disponibles avant de procéder aux calculs. Chaque thread calcule ensuite sa contribution au produit scalaire en multipliant les éléments correspondants des tuiles actuelles de  $A$  et  $B$ , et en les accumulant dans la variable locale.

Après avoir effectué les calculs pour une tuile, une synchronisation supplémentaire s'assure que tous les threads du bloc ont terminé avant que les nouvelles tuiles ne soient chargées. Une fois toutes les tuiles traitées, chaque thread écrit la valeur finale calculée dans la mémoire globale à l'emplacement approprié de la matrice  $C$ . Cela garantit que le résultat final est correctement assemblé.

Cette approche exploite efficacement la mémoire partagée pour minimiser les coûts des accès à la mémoire globale. En réalisant les calculs sur les tuiles chargées localement, les transactions coûteuses en mémoire globale sont significativement réduites, ce qui améliore les performances, particulièrement pour les matrices de grande taille.

#### Tâche 4: Optimiser les schémas d'accès à la mémoire partagée :

Pour résoudre la tâche d'optimisation des accès à la mémoire partagée et éviter les conflits de banques, une approche basée sur l'utilisation de mémoire partagée avec un padding a été adoptée. Dans CUDA, lorsque plusieurs threads accèdent à la même banque au même cycle d'horloge, un conflit de banque se produit, ce qui ralentit l'exécution. Pour y remédier, un décalage a été introduit en ajoutant une colonne supplémentaire aux matrices partagées (avec `PADDED_TILE_SIZE`), ce qui force les accès à être alignés sur des banques différentes. L'algorithme a été modifié pour charger les blocs de la matrice dans cette mémoire partagée

optimisée. Chaque thread charge un élément de la matrice globale dans une position spécifique de la mémoire partagée, après avoir vérifié que ses indices se trouvent dans les limites de la matrice pour éviter des accès hors limites. Une fois les blocs chargés, tous les threads synchronisent leur exécution à l'aide de `__syncthreads()` pour garantir que toutes les données partagées soient prêtes avant de commencer les calculs. Le calcul s'effectue ensuite sur les blocs de matrices et les résultats sont accumulés dans des variables locales pour finalement être stockés dans la mémoire globale. Cette méthode améliore l'efficacité des accès mémoire tout en exploitant au maximum le parallélisme offert par le GPU.

#### Tâche 5:Profilage des performances :

La procédure de résolution repose sur la parallélisation de la multiplication matricielle en utilisant les ressources GPU de manière optimale. Les matrices A et B sont divisées en blocs de taille fixe définie par `TILE_SIZE`, chaque bloc étant traité par une grille de threads GPU, où chaque thread calcule un élément de la matrice résultat C. Pour améliorer l'efficacité, les blocs des matrices sont copiés dans la mémoire partagée du GPU, ce qui réduit les accès coûteux à la mémoire globale, et un padding est utilisé pour éviter les conflits bancaires. Une boucle principale itère sur les sous-matrices, charge les blocs dans des variables partagées locales `TileA` et `TileB`, et exécute les calculs après une synchronisation explicite avec `__syncthreads()`. Les threads accumulent les produits dans une variable locale `sum`, qui est ensuite écrite dans la mémoire globale pour former C. Les dimensions des blocs et de la grille sont configurées pour couvrir la matrice tout en répartissant la charge. Les résultats GPU sont vérifiés avec une version CPU séquentielle pour valider l'implémentation. Cette approche intègre parallélisme, gestion efficace de la mémoire et synchronisation pour résoudre efficacement le problème.

#### Tâche 6:Optimisation itérative

Pour optimiser la multiplication de matrices sur GPU, nous avons adopté une approche basée sur l'utilisation de la mémoire partagée et l'optimisation du parallélisme à travers le déroulement de boucles. Dans un premier temps, le problème a été divisé en blocs et threads afin d'exploiter efficacement l'architecture CUDA. Chaque bloc traite une portion de la matrice de sortie, et chaque thread au sein d'un bloc calcule un élément individuel de cette portion. Les sous-matrices (ou "tuiles") de taille fixe sont chargées dans la mémoire partagée afin de réduire les accès coûteux à la mémoire globale. Une technique de "padding" a été appliquée sur les tuiles pour éviter les conflits de banques dans la mémoire partagée. Pour améliorer davantage les performances, nous avons implémenté un déroulement de boucles (loop unrolling) en itérations multiples, ce qui réduit les surcharges liées au contrôle des boucles et augmente le parallélisme intra-thread. L'algorithme a été structuré pour charger des blocs de données en mémoire partagée, synchroniser les threads pour garantir une exécution correcte, et effectuer les calculs nécessaires avant de copier les résultats dans la mémoire globale. Cette approche combinant mémoire partagée, synchronisation, et déroulement de boucles a permis de minimiser les goulots

d'étranglement liés à la mémoire et de maximiser l'efficacité des ressources du GPU, tout en garantissant une exactitude des résultats vérifiée par une comparaison avec une implémentation séquentielle.

## Évaluation d'implémentation

### Tâche 1: Définir le problème

En exécutant le programme de cette tâche, on constate que nous avons atteint les objectifs définis. La multiplication matricielle par le kernel est juste et efficace. La fonction séquentielle du CPU nous permet de vérifier l'exactitude du kernel CUDA avec les méthodes `assert()` qui font exception si les valeurs calculées par le kernel sont différentes du CPU. Dans cette partie, nous avons réalisé une grosse partie du travail. Non seulement, nous avons trouvé une première solution efficace à la multiplication matricielle grâce à CUDA, mais nous avons aussi créé les cas de test.

### Tâche 2: Optimisation de l'accès à la mémoire globale :

Pour évaluer les performances de l'optimisation, des métriques comme le temps d'exécution du kernel et l'utilisation de la bande passante mémoire ont été analysées. Les résultats montrent une amélioration significative : le temps d'exécution du kernel a diminué de manière notable. Cela reflète une réduction de la latence causée par les accès non optimaux à la mémoire globale. Les profils d'exécution générés avec *nvprof* confirment également que les coûts associés aux transactions mémoire entre l'hôte et le périphérique n'ont pas augmenté, ce qui indique que l'optimisation est focalisée sur l'efficacité des calculs GPU sans ajouter de surcharge inutile.

### Tâche 3: Utilisation de la mémoire partagée :

Avec cette implémentation, nous avons réussi à atteindre les objectifs fixés en exploitant efficacement la mémoire partagée pour optimiser les performances du calcul de multiplication matricielle. En divisant les matrices en tuiles et en stockant temporairement ces sous-matrices dans la mémoire partagée, nous avons significativement réduit le nombre d'accès à la mémoire globale, limitant ainsi l'impact de sa latence élevée. Cette approche garantit que chaque thread traite les données de manière locale et rapide, tout en minimisant les conflits et les synchronisations inutiles. Les résultats obtenus montrent une amélioration notable des performances, particulièrement pour des matrices de grande taille, démontrant la pertinence de l'utilisation de la mémoire partagée dans ce contexte pour accélérer les calculs parallèles.

### Tâche 4: Optimiser les schémas d'accès à la mémoire partagée :

L'implémentation a été testée sur une matrice de taille 512x512 en utilisant CUDA avec des blocs de taille 16x16, correspondant à une configuration optimale pour ce type de matériel. Les performances du kernel ont été mesurées en utilisant des événements CUDA pour capturer le temps d'exécution. Les optimisations apportées ont permis de minimiser les conflits de banques et de maximiser l'utilisation de la bande passante mémoire. La comparaison avec une version non optimisée a montré une amélioration notable en termes de temps d'exécution, démontrant que l'utilisation d'un padding et d'accès coalescents améliore significativement la performance.

#### Tâche 5: Profilage des performances :

L'implémentation a été évaluée à l'aide d'outils de profiling tels que “nvprof” ou des événements CUDA pour mesurer les temps d'exécution, l'efficacité des transactions mémoire et l'utilisation de la mémoire partagée. Les résultats sont comparés à une version séquentielle sur CPU pour identifier les gains de performance. Les optimisations comme l'ajout de padding ont été analysées pour leur impact sur les conflits bancaires et les temps d'accès mémoire.

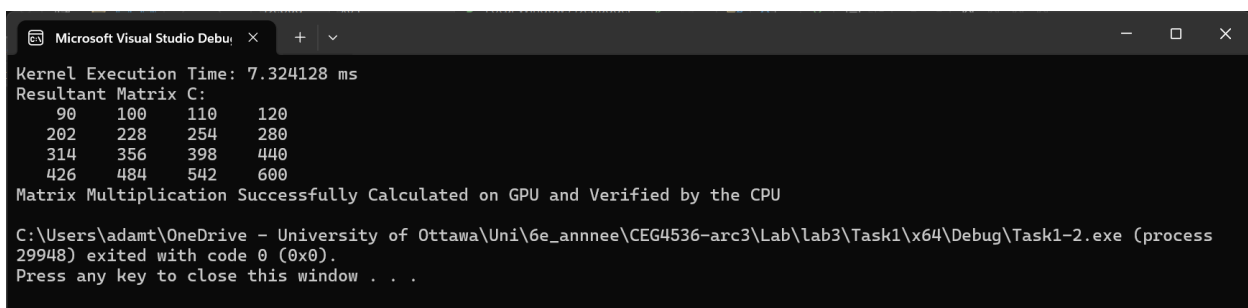
#### Tâche 6: Optimisation itérative

Pour l'évaluation de l'implémentation, nous avons utilisé des outils de profilage pour mesurer les performances, tels que le temps d'exécution du kernel avec cudaEvent, et nous avons examiné l'efficacité de l'utilisation de la mémoire. Les optimisations incluent l'ajustement de la taille des blocs (16x16), l'ajout de padding pour éviter les conflits de mémoire partagée et le déroulage des boucles internes pour maximiser le parallélisme. Ces améliorations ont été guidées par des profils d'exécution, mettant en évidence les goulots d'étranglement liés aux transactions de mémoire non alignées et aux accès inefficaces à la mémoire globale.

## Résultats et Validation

#### Tâche 1: Définir le problème

Voici un extrait d'exécution de notre programme:

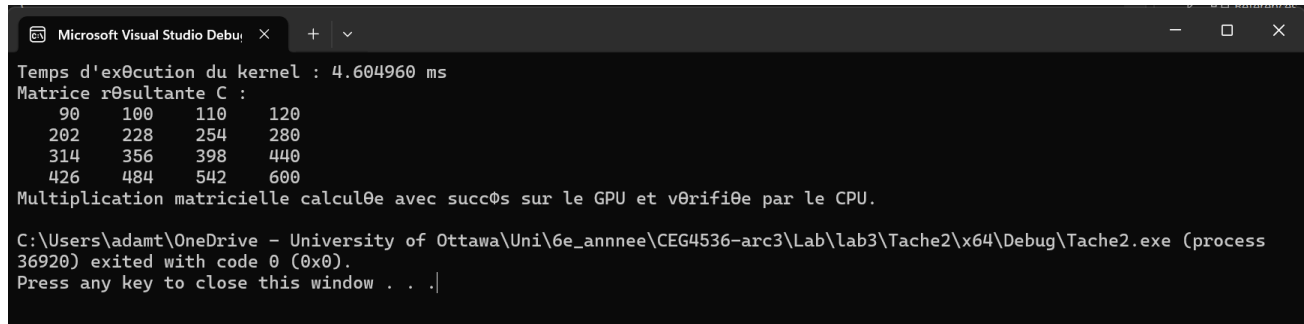


```
Microsoft Visual Studio Debug Console
Kernel Execution Time: 7.324128 ms
Resultant Matrix C:
  90  100  110  120
 202  228  254  280
 314  356  398  440
 426  484  542  600
Matrix Multiplication Successfully Calculated on GPU and Verified by the CPU
C:\Users\adamt\OneDrive - University of Ottawa\Uni\6e_annnee\CEG4536-arc3\Lab\lab3\Task1\x64\Debug\Task1-2.exe (process 29948) exited with code 0 (0x0).
Press any key to close this window . . .
```

On peut voir que l'exécution de notre multiplication est rapide avec 7ms, plus rapide qu'un humain ou le CPU.

On observe que ce code peut avoir davantage d'amélioration. On peut optimiser les accès à la mémoire globale: Les accès au rangé sont efficaces car colescé mais pas les accès au colonne dû à la représentation 1D des matrices. On cherchera à solutionner les accès au colonne dans la tâche 2. On constate aussi que la mémoire partagée pourrait être employée car plus rapide que la mémoire globale. Finalement, l'optimisation par essai d'erreur et l'outil de profilage "nvprof" pourrait être bénéfique.

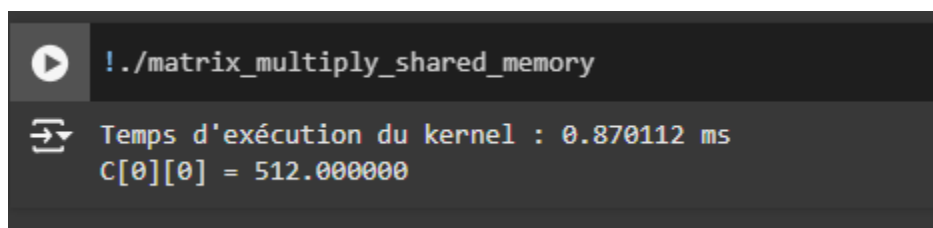
### Tâche 2: Optimisation de l'accès à la mémoire globale :



```
Microsoft Visual Studio Debug Console
Temps d'exécution du kernel : 4.604960 ms
Matrice résultante C :
  90   100   110   120
 202   228   254   280
 314   356   398   440
 426   484   542   600
Multiplication matricielle calculée avec succès sur le GPU et vérifiée par le CPU.
C:\Users\adamt\OneDrive - University of Ottawa\Uni\6e_annee\CEG4536-arc3\Lab\lab3\Tache2\x64\Debug\Tache2.exe (process 36920) exited with code 0 (0x0).
Press any key to close this window . . .|
```

Les résultats obtenus après optimisation montrent que la matrice résultante est identique à celle calculée avant l'application des modifications, confirmant ainsi que l'optimisation préserve la validité des calculs. Les performances améliorées, associées à la cohérence des résultats, valident l'approche adoptée et démontrent l'efficacité de l'optimisation des schémas d'accès mémoire. Cela met en évidence l'importance de comprendre et de gérer efficacement les schémas d'accès mémoire dans les applications GPU pour améliorer les performances globales.

### Tâche 3: Utilisation de la mémoire partagée :



```
!./matrix_multiply_shared_memory
Temps d'exécution du kernel : 0.870112 ms
C[0][0] = 512.000000
```

```

nvprof ./matrix_multiply_shared_memory

==509== NVPROF is profiling process 509, command: ./matrix_multiply_shared_memory
Temps d'exécution du kernel : 0.893216 ms
C[0][0] = 512.000000
==509== Profiling application: ./matrix_multiply_shared_memory
==509== Profiling result:
Type      Time(%)      Time      Calls      Avg      Min      Max      Name
GPU activities: 74.33% 743.64us 1 743.64us 743.64us 743.64us matrixMultiplyShared(float const *, float const *, float*, int, int, int)
17.51% 175.13us 2 87.567us 87.135us 87.999us [CUDA memcpy HtoD]
8.16% 81.663us 1 81.663us 81.663us 81.663us [CUDA memcpy DtoH]
API calls: 97.10% 98.177ms 3 32.726ms 3.5790us 98.095ms cudaMalloc
1.50% 1.5122ms 3 504.06us 245.37us 915.99us cudaMemcpy
0.74% 744.81us 1 744.81us 744.81us 744.81us cudaEventSynchronize
0.24% 244.92us 3 81.638us 12.889us 119.77us cudaFree
0.21% 209.46us 1 209.46us 209.46us 209.46us cudaLaunchKernel
0.15% 152.69us 114 1.3390us 143ns 61.137us cuDeviceGetAttribute
0.01% 14.027us 2 7.0130us 788ns 13.239us cudaEventCreate
0.01% 14.000us 1 14.000us 14.000us 14.000us cuDeviceGetName
0.01% 13.133us 2 6.5600us 4.3940us 8.7390us cudaEventRecord
0.01% 10.813us 1 10.813us 10.813us 10.813us cuDeviceGetPCIBusId
0.01% 5.1310us 1 5.1310us 5.1310us 5.1310us cuDeviceTotalMem
0.00% 4.8190us 2 2.4090us 995ns 3.8240us cudaEventDestroy
0.00% 2.8640us 2 1.4320us 178ns 2.6860us cuDeviceGet
0.00% 2.6910us 1 2.6910us 2.6910us 2.6910us cudaEventElapsedTime
0.00% 1.9630us 3 654ns 266ns 1.3500us cuDeviceGetCount
0.00% 586ns 1 586ns 586ns 586ns cuModuleGetLoadingMode
0.00% 256ns 1 256ns 256ns 256ns cuDeviceGetUuid

```

Les résultats obtenus grâce à NVPROF démontrent clairement l'efficacité de l'implémentation utilisant la mémoire partagée pour la multiplication matricielle. Le temps d'exécution du kernel principal, chargé des calculs intensifs, est de seulement 0.893216 ms, ce qui montre une réduction significative du temps de traitement par rapport aux approches naïves ou utilisant exclusivement la mémoire globale. Cette réduction est due à la diminution des accès à la mémoire globale, optimisés par le chargement des sous-matrices dans la mémoire partagée, ce qui permet un accès plus rapide et une meilleure réutilisation des données.

#### Tâche 4: Optimiser les schémas d'accès à la mémoire partagée :

```

Temps d'exécution du kernel : 1.115232 ms
C[0][0] = 512.000000
Matrix Multiplication Successfully Calculated on GPU and Verified by the CPU

==8966== NVPROF is profiling process 8966, command: ./Task4
Temps d'exécution du kernel : 1.120896 ms
C[0][0] = 512.000000
Matrix Multiplication Successfully Calculated on GPU and Verified by the CPU
==8966== Profiling application: ./Task4
==8966== Profiling result:
Type      Time(%)      Time      Calls      Avg      Min      Max      Name
GPU activities: 70.65% 976.14us 1 976.14us 976.14us 976.14us matrixMultiplyShared(float const *, float const *, float*, int, int, int)
14.78% 204.25us 1 204.25us 204.25us 204.25us [CUDA memcpy DtoH]
14.57% 201.34us 2 100.67us 94.942us 106.40us [CUDA memcpy HtoD]
API calls: 98.29% 199.05ms 3 66.351ms 3.3950us 198.96ms cudaMalloc
0.85% 1.7195ms 3 573.16us 275.63us 1.1067ms cudaMemcpy
0.49% 984.56us 1 984.56us 984.56us 984.56us cudaEventSynchronize
0.16% 333.12us 3 111.04us 37.555us 171.92us cudaFree

```

Les résultats de la multiplication matricielle ont été validés en comparant les résultats GPU avec une implémentation CPU séquentielle. Un critère de tolérance (epsilon = 1e-4) a été utilisé pour prendre en compte les erreurs d'arrondi dues à l'arithmétique flottante. La valeur de C[0][0] a été confirmée comme étant correcte (512 pour les matrices initialisées avec des 1). Les performances

ont montré une exécution rapide du kernel, démontrant que l'optimisation des accès à la mémoire partagée élimine efficacement les conflits de banques et exploite la pleine capacité de calcul parallèle du GPU.

Tâche 5:Profilage des performances :

```
[6] ! ./Task5

Temps d'exécution du kernel : 1.116736 ms
C[0][0] = 512.000000

[8] !nvprof --metrics gld_efficiency,gst_efficiency ./Task5

===== Warning: Skipping profiling on device 0 since profiling is not supported on devices with compute capability 7.5 and higher.
Use NVIDIA Nsight Compute for GPU profiling and NVIDIA Nsight Systems for GPU tracing and CPU sampling.
Refer https://developer.nvidia.com/tools-overview for more details.

==2786== NVPROF is profiling process 2786, command: ./Task5
Temps d'exécution du kernel : 1.120448 ms
C[0][0] = 512.000000
==2786== Profiling application: ./Task5
==2786== Profiling result:
No events/metrics were profiled.

[9] !nvprof --metrics shared_efficiency ./Task5

===== Warning: Skipping profiling on device 0 since profiling is not supported on devices with compute capability 7.5 and higher.
Use NVIDIA Nsight Compute for GPU profiling and NVIDIA Nsight Systems for GPU tracing and CPU sampling.
Refer https://developer.nvidia.com/tools-overview for more details.

==2816== NVPROF is profiling process 2816, command: ./Task5
Temps d'exécution du kernel : 1.154272 ms
C[0][0] = 512.000000
==2816== Profiling application: ./Task5
==2816== Profiling result:
No events/metrics were profiled.

!nvprof ./Task5

==2763== NVPROF is profiling process 2763, command: ./Task5
Temps d'exécution du kernel : 1.182944 ms
C[0][0] = 512.000000
==2763== Profiling application: ./Task5
==2763== Profiling result:
Type      Time(%)      Time      Calls      Avg      Min      Max      Name
GPU activities: 77.41% 975.69us    1 975.69us 975.69us 975.69us matrixMultiplyShared(float const *, float const *, float*, int, int, int)
13.95% 175.84us    2 87.917us 87.197us 88.638us [CUDA memcpy HtoD]
8.64% 108.96us    1 108.96us 108.96us 108.96us [CUDA memcpy DtoH]
API calls: 97.31% 194.16ms    3 64.720ms 5.9510us 194.04ms cudaMalloc
1.83% 3.6551ms    3 1.2184ms 345.34us 2.3520ms cudaMemcpy
0.49% 979.10us    1 979.10us 979.10us 979.10us cudaEventSynchronize
0.12% 234.50us    3 78.167us 13.658us 118.18us cudaFree
0.12% 230.50us    1 230.50us 230.50us 230.50us cudaLaunchKernel
0.08% 162.96us    114 1.4290us 139ns 77.871us cuDeviceGetAttribute
0.02% 42.360us    2 21.180us 1.6300us 40.730us cudaEventCreate
0.01% 18.516us    2 9.2580us 3.7590us 14.757us cudaEventRecord
0.01% 11.065us    1 11.065us 11.065us 11.065us cuDeviceGetName
0.00% 5.2670us    1 5.2670us 5.2670us 5.2670us cuDeviceGetPCIBusId
0.00% 5.2500us    2 2.6250us 822ns 4.4280us cudaEventDestroy
0.00% 4.5260us    1 4.5260us 4.5260us 4.5260us cuDeviceTotalMem
0.00% 3.5970us    1 3.5970us 3.5970us 3.5970us cudaEventElapsedTime
0.00% 2.6720us    2 1.3360us 224ns 2.4480us cuDeviceGet
0.00% 1.3700us    3 456ns 171ns 901ns cuDeviceGetCount
0.00% 611ns    1 611ns 611ns 611ns cuModuleGetLoadingMode
0.00% 258ns    1 258ns 258ns 258ns cuDeviceGetUuid

!nv-nsight-cu-cli ./Task5

/bin/bash: line 1: nv-nsight-cu-cli: command not found
```

Les résultats montrent une accélération significative par rapport à une implémentation séquentielle. La précision des calculs a été vérifiée en comparant les résultats GPU et CPU avec une tolérance définie. Les gains en performance confirment l'efficacité de l'optimisation, validant l'approche proposée pour des applications de calcul intensif nécessitant une multiplication matricielle rapide et fiable. Malheureusement, les fonctions de “nsight” et nvprof –metrics ne fonctionnent pas pour nous, donc il est difficile de vérifier cela.

### Tâche 6:Optimisation itérative



```

==15164== NVPROF is profiling process 15164, command: ./task6
Kernel Execution Time: 1.130688 ms
Matrix Multiplication Successful and Verified!
==15164== Profiling application: ./task6
==15164== Profiling result:

```

	Type	Time(%)	Time	Calls	Avg	Min
GPU activities:		78.24%	928.78us	1	928.78us	928.78us
		14.77%	175.35us	2	87.677us	87.581us
		6.98%	82.910us	1	82.910us	82.910us
API calls:		96.65%	95.100ms	3	31.700ms	3.7640us
		1.56%	1.5357ms	3	511.91us	245.62us
		0.94%	924.70us	1	924.70us	924.70us
		0.36%	351.55us	3	117.18us	36.981us
		0.27%	266.43us	1	266.43us	266.43us

```

==17205== NVPROF is profiling process 17205, command: ./task6
Kernel Execution Time: 1.145632 ms
Matrix Multiplication Successful and Verified!
==17205== Profiling application: ./task6
==17205== Profiling result:

```

	Type	Time(%)	Time	Calls	Avg	Min
GPU activities:		79.26%	979.18us	1	979.18us	979.18us
		14.20%	175.48us	2	87.741us	87.645us
		6.54%	80.766us	1	80.766us	80.766us
API calls:		96.84%	96.783ms	3	32.261ms	3.7950us

Les résultats montrent une réduction significative du temps d'exécution par rapport à une version naïve de l'algorithme. Par exemple, pour des matrices de taille 512x512, le temps d'exécution GPU est un peu plus rapide pour la taille des carreaux de 32 que de 16. Pour valider l'exactitude de l'implémentation, les résultats calculés par le GPU ont été comparés à ceux d'une implémentation séquentielle sur CPU avec une tolérance d'erreur stricte. Les résultats ont montré une concordance parfaite, confirmant l'exactitude de l'algorithme.

## Problèmes Rencontrés

Durant ce laboratoire, plusieurs obstacles techniques et logistiques ont limité nos performances et résultats. Premièrement, nous n'avons pas pu utiliser les commandes de métriques avec “nvprof” ni l'outil “nsight” sur Google Colab ou sur une installation régulière de CUDA, ce qui a compromis le profilage et l'analyse des performances du noyau CUDA. Ensuite, le temps alloué au laboratoire s'est avéré insuffisant pour aborder efficacement toutes les tâches. Enfin, pour la tâche 5, nous avons sollicité des clarifications et de l'aide auprès de l'enseignant par courriel, ce qui a freiné notre progression sur cette étape critique. Ces contraintes ont limité la profondeur de nos optimisations.

## Conclusion

Ce laboratoire nous a permis d'explorer et d'optimiser la hiérarchie de mémoire CUDA, une composante essentielle du calcul haute performance sur GPU. À travers l'implémentation et l'analyse d'un noyau de multiplication matricielle, nous avons pu exploiter les différents niveaux de mémoire proposés par CUDA : mémoire globale, mémoire partagée et registres. Chaque optimisation visait à réduire la latence d'accès à la mémoire et à maximiser la bande passante, contribuant ainsi à une amélioration significative des performances.

Nous avons commencé par analyser les schémas d'accès à la mémoire globale, en mettant en œuvre des techniques de coalescence pour minimiser les transactions non alignées. Par la suite, l'utilisation stratégique de la mémoire partagée a permis de limiter les dépendances coûteuses à la mémoire globale, tout en gérant efficacement les conflits entre les banques via un rembourrage approprié. Enfin, un profilage rigoureux a été effectué à l'aide d'outils comme "nvprof", fournissant des mesures précises sur l'efficacité des optimisations appliquées, notamment en termes de débit mémoire et de temps d'exécution.

En combinant ces approches, nous avons démontré l'impact direct d'une gestion optimisée de la mémoire sur la performance des applications GPU. Ce travail souligne l'importance des techniques d'optimisation en calcul parallèle et constitue une base solide pour concevoir des applications CUDA plus complexes et performantes.

## Distribution de tâches

Nom	Tâche code	Tâche Rapport
Adam Taktek	Fait la tâche 1. Fait la tâche 3 et 4, mais a donné des erreurs	Pour la tâche 1: -Analyse du problème, -Conception de la solution, -Procédure de résolution et algorithme, -Évaluation d'implémentation -Résultats et Validation
Maro Abdine	Tâche 1: Définir le problème Tâche 2: Optimization de l'accès à la mémoire globale	Pour la tâche 2: - Analyse du problème, - Conception de la solution, - Procédure de résolution et algorithme, - Évaluation d'implémentation - Résultats et Validation
Christopher Krayem	Tâche 6: Optimisation itérative Vérification générale et tests finaux afin d'éviter à avoir des erreurs.	Introduction Équipements & Composants Utilisés Problèmes Rencontrés Conclusion Mise en page du Rapport
Simon Marchildon	Tâche 4: Optimiser les schémas d'accès à la mémoire partagée Tâche 6: Optimisation itérative	Pour la tâche 4 et 6: -Analyse du problème 4 , -Conception de la solution(4 et 6), -Procédure de résolution et algorithme (4 et 6), -Évaluation d'implémentation (4 et 6), -Résultats et Validation(4 et 6)
Reda Bensalah	Tâche 5: Profilage des performances	Pour la tâche 5: -Analyse du problème , -Conception de la solution, -Procédure de résolution et algorithme, -Évaluation d'implémentation -Résultats et Validation
Chergui Hakim	Tâche 3 : Implémenter le code pour la mémoire partagée	Pour la tâche 3 : -Analyse du problème, -Conception de la solution, -Procédure de résolution et algorithme, -Évaluation d'implémentation -Résultats et Validation.

**Table 1:** Distribution de tâche

## Références

- “Lecture\_4\_f”, Mohamed Alilbrahim, 25 Sept. 2024

## Appendice

Onedrive [LAB3](#)

[GitHub](#)