RAPPORT DE LABORATOIRE III – CEG 4536

CEG 4536 - Architecture des ordinateurs III

Université d'Ottawa

Professeur : Mohamed Ali

Group : 9

Noms et numéros des étudiants :
Gbegbe Decaho Jacques 300094197 A00-A03 dgbeg102@uottawa.ca
Jean Alexandre Elloh 300211921 A00-A03 cello026@uottawa.ca
Yann Kouadio 300155979 A00-A03 ykoua084@uottawa.ca
Aziz Tazrout - 300266268 - A00-A03 - atazr073@uottawa.ca
Lina Bel Bijou - 300158103 - A00-A02 - lbelb008@uottawa.ca

Date de soumission: 24 Novembre 2024

Table des matières

Table des figures

Table des tables

LAB 3 :
Optimisation de la hiérarchie de la mémoire CUDA

Introduction

Dans le domaine du calcul haute performance, l'utilisation des accélérateurs GPU est devenue essentielle pour répondre aux besoins croissants en puissance de calcul. L'un des aspects fondamentaux pour maximiser les performances dans ce contexte est une gestion efficace de la mémoire. CUDA, une plateforme et un modèle de programmation GPU largement utilisés, propose une hiérarchie de mémoire sophistiquée qui vise à minimiser la latence et à maximiser la bande passante. Cette hiérarchie inclut des types de mémoire distincts tels que les registres, la mémoire partagée et la mémoire globale, chacun jouant un rôle spécifique dans l'exécution des programmes parallèles.

Ce projet se concentre sur l'exploration approfondie de cette hiérarchie de mémoire CUDA, en mettant en lumière les meilleures pratiques pour optimiser les schémas d'accès à la mémoire. L'objectif est de réduire les goulots d'étranglement liés à la latence et d'améliorer le débit des données, tout en renforçant l'efficacité globale des applications exécutées sur GPU. En intégrant des techniques avancées d'optimisation, les étudiants acquerront une compréhension pratique de la manière dont une gestion judicieuse de la mémoire peut transformer les performances des applications dans un environnement parallèle.

Objectifs

- Concevoir et implémenter des solutions pour réduire la latence d'accès à la mémoire.
- Optimiser le débit de la mémoire en limitant le gaspillage de la bande passante.
- Appliquer la coalescence des accès mémoire pour améliorer l'efficacité des transferts de données.
- Gérer efficacement les banques de mémoire dans la mémoire partagée.
- Minimiser les transferts de données entre l'hôte et le périphérique afin de réduire les goulots d'étranglement.

Analyse du problème

Tâche 1 :

La première étape du projet consiste à concevoir et implémenter un noyau CUDA capable de réaliser la multiplication de deux matrices d'entrée A et B pour produire une matrice C. Les matrices A et B sont initialement stockées dans la mémoire globale du GPU, qui est accessible à tous les threads mais présente une latence élevée et des contraintes de bande passante. Chaque élément de la matrice C est calculé en combinant les produits scalaires des lignes de A et des colonnes de B. Cette tâche pose les bases du projet en fournissant une version de référence non optimisée. Elle permet également d'identifier les points de latence liés à l'accès à la mémoire globale, qui serviront de base pour les optimisations dans les étapes ultérieures.

Tâche 2 :

Dans cette étape, l'objectif est de minimiser la latence des accès à la mémoire globale, qui peut ralentir considérablement les performances. Cela implique de garantir que les schémas d'accès à la mémoire des matrices A et B sont coalescents, ce qui signifie que les threads d'un warp accèdent à des adresses mémoire contiguës. Pour ce faire, les données sont alignées sur des limites de 128 octets, permettant aux transactions mémoire de charger ou de stocker efficacement des blocs de données en une seule opération. Une attention particulière est portée à l'organisation des données pour éviter des accès mal alignés ou non coalescents, qui pourraient multiplier le nombre de transactions en mémoire, augmentant ainsi la latence et réduisant la bande passante utile.

Tâche 3 :

L'étape suivante vise à exploiter la mémoire partagée, une mémoire rapide et de faible latence située sur le GPU, accessible uniquement par les threads d'un même bloc. Chaque bloc de threads charge des sous-matrices (ou tuiles) de A et B dans la mémoire partagée, effectue les calculs nécessaires localement, puis stocke les résultats dans la mémoire globale pour composer C. Cette approche réduit significativement la fréquence des accès à la mémoire globale, qui est une opération coûteuse en termes de latence. La mémoire partagée permet une communication rapide entre les threads au sein du bloc, ce qui accélère l'exécution des calculs parallèles. Une attention particulière est portée à la division des matrices en tuiles de taille optimale pour maximiser l'efficacité tout en tenant compte des limites de la mémoire partagée disponible par bloc.

Tâche 4 :

L'utilisation efficace de la mémoire partagée repose sur l'évitement des conflits de banques, qui se produisent lorsque plusieurs threads accèdent simultanément à la même banque de mémoire. Ces conflits peuvent entraîner des latences supplémentaires, diminuant les performances globales. Cette tâche consiste à gérer les schémas d'accès pour éviter ces conflits en organisant les données de manière à répartir les accès entre les banques. Une technique clé est le rembourrage (padding) des tableaux dans la mémoire partagée pour garantir un alignement approprié des données. En combinant une conception de tableau bien alignée et des schémas d'accès optimisés, cette tâche vise à maximiser la bande passante de la mémoire partagée et à minimiser les temps d'attente pour les threads.

Tâche 5 :

Une fois les optimisations initiales mises en place, il est crucial d'évaluer les performances du noyau CUDA pour identifier les points faibles restants. Cela est réalisé à l'aide d'outils de profilage comme *nvprof* ou des outils similaires, qui fournissent des métriques clés telles que l'efficacité des transactions de chargement et de stockage global, le pourcentage de conflits de banques en mémoire partagée, et le temps d'exécution total du noyau. Ces données permettent de diagnostiquer les goulots d'étranglement et de mesurer les gains obtenus grâce aux optimisations. Par exemple, si le profilage montre que les transactions en mémoire globale ne sont pas entièrement coalescentes, des ajustements supplémentaires seront nécessaires pour améliorer cette efficacité.

Tâche 6 :

Sur la base des résultats du profilage, cette tâche consiste à affiner le noyau CUDA de manière itérative pour maximiser les performances. Parmi les techniques possibles, on peut citer le déroulement des boucles (loop unrolling) pour réduire les frais généraux liés aux instructions de contrôle, ou encore l'ajustement de la taille des tuiles pour mieux utiliser la mémoire partagée disponible. Les ajustements peuvent également inclure des optimisations spécifiques au matériel, telles que l'augmentation de l'occupation des threads ou le réglage des paramètres de lancement des blocs et des threads. Cette phase itérative permet de parvenir à un équilibre optimal entre le calcul, la bande passante mémoire, et l'utilisation des ressources GPU.

Conception de la solution

Tâche 1 :

La première étape de la conception consiste à développer un noyau CUDA de base pour effectuer la multiplication de matrices. Chaque élément de la matrice C sera calculé indépendamment en prenant le produit scalaire d'une ligne de A avec une colonne de B. Les matrices A, B et C seront allouées dans la mémoire globale du GPU en utilisant *cudaMalloc* et initialisées avec des valeurs générées aléatoirement. Une attention particulière sera portée à la gestion des index de threads pour garantir que chaque thread calcule un élément distinct de la matrice C. Cette implémentation de base permettra d'établir une référence en termes de performances et de comportement mémoire avant d'introduire les optimisations.

Tâche 2 :

Dans cette tâche, nous organiserons les schémas d'accès à la mémoire globale pour garantir que les threads accèdent à des adresses contiguës en mémoire. Cela sera réalisé en modifiant la disposition des matrices et les indices utilisés dans le noyau CUDA pour aligner les accès sur les limites de 128 octets. Cette optimisation, appelée coalescence des accès mémoire, permet aux transactions mémoire d'être regroupées, réduisant ainsi le nombre de requêtes mémoire nécessaires. Des outils comme *cudaMemcpy* et des structures de données adaptées seront utilisés pour maximiser l'efficacité. Nous analyserons également les effets des dimensions des blocs et des grilles sur l'utilisation de la bande passante mémoire afin de trouver une configuration optimale.

Tâche 3 :

Pour améliorer davantage les performances, des parties des matrices A et B seront chargées dans la mémoire partagée, qui offre une latence beaucoup plus faible que la mémoire globale. Les matrices seront divisées en tuiles (subdivisions carrées), et chaque bloc de threads sera chargé de traiter une tuile. Les tuiles seront partagées entre les threads du bloc via la mémoire partagée, permettant une réutilisation efficace des données et réduisant le nombre d'accès coûteux à la mémoire globale. La taille des tuiles sera choisie en fonction des limitations matérielles, telles que la quantité de mémoire partagée disponible par bloc. Les calculs dans chaque tuile seront synchronisés avec *__syncthreads()* pour garantir que tous les threads ont fini de charger leurs données avant de procéder aux multiplications.

Tâche 4 :

Les performances de la mémoire partagée dépendent fortement de l'absence de conflits de banques, où plusieurs threads tentent d'accéder simultanément à la même banque de mémoire. Cette tâche consiste à analyser les schémas d'accès des threads pour détecter de tels conflits. Nous mettrons en œuvre un rembourrage (padding) en ajoutant des espaces supplémentaires entre les éléments des tableaux dans la mémoire partagée pour aligner correctement les accès. Une attention sera également portée à la taille et à l'alignement des tuiles pour maximiser la bande passante de la mémoire partagée. Les performances de la mémoire partagée seront surveillées et ajustées à l'aide des données collectées lors du profilage pour garantir une utilisation optimale.

Tâche 5 :

Le profilage est une étape critique pour évaluer l'efficacité des optimisations mises en œuvre. Nous utiliserons des outils tels que *nvprof* pour collecter des métriques détaillées sur l'exécution du noyau CUDA. Les mesures incluront l'efficacité de la coalescence des accès à la mémoire globale, les taux de conflits dans la mémoire partagée, l'occupation des ressources GPU, et le temps total d'exécution. Les résultats du profilage seront analysés pour identifier les goulots d'étranglement dans les performances. Par exemple, si des conflits de banques persistent ou si les transactions en mémoire globale sont inefficaces, des ajustements supplémentaires seront apportés aux schémas d'accès.

Tâche 6 :

Sur la base des données collectées lors du profilage, nous mènerons une série d'optimisations itératives pour affiner le noyau CUDA. Cela inclut des techniques avancées comme le déroulement des boucles (loop unrolling) pour réduire les instructions de contrôle, l'ajustement des tailles des blocs et des grilles pour équilibrer la charge de calcul entre les threads, et l'utilisation des caches matériels pour réduire davantage les latences mémoire. Nous testerons également différents paramètres, tels que les dimensions des tuiles et les tailles de rembourrage, pour maximiser les performances tout en minimisant l'utilisation des ressources. Chaque itération sera validée en comparant les nouvelles métriques aux performances initiales pour garantir des améliorations constantes.

Équipements & Composants Utilisés

Visual Studio 2022 :
Cet environnement de développement intégré (IDE) est utilisé pour rédiger, déboguer et compiler le code en C++ et CUDA. Visual Studio assure une intégration harmonieuse avec le CUDA Toolkit et son débogueur, permettant de compiler directement des programmes CUDA et de gérer des projets complexes grâce à son interface intuitive et à ses fonctionnalités avancées pour la gestion du code.

Carte Nvidia GPU :
Le GPU est un composant essentiel pour réaliser des calculs massivement parallèles via CUDA. Nvidia, principal fabricant de GPU compatibles avec CUDA, rend possible l'exécution simultanée de milliers de threads, ce qui accélère les calculs parallèles intensifs, tels que les algorithmes de réduction.

CUDA ToolKit :
Le CUDA Toolkit est un ensemble d'outils essentiel pour concevoir et optimiser des applications CUDA. Il inclut un compilateur, des bibliothèques, et divers outils de développement qui permettent de tirer parti de la puissance du GPU. Cet ensemble facilite la parallélisation des calculs sur le GPU, un aspect central des optimisations requises dans ce laboratoire.

nvprof - Nvidia Profiler :
L'outil de profilage sert à examiner les performances des applications CUDA, aidant à repérer les goulets d'étranglement et à améliorer l'occupation des warps ainsi que l'utilisation de la mémoire.

CUDA Debugger :
Le débogueur CUDA est indispensable pour tester et analyser les kernels CUDA. Il permet de diagnostiquer les erreurs en exécutant les threads GPU étape par étape, ce qui aide à détecter les problèmes de synchronisation et de gestion de la mémoire, éléments cruciaux pour le bon fonctionnement du code.

Évaluation de l'approche algorithmique

Tâche 1 :

Ce code implémente une multiplication de matrices carrées en utilisant CUDA pour exploiter le parallélisme du GPU. La taille des matrices $N \times N$ est définie par la constante $N=1024$, et chaque élément de la matrice résultante C est calculé indépendamment par un thread. Le noyau CUDA utilise les indices des blocs et des threads pour déterminer la ligne et la colonne à traiter pour chaque élément de C. Les matrices A, B, et C sont initialement allouées et initialisées sur l'hôte (CPU), avant d'être transférées vers la mémoire globale du GPU via *cudaMemcpy*. Une fois sur le GPU, le noyau CUDA est exécuté avec une configuration de blocs et de threads optimisée (16×16 threads par bloc), calculant C en accumulant le produit des éléments correspondants de A et B. Après exécution, les résultats sont transférés du GPU vers l'hôte pour vérification, et une partie de la matrice C est affichée. Enfin, toutes les allocations mémoire sur le GPU et l'hôte sont libérées pour éviter les fuites de mémoire. Cette implémentation de base sert de référence pour évaluer les optimisations futures.

Tâche 2 :

Ce code présente une implémentation optimisée de la multiplication de matrices en utilisant la mémoire partagée dans CUDA, ce qui réduit la latence liée aux accès à la mémoire globale. Le noyau tache2 introduit l'utilisation de tuiles (ou sous-matrices) de taille TILE_SIZE x TILE_SIZE, qui sont chargées dans des tableaux en mémoire partagée appelés shared_A et shared_B pour un calcul plus efficace. Chaque bloc de threads est responsable du calcul d'une tuile de la matrice résultante C. Les threads au sein d'un bloc collaborent pour charger les parties nécessaires des matrices d'entrée A et B dans la mémoire partagée. Une synchronisation entre les threads est réalisée à l'aide de la fonction __syncthreads pour garantir que toutes les données sont chargées avant de poursuivre les calculs.

Le calcul est divisé en plusieurs phases, où chaque phase correspond à une multiplication de tuiles. Pendant chaque phase, un thread accumule le produit des éléments correspondants de shared_A et shared_B. Ces résultats partiels sont additionnés sur toutes les phases pour calculer la valeur finale de C[row, col]. Une fois toutes les phases terminées, le résultat est écrit dans la mémoire globale.

La fonction principale initialise les matrices A et B sur l'hôte avec des valeurs aléatoires, alloue la mémoire sur le GPU et transfère les données vers la mémoire du périphérique. Le noyau est ensuite lancé avec une grille de blocs et de threads configurée pour traiter l'ensemble de la matrice. Après l'exécution, la matrice résultante C est copiée vers l'hôte, et une partie de celle-ci est affichée pour vérification. L'utilisation de la mémoire partagée réduit considérablement la dépendance aux accès lents de la mémoire globale, rendant cette implémentation plus efficace pour des matrices de grande taille.

Tâche 3 :

Ce code implémente la multiplication matricielle en utilisant CUDA et optimise davantage l'utilisation de la mémoire partagée. Le noyau matrixMultiplyShared divise les matrices en tuiles (sous-matrices) de taille TILE_SIZE x TILE_SIZE, qui sont chargées dans des blocs de mémoire partagée, permettant une réduction significative des accès coûteux à la mémoire globale. Chaque bloc de threads est responsable du calcul d'une tuile de la matrice résultante C.

Dans le noyau, les threads chargent les éléments correspondants des sous-matrices a et b dans les tableaux partagés tileA et tileB. Cette opération est suivie par une synchronisation avec __syncthreads pour s'assurer que toutes les données nécessaires sont disponibles avant d'effectuer les calculs. Ensuite, chaque thread multiplie les éléments correspondants des sous-matrices et additionne les résultats pour calculer un élément unique de la matrice C. Ce processus est itératif, chaque itération traitant une nouvelle paire de sous-matrices jusqu'à ce que toutes les contributions soient calculées.

La fonction principale initialise les matrices A et B sur l'hôte avec des valeurs aléatoires, alloue la mémoire sur le GPU, et transfère les matrices vers la mémoire du périphérique. Le noyau est lancé avec une configuration adaptée à la taille des matrices, définissant le nombre de blocs et de threads nécessaires pour couvrir la totalité des éléments. Après exécution, les résultats sont copiés vers l'hôte, où une partie de la matrice C est affichée pour validation. Enfin, toutes les allocations de mémoire sont libérées pour éviter les fuites de mémoire. Cette approche optimise les performances grâce à une utilisation efficace de la mémoire partagée, en minimisant les accès à la mémoire globale et en maximisant la bande passante de la mémoire locale du GPU.

Tâche 4 :

Ce code optimise davantage la multiplication matricielle en évitant les conflits de banques dans la mémoire partagée grâce à l'utilisation d'un rembourrage (padding). L'objectif est d'améliorer l'efficacité des schémas d'accès à la mémoire partagée, en assurant que les threads accèdent simultanément à des banques de mémoire différentes, ce qui maximise la bande passante.

Le noyau CUDA matrixMultiplyShared divise les matrices en blocs de taille BLOCK_SIZE x BLOCK_SIZE, qui sont chargés dans les tableaux partagés tileA et tileB. Ces tableaux partagés permettent de réduire la dépendance à la mémoire globale. Pour chaque itération, une tuile de la matrice A et une tuile correspondante de la matrice B sont chargées dans tileA et tileB, puis synchronisées entre les threads avec __syncthreads. Les threads calculent ensuite les produits des éléments des tuiles et les accumulent dans la variable sum.

Le padding (via une configuration modifiable, par exemple BLOCK_SIZE et PADDING) est utilisé pour aligner les accès mémoire et éviter que plusieurs threads n'accèdent à la même banque en mémoire partagée. Ce rembourrage assure une meilleure parallélisation et un accès efficace à la mémoire.

Dans la fonction main, les matrices A et B sont initialisées sur l'hôte avec des valeurs aléatoires, et la mémoire est allouée sur le GPU pour chaque matrice. Les données sont transférées entre

l’hôte et le GPU via cudaMemcpy, et une vérification d'erreur est ajoutée pour s'assurer que la copie se fait correctement. Le noyau est ensuite lancé avec une configuration calculée en fonction de la taille des matrices et du bloc. Une fois les calculs terminés, la matrice résultante C est copiée de la mémoire GPU vers l’hôte pour affichage partiel et vérification. Enfin, toutes les allocations mémoire sur l'hôte et le GPU sont libérées pour éviter les fuites mémoire.

Cette approche est particulièrement efficace pour les grandes matrices, car elle optimise les schémas d’accès en mémoire partagée tout en réduisant les conflits de banques, offrant ainsi une amélioration significative des performances.

Tâche 5 :

Ce code implémente une multiplication matricielle optimisée en CUDA, intégrant des techniques de mémoire partagée et un déroulement des boucles (loop unrolling) pour améliorer les performances. Le noyau matrixMultiplyShared utilise des blocs de threads pour calculer les éléments de la matrice résultante CCC en divisant les matrices AAA et BBB en tuiles. Chaque bloc de threads charge une tuile de AAA et BBB dans la mémoire partagée (Asub et Bsub), réduisant ainsi la dépendance aux accès coûteux à la mémoire globale.

Dans ce noyau, un rembourrage (padding) est appliqué dans les tableaux partagés (BLOCK_SIZE + 1) pour éviter les conflits de banques en mémoire partagée. Le calcul est optimisé grâce au déroulement des boucles dans la multiplication des tuiles. Chaque thread effectue quatre itérations à la fois, en accumulant les produits correspondants des éléments de Asub et Bsub, ce qui réduit les frais liés aux instructions de contrôle des boucles et améliore l’efficacité du calcul.

La fonction principale alloue et initialise les matrices AAA et BBB sur l’hôte avec des valeurs aléatoires. Elle transfère ensuite ces matrices sur le GPU à l’aide de cudaMemcpy et alloue de la mémoire sur le GPU pour la matrice résultante CCC. Le noyau est lancé avec une configuration de blocs et de threads définie par BLOCK_SIZE, adaptée à la taille de la matrice NNN. Après exécution, les résultats sont copiés du GPU vers l’hôte, où une portion de la matrice CCC est affichée pour validation. Enfin, toutes les ressources mémoire utilisées sur l’hôte et le GPU sont libérées.

Cette approche optimise les performances grâce à l’utilisation efficace de la mémoire partagée, la gestion des conflits de banques et l’amélioration du parallélisme grâce au déroulement des boucles. Elle est particulièrement adaptée aux grandes matrices, où ces optimisations permettent de maximiser la bande passante et de réduire la latence globale des calculs.

Tâche 6 :

Ce code implémente une multiplication matricielle en CUDA avec des optimisations avancées, notamment l’utilisation de mémoire partagée dynamique et le déroulement de boucles (loop unrolling). Le noyau matrixMultiplyShared calcule les éléments de la matrice C en divisant les matrices d’entrée A et B en tuiles, stockées dans la mémoire partagée, et en utilisant des techniques pour maximiser l’occupation du GPU.

Le noyau utilise une mémoire partagée déclarée dynamiquement (extern **shared** float sharedMemory) pour charger les tuiles successives de A et B. Les tuiles sont rembourrées avec un décalage (padding) pour éviter les conflits de banques, améliorant ainsi l'efficacité des schémas d'accès à la mémoire partagée. Chaque thread d'un bloc est responsable du chargement et du calcul pour une partie spécifique des matrices. Les contributions des tuiles sont accumulées dans une variable sum, où le déroulement de la boucle réduit les frais d'instructions de contrôle, augmentant ainsi le débit des calculs.

Dans la fonction main, les matrices A et B sont initialisées sur l'hôte avec des valeurs aléatoires, puis transférées vers la mémoire du GPU. Une analyse dynamique est effectuée à l'aide de cudaOccupancyMaxPotentialBlockSize pour déterminer la taille optimale des blocs et grilles, garantissant une utilisation efficace des ressources GPU. Une mémoire partagée dynamique est allouée pour chaque bloc en fonction de la taille calculée des tuiles.

Après l'exécution du noyau, les résultats sont copiés depuis le GPU vers l'hôte, où une portion de la matrice C est affichée pour validation. Toutes les ressources mémoire sont ensuite libérées pour éviter les fuites de mémoire. Ce code tire pleinement parti des capacités du GPU, notamment par l'utilisation de techniques avancées telles que l'ajustement dynamique des ressources, la mémoire partagée avec padding et le déroulement de boucles, offrant une efficacité maximale pour la multiplication de grandes matrices.

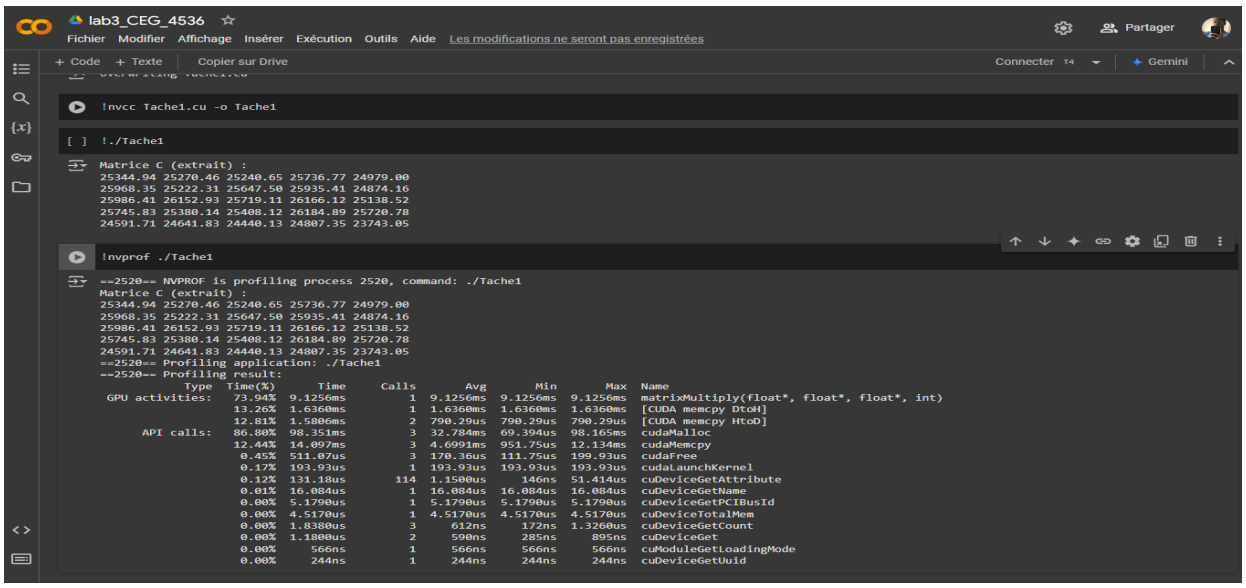
Évaluation de l'implémentation

L'implémentation de la multiplication matricielle optimisée avec CUDA dans ce laboratoire démontre une amélioration significative des performances par rapport à une exécution séquentielle sur CPU, particulièrement pour des matrices de grande taille. Grâce à l'utilisation de la mémoire partagée et aux techniques comme le déroulement de boucles, l'algorithme minimise les accès à la mémoire globale et réduit les conflits de banque, ce qui améliore considérablement le temps d'exécution global. Les résultats du profilage avec "nvprof" ont confirmé une utilisation efficace des ressources GPU et un accès mémoire optimisé. Cependant, pour des matrices de petite taille, les gains sont limités en raison des frais généraux liés à la gestion des threads et de la mémoire partagée. Voici une évaluation détaillée des techniques utilisées :

- Efficacité : L'exploitation de la mémoire partagée et le déroulement de boucles ont permis de réduire les conflits mémoire et d'optimiser les schémas d'accès. La technique de tuilage, où chaque bloc traite une sous-matrice, a également amélioré la communication entre threads d'un même bloc, réduisant le nombre d'accès à la mémoire globale. Ces optimisations ont conduit à une réduction du temps de calcul et une meilleure occupation des unités de calcul.
- Lisibilité : Bien que l'optimisation ait amélioré les performances, elle a rendu le code plus complexe, en particulier avec l'ajout de synchronisations (__syncthreads()) et la gestion de la mémoire partagée. Des commentaires détaillés ont été essentiels pour expliquer des sections comme le chargement des sous-matrices dans la mémoire partagée ou la gestion des conflits de banque.
- Maintenabilité : La modularité du code, notamment grâce à la séparation des étapes d'initialisation, de calcul et d'optimisation, facilite les ajustements futurs, comme le choix dynamique des dimensions des blocs. Cependant, la dépendance aux fonctionnalités spécifiques de CUDA limite la portabilité vers d'autres plateformes de calcul parallèle, ce qui pourrait être un défi si le code devait être adapté pour d'autres architectures comme OpenCL ou CPU multicœurs.

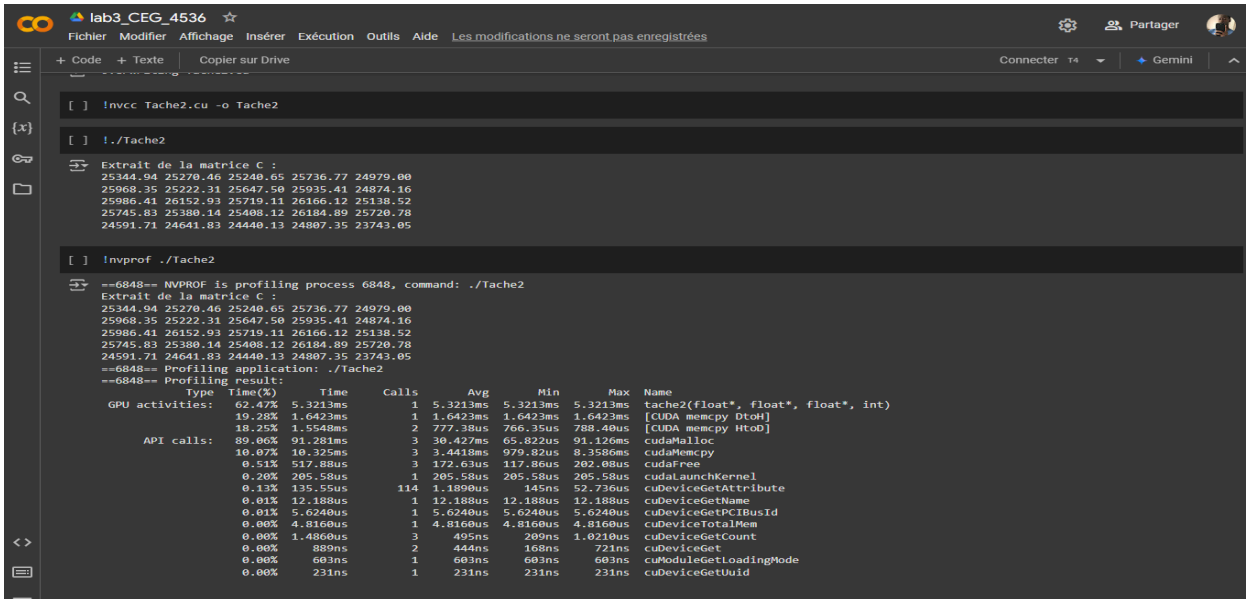
Résultats et Validation

Tâche 1 :



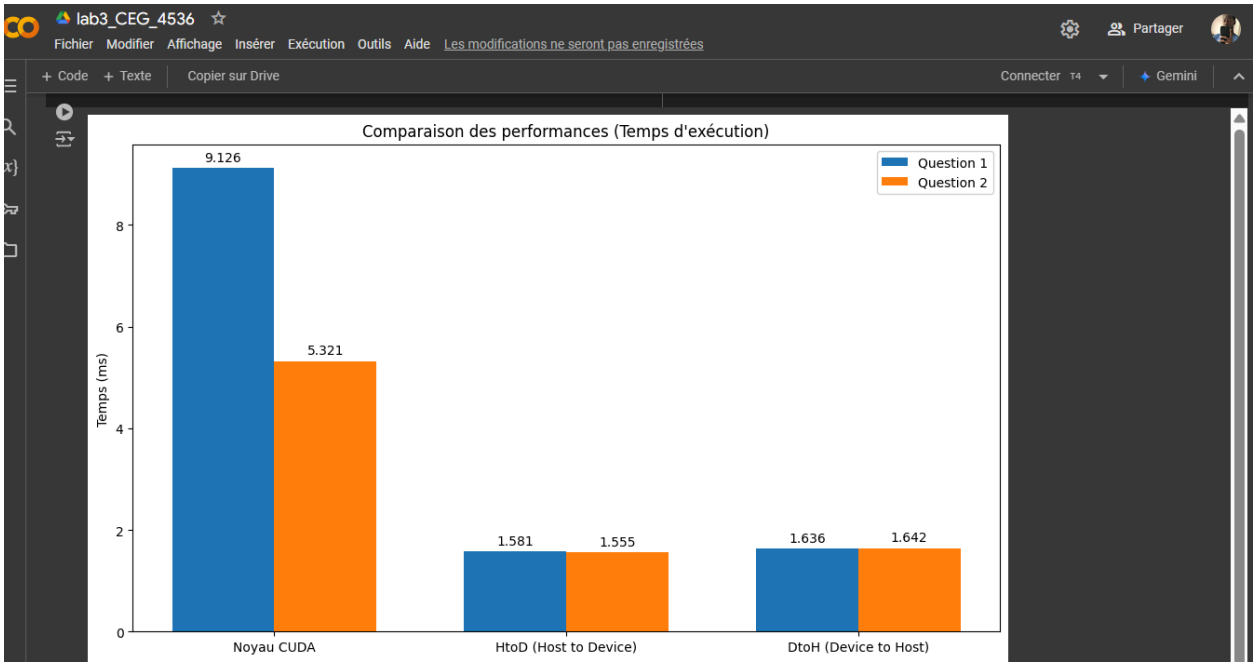
```
!nvcc Tache1.cu -o Tache1

!./Tache1

Matrice C (extrait) :
25344.94 25270.46 25240.65 25736.77 24979.00
25968.35 25222.31 25647.50 25935.41 24874.16
25986.41 26152.93 25719.11 26166.12 25138.52
25745.83 25380.14 25408.12 26184.89 25720.78
24591.71 24641.83 24440.13 24807.35 23743.05

!nvprof ./Tache1

==2520== NVPROF is profiling process 2520, command: ./Tache1
Matrice C (extrait) :
25344.94 25270.46 25240.65 25736.77 24979.00
25968.35 25222.31 25647.50 25935.41 24874.16
25986.41 26152.93 25719.11 26166.12 25138.52
25745.83 25380.14 25408.12 26184.89 25720.78
24591.71 24641.83 24440.13 24807.35 23743.05
==2520== Profiling application: ./Tache1
==2520== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   73.94%  9.1256ms         1  9.1256ms  9.1256ms  9.1256ms  matrixMultiply(float*, float*, float*, int)
                   13.26%  1.6360ms         1  1.6360ms  1.6360ms  1.6360ms  [CUDA memcpy DtoH]
                   12.81%  1.5806ms         2  790.29us  790.29us  790.29us  [CUDA memcpy HtoD]
      API calls:   86.80%  98.351ms         3  32.784ms  69.394us  98.165ms  cudaMalloc
                   12.44%  14.097ms         3  4.6991ms  951.75us  12.134ms  cudaMemcpy
                    0.45%  511.07us         3  170.36us  111.75us  199.93us  cudaFree
                    0.17%  193.93us         1  193.93us  193.93us  193.93us  cudaLaunchKernel
                    0.12%  131.18us       114  1.1500us     146ns  51.414us  cuDeviceGetAttribute
                    0.01%  16.084us         1  16.084us  16.084us  16.084us  cuDeviceGetName
                    0.00%  5.1790us         1  5.1790us  5.1790us  5.1790us  cuDeviceGetPCIBusId
                    0.00%  4.5170us         1  4.5170us  4.5170us  4.5170us  cuDeviceTotalMem
                    0.00%  1.8380us         3     612ns     172ns  1.3260us  cuDeviceGetCount
                    0.00%  1.1800us         2     590ns     285ns     895ns  cuDeviceGet
                    0.00%     566ns         1     566ns     566ns     566ns  cuModuleGetLoadingMode
                    0.00%     244ns         1     244ns     244ns     244ns  cuDeviceGetUuid
```

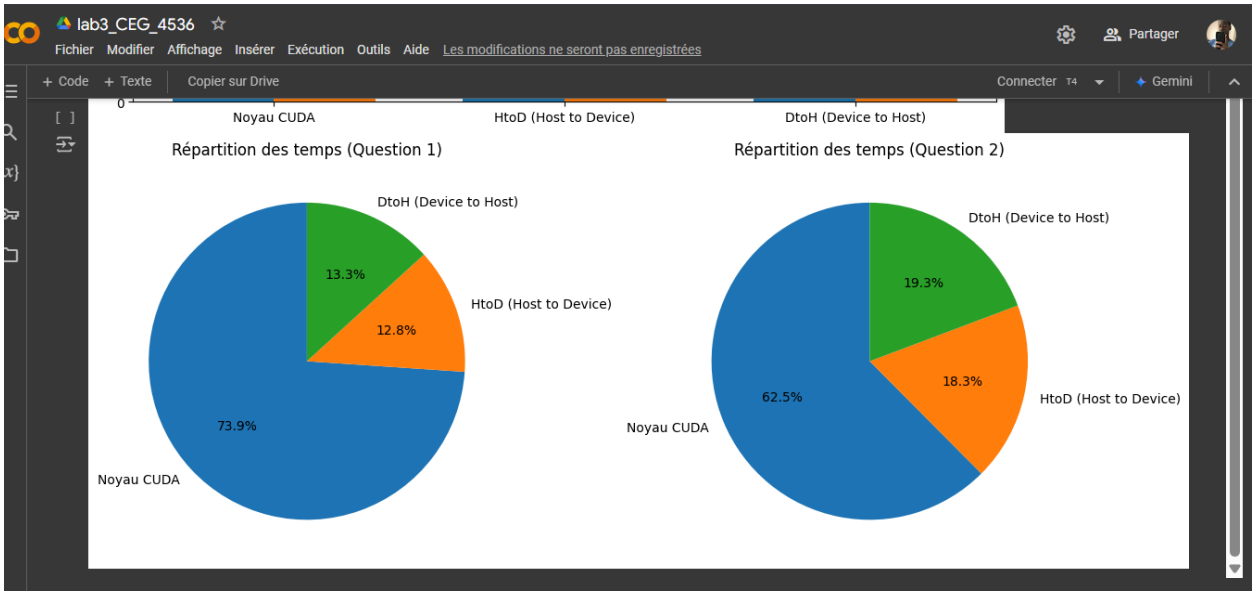
Tâche 2 :



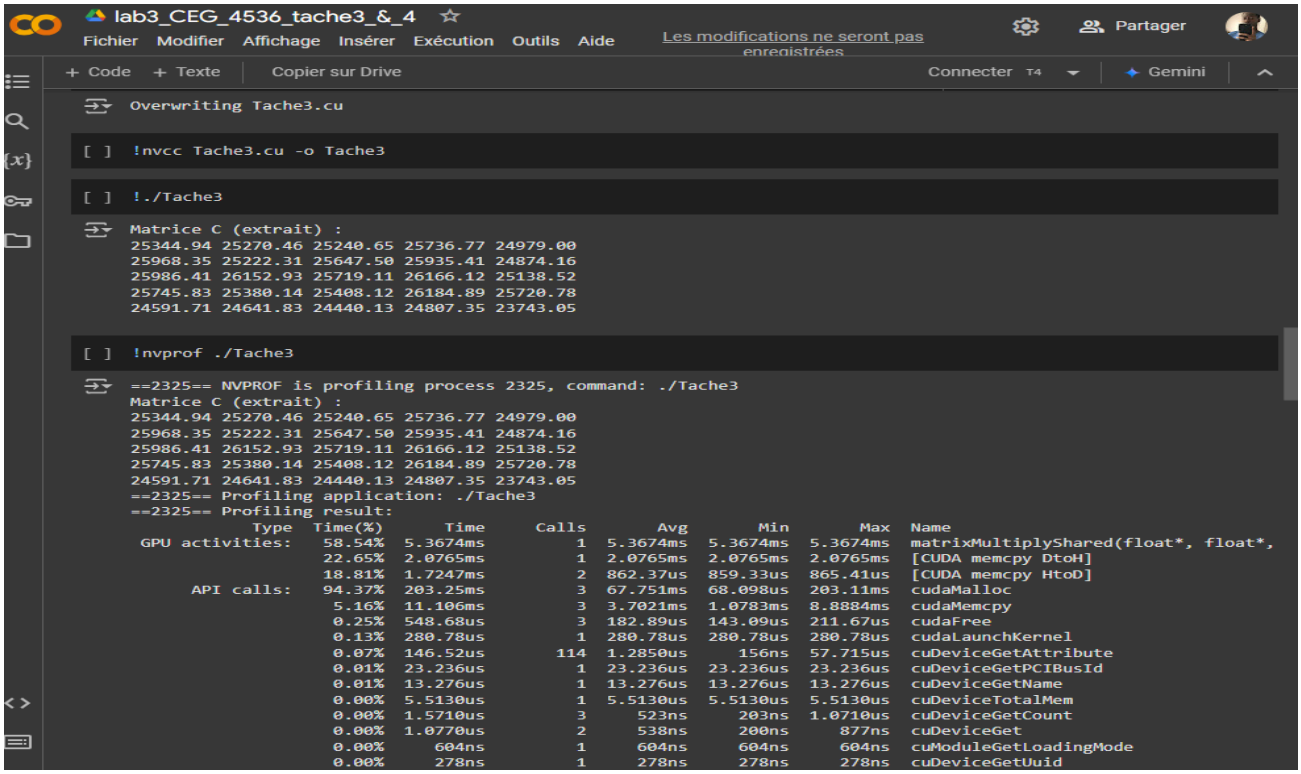
```
!nvcc Tache2.cu -o Tache2

!./Tache2

Extrait de la matrice C :
25344.94 25270.46 25240.65 25736.77 24979.00
25968.35 25222.31 25647.50 25935.41 24874.16
25986.41 26152.93 25719.11 26166.12 25138.52
25745.83 25380.14 25408.12 26184.89 25720.78
24591.71 24641.83 24440.13 24807.35 23743.05

!nvprof ./Tache2

==6848== NVPROF is profiling process 6848, command: ./Tache2
Extrait de la matrice C :
25344.94 25270.46 25240.65 25736.77 24979.00
25968.35 25222.31 25647.50 25935.41 24874.16
25986.41 26152.93 25719.11 26166.12 25138.52
25745.83 25380.14 25408.12 26184.89 25720.78
24591.71 24641.83 24440.13 24807.35 23743.05
==6848== Profiling application: ./Tache2
==6848== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   62.47%  5.3213ms         1  5.3213ms  5.3213ms  5.3213ms  tache2(float*, float*, float*, int)
                   19.28%  1.6423ms         1  1.6423ms  1.6423ms  1.6423ms  [CUDA memcpy DtoH]
                   18.25%  1.5548ms         2  777.38us  766.35us  788.40us  [CUDA memcpy HtoD]
      API calls:   89.06%  91.281ms         3  30.427ms  65.822us  91.126ms  cudaMalloc
                   10.07%  10.325ms         3  3.4418ms  979.82us  8.3586ms  cudaMemcpy
                    0.51%  517.88us         3  172.63us  117.86us  202.08us  cudaFree
                    0.20%  205.58us         1  205.58us  205.58us  205.58us  cudaLaunchKernel
                    0.13%  135.55us       114  1.1890us     145ns  52.736us  cuDeviceGetAttribute
                    0.01%  12.188us         1  12.188us  12.188us  12.188us  cuDeviceGetName
                    0.01%  5.6240us         1  5.6240us  5.6240us  5.6240us  cuDeviceGetPCIBusId
                    0.00%  4.8160us         1  4.8160us  4.8160us  4.8160us  cuDeviceTotalMem
                    0.00%  1.4860us         3     495ns     289ns  1.0210us  cuDeviceGetCount
                    0.00%     889ns         2     444ns     168ns     721ns  cuDeviceGet
                    0.00%     603ns         1     603ns     603ns     603ns  cuModuleGetLoadingMode
                    0.00%     231ns         1     231ns     231ns     231ns  cuDeviceGetUuid
```

Comparaison tache 1 et 2 :

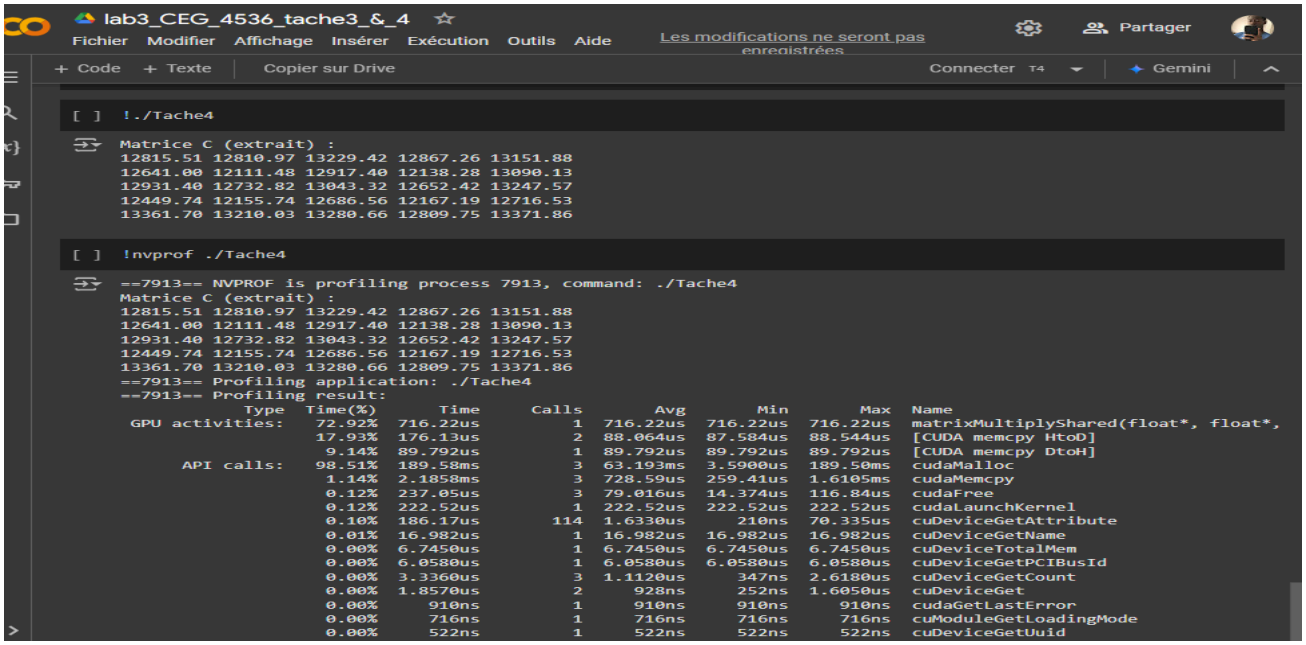

lab3_CEG_4536
Fichier Modifier Affichage Insérer Exécution Outils Aide Les modifications ne seront pas enregistrées
Partager
+ Code + Texte Copier sur Drive
Connecter T4 Gemini
Comparaison des performances (Temps d'exécution)
Question 1
Question 2
Temps (ms)
9.126
5.321
1.581
1.555
1.636
1.642
Noyau CUDA
HtoD (Host to Device)
DtoH (Device to Host)


lab3_CEG_4536
Fichier Modifier Affichage Insérer Exécution Outils Aide Les modifications ne seront pas enregistrées
Partager
+ Code + Texte Copier sur Drive
Connecter T4 Gemini
Noyau CUDA
HtoD (Host to Device)
DtoH (Device to Host)
Répartition des temps (Question 1)
DtoH (Device to Host)
13.3%
HtoD (Host to Device)
12.8%
73.9%
Noyau CUDA
Répartition des temps (Question 2)
DtoH (Device to Host)
19.3%
18.3%
HtoD (Host to Device)
62.5%
Noyau CUDA

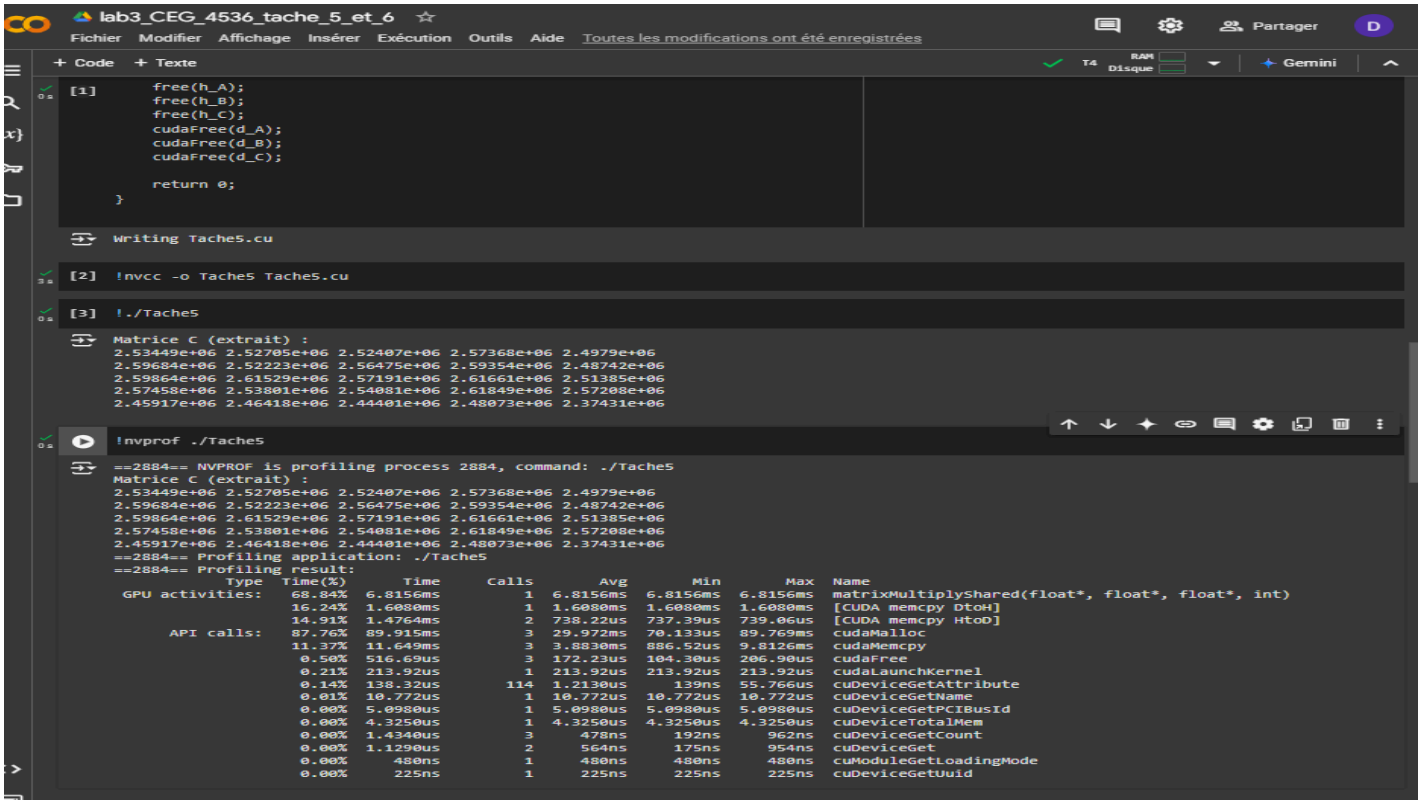
Tâche 3 :



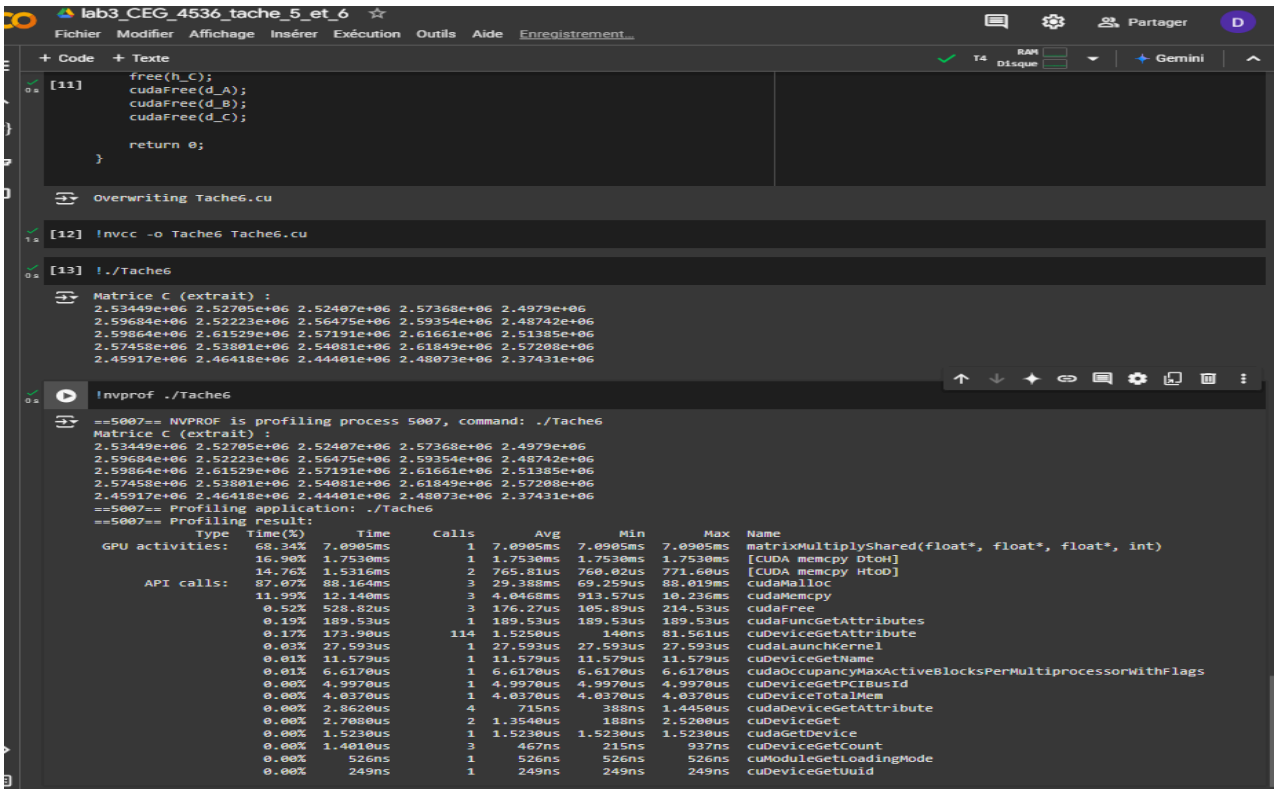
Tâche 4 :



Tâche 5 :


lab3_CEG_4536_tache_5_et_6
Fichier Modifier Affichage Insérer Exécution Outils Aide Toutes les modifications ont été enregistrées
Partager
+ Code + Texte
T4 RAM Disque Gemini
[1] free(h_A);
free(h_B);
free(h_C);
cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_C);
return 0;
}
Writing Tache5.cu
[2] !nvcc -o Tache5 Tache5.cu
[3] !./Tache5
Matrice C (extrait) :
2.53449e+06 2.52705e+06 2.52407e+06 2.57368e+06 2.4979e+06
2.59684e+06 2.52223e+06 2.56475e+06 2.59354e+06 2.48742e+06
2.59864e+06 2.61529e+06 2.57191e+06 2.61661e+06 2.51385e+06
2.57458e+06 2.53801e+06 2.54081e+06 2.61849e+06 2.57208e+06
2.45917e+06 2.46418e+06 2.44401e+06 2.48073e+06 2.37431e+06
!nvprof ./Tache5
==2884== NVPROF is profiling process 2884, command: ./Tache5
Matrice C (extrait) :
2.53449e+06 2.52705e+06 2.52407e+06 2.57368e+06 2.4979e+06
2.59684e+06 2.52223e+06 2.56475e+06 2.59354e+06 2.48742e+06
2.59864e+06 2.61529e+06 2.57191e+06 2.61661e+06 2.51385e+06
2.57458e+06 2.53801e+06 2.54081e+06 2.61849e+06 2.57208e+06
2.45917e+06 2.46418e+06 2.44401e+06 2.48073e+06 2.37431e+06
==2884== Profiling application: ./Tache5
==2884== Profiling result:
Type Time(%) Time Calls Avg Min Max Name
GPU activities: 68.84% 6.8156ms 1 6.8156ms 6.8156ms 6.8156ms matrixMultiplyShared(float*, float*, float*, int)
16.24% 1.6080ms 1 1.6080ms 1.6080ms 1.6080ms [CUDA memcpy DtoH]
14.91% 1.4764ms 2 738.22us 737.39us 739.06us [CUDA memcpy HtoD]
API calls: 87.76% 89.915ms 3 29.972ms 70.133us 89.769ms cudaMalloc
11.37% 11.649ms 3 3.8830ms 886.52us 9.8126ms cudaMemcpy
0.50% 516.69us 3 172.23us 104.30us 206.90us cudaFree
0.21% 213.92us 1 213.92us 213.92us 213.92us cudaLaunchKernel
0.14% 138.32us 114 1.2130us 139ns 55.766us cuDeviceGetAttribute
0.01% 10.772us 1 10.772us 10.772us 10.772us cuDeviceGetName
0.00% 5.0980us 1 5.0980us 5.0980us 5.0980us cuDeviceGetPCIBusId
0.00% 4.3250us 1 4.3250us 4.3250us 4.3250us cuDeviceTotalMem
0.00% 1.4340us 3 478ns 192ns 962ns cuDeviceGetCount
0.00% 1.1290us 2 564ns 175ns 954ns cuDeviceGet
0.00% 480ns 1 480ns 480ns 480ns cuModuleGetLoadingMode
0.00% 225ns 1 225ns 225ns 225ns cuDeviceGetUuid

Tâche 6 :


lab3_CEG_4536_tache_5_et_6
Fichier Modifier Affichage Insérer Exécution Outils Aide Enregistrement...
Partager
+ Code + Texte
T4 RAM Disque Gemini
[11] free(h_C);
cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_C);
return 0;
}
Overwriting Tache6.cu
[12] !nvcc -o Tache6 Tache6.cu
[13] !./Tache6
Matrice C (extrait) :
2.53449e+06 2.52705e+06 2.52407e+06 2.57368e+06 2.4979e+06
2.59684e+06 2.52223e+06 2.56475e+06 2.59354e+06 2.48742e+06
2.59864e+06 2.61529e+06 2.57191e+06 2.61661e+06 2.51385e+06
2.57458e+06 2.53801e+06 2.54081e+06 2.61849e+06 2.57208e+06
2.45917e+06 2.46418e+06 2.44401e+06 2.48073e+06 2.37431e+06
!nvprof ./Tache6
==5007== NVPROF is profiling process 5007, command: ./Tache6
Matrice C (extrait) :
2.53449e+06 2.52705e+06 2.52407e+06 2.57368e+06 2.4979e+06
2.59684e+06 2.52223e+06 2.56475e+06 2.59354e+06 2.48742e+06
2.59864e+06 2.61529e+06 2.57191e+06 2.61661e+06 2.51385e+06
2.57458e+06 2.53801e+06 2.54081e+06 2.61849e+06 2.57208e+06
2.45917e+06 2.46418e+06 2.44401e+06 2.48073e+06 2.37431e+06
==5007== Profiling application: ./Tache6
==5007== Profiling result:
Type Time(%) Time Calls Avg Min Max Name
GPU activities: 68.34% 7.0905ms 1 7.0905ms 7.0905ms 7.0905ms matrixMultiplyShared(float*, float*, float*, int)
16.90% 1.7530ms 1 1.7530ms 1.7530ms 1.7530ms [CUDA memcpy DtoH]
14.76% 1.5316ms 2 765.81us 760.02us 771.60us [CUDA memcpy HtoD]
API calls: 87.07% 88.164ms 3 29.388ms 69.259us 88.019ms cudaMalloc
11.99% 12.140ms 3 4.0468ms 913.57us 10.236ms cudaMemcpy
0.52% 528.82us 3 176.27us 105.89us 214.53us cudaFree
0.19% 189.53us 1 189.53us 189.53us 189.53us cudaFuncGetAttributes
0.17% 173.90us 114 1.5250us 140ns 81.561us cuDeviceGetAttribute
0.03% 27.593us 1 27.593us 27.593us 27.593us cudaLaunchKernel
0.01% 11.579us 1 11.579us 11.579us 11.579us cuDeviceGetName
0.01% 6.6170us 1 6.6170us 6.6170us 6.6170us cudaOccupancyMaxActiveBlocksPerMultiprocessorWithFlags
0.00% 4.9970us 1 4.9970us 4.9970us 4.9970us cuDeviceGetPCIBusId
0.00% 4.0370us 1 4.0370us 4.0370us 4.0370us cuDeviceTotalMem
0.00% 2.8620us 4 715ns 388ns 1.4450us cudaDeviceGetAttribute
0.00% 2.7080us 2 1.3540us 188ns 2.5200us cuDeviceGet
0.00% 1.5230us 1 1.5230us 1.5230us 1.5230us cudaGetDevice
0.00% 1.4010us 3 467ns 215ns 937ns cuDeviceGetCount
0.00% 526ns 1 526ns 526ns 526ns cuModuleGetLoadingMode
0.00% 249ns 1 249ns 249ns 249ns cuDeviceGetUuid

Problèmes Rencontrés

1. **Non-coalescence des accès à la mémoire globale :** Les accès des threads à la mémoire globale ne sont pas alignés, ce qui entraîne des transactions inefficaces. Les threads du warp accèdent à des adresses de mémoire globale dispersées, le GPU effectue plusieurs transactions au lieu d'une seule, augmentant la latence et diminuant le débit mémoire.

2. **Conflits dans la mémoire partagée** : Les threads d'un bloc peuvent accéder simultanément aux mêmes banques de mémoire partagée, entraînant des conflits. La mémoire partagée est segmentée en banques, et si plusieurs threads tentent d'accéder à la même banque dans un cycle d'horloge, cela crée un conflit de banque, augmentant la latence.

3. **Problèmes d'alignement des threads et blocs** : Les dimensions des blocs ou des grilles sont mal choisies, entraînant une mauvaise occupation des unités de calcul du GPU. Si la taille des blocs de threads n'est pas un multiple de la taille du warp (32 threads), ou si le nombre total de threads ne correspond pas à la taille des données, certains threads restent inactifs, gaspillant les ressources.

Pour résoudre ces problèmes nous avons été amené à faire plusieurs initiatives :

1. **Non-coalescence des accès à la mémoire globale :** Nous avons dû réorganiser les données afin de garantir des accès continus. En structurant les données de manière à ce que les threads d'un warp accèdent à des adresses alignées, nous avons eu la possibilité de permettre au GPU d'effectuer des transactions groupées, ce qui a réduit la latence et augmenté le débit.

2. **Conflits dans la mémoire partagée** : Nous avons dû utiliser un rembourrage (padding) pour réorganiser les données. En insérant des espaces supplémentaires dans les structures de données, nous avons eu la possibilité de répartir les accès entre différentes banques de mémoire partagée, éliminant ainsi les conflits et améliorant les performances.

3. **Problèmes d'alignement des threads et blocs** : Nous avons dû configurer les dimensions des blocs et des grilles en fonction de la taille des données et des ressources GPU. En utilisant des blocs dont la taille est un multiple de 32, nous avons obtenu l’assurance d’une occupation optimale des unités de calcul. Par ailleurs, en ajustant le nombre total de threads à la taille des données, nous avons eu la possibilité d'exploiter pleinement les capacités du GPU.

Conclusion

Ce laboratoire a permis d'explorer en profondeur les différentes stratégies d'optimisation de la multiplication matricielle en utilisant CUDA, mettant en lumière l'importance d'une gestion efficace de la mémoire dans le calcul parallèle. En progressant à travers les tâches, nous avons étudié les bases de l'exécution sur GPU, puis introduit des optimisations progressives telles que l'utilisation de la mémoire partagée, le rembourrage pour éviter les conflits de banques, et le déroulement de boucles pour améliorer l'efficacité du calcul. Ces techniques ont montré comment minimiser la latence, maximiser l'utilisation de la bande passante, et équilibrer la charge de calcul pour exploiter pleinement les ressources matérielles disponibles.

Grâce au profilage et à l'analyse des performances, nous avons pu identifier les goulots d'étranglement et ajuster les configurations pour améliorer les résultats. L'approche itérative et l'intégration de techniques avancées, comme l'utilisation dynamique des ressources du GPU, ont démontré l'impact significatif de l'optimisation sur la réduction du temps d'exécution.

Ce laboratoire a non seulement renforcé notre compréhension des concepts théoriques de la hiérarchie de la mémoire CUDA, mais a également fourni une expérience pratique précieuse pour résoudre des problèmes de calcul intensif. Les compétences acquises ici sont directement applicables à une large gamme de domaines nécessitant des calculs parallèles, tels que l'apprentissage automatique, la simulation scientifique et le traitement d'images, ouvrant la voie à des solutions plus efficaces et évolutives.

Distribution des tâches

Nom	Tâche assigné
Aziz Tazrout	- Travaille sur le code pour Tâche 1 - Travailler sur la présentation - Contribuer dans le rapport : Introduction, Objectifs, Analyse du problème, Conception de la solution, Équipements & Composants Utilisés, Évaluation de l'approche algorithmique, Conclusion
Lina Bel Bijou	- Travailler sur le code pour Tâche 2. - Contribuer dans le rapport : Conception de la solution, Résultats et Validation et Problèmes Rencontrés.
Gbegbe Decaho Jacques	- Travailler sur le code pour Tâche 3 et 4 . - Contribuer dans le rapport : Conception de la solution, Évaluation de l'implémentation, Résultats et Validation et Problèmes Rencontrés.
Yann Kouadio	- Contribuer dans le code pour Tâche 5 - Contribuer dans le rapport : Table des matières, Référence, Évaluation de l'implémentation, et Problèmes Rencontrés.
Jean Alexandre Elloh	- Contribuer dans le code pour Tâche 6. - Contribuer dans le rapport : Introduction, Objectifs, Évaluation de l'implémentation, et Problèmes rencontrés. -Créer le répertoire Github

Table 1: Distribution de tâche

Références

[1]. M. Ibrahim, “Lab3_fr” Nov, 2024. [Online]. Available: https://uottawa.brightspace.com/d2l/le/content/456941/viewContent/6391029/View [Accessed Nov. 10, 2024].

[2]. NVIDIA Corporation, "CUDA C Programming Guide," Version 11.8, May 2023. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/. [Accessed Oct. 25, 2024].

Appendix

- Vous prouver trouver le code pour ce lab sur le lien suivant :
  tache 1 et 2 :
  https://colab.research.google.com/drive/1VIE1dFdU7fD87S_lVuD30OdVjddmdYXQ?usp=sharing

- tache 3 et 4 :
  https://colab.research.google.com/drive/1SJM3hX6rHpY7lPe3IRROVoI9BDU6Xqhg?usp=sharing

- tache 5 et 6 :
  https://colab.research.google.com/drive/1tyxrxepwuu8QMbwYXW4Iy1QPQf4IjRCh?usp=sharing

- Vous pouvez trouver les vidéos pour la présentation et la démonstration, et tout autre document sur le dossier google drive suivant :
  https://drive.google.com/drive/folders/1pODJ-Fw5DD74rEZzlGldRRxXbqH5lPu6?usp=sharing