RAPPORT DE LABORATOIRE IV – 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: 3 Décembre 2024

Table des matières

Table des figures

Table des tables

LAB 4 :
Optimisation des kernels CUDA pour une utilisation efficace de la mémoire partagée

Introduction

Ce laboratoire explore les techniques avancées d'optimisation des performances des programmes CUDA grâce à une utilisation efficace de la mémoire partagée du GPU. L'objectif est de démontrer comment minimiser les conflits de banque, améliorer l'accès à la mémoire globale et exploiter des concepts tels que le padding de mémoire et les instructions de shuffle de warp pour optimiser les opérations parallèles. En mettant l'accent sur la hiérarchie de mémoire de CUDA, ce projet vise à fournir une compréhension approfondie des stratégies nécessaires pour maximiser l'efficacité des calculs parallèles sur GPU.

Objectifs

- Comprendre et implémenter les opérations de mémoire partagée dans les kernels CUDA.
- Optimiser les performances des kernels en réduisant les conflits de banque et les accès à la mémoire globale.
- Analyser les effets du padding et des instructions de shuffle de warp dans les réductions parallèles et les opérations matricielles.

Analyse du problème

Tâche 1 :

Le problème posé par cette tâche est d'optimiser la transposition d'une matrice sur un GPU en utilisant CUDA pour exploiter efficacement la mémoire partagée dans le but de réduire les latences d'accès et éviter les conflits de banque. Pour ce faire, nous devons implémenter un kernel CUDA permettant la transposition de matrice en utilisant la mémoire partagée. Ensuite, ajouter a notre implémentation du padding mémoire afin de résoudre les conflits de banques éventuels et enfin, à l'aide de l'outil nvprof, étudier les résultats de simulations avec et sans padding mémoire pour faire une comparaison des performances des deux implémentations.

Tâche 2 :

Le problème posé dans cette tâche est de concevoir et d'optimiser un kernel CUDA pour effectuer une réduction parallèle d'un grand tableau de données, en exploitant efficacement la mémoire partagée et les instructions intra-warp pour minimiser le temps d'exécution. La réduction parallèle est une opération clé dans de nombreuses applications, et son implémentation naïve souffre souvent de goulots d'étranglement liés aux conflits de mémoire et à la synchronisation excessive entre threads.

Pour répondre à ce problème, il est nécessaire :

- D'implémenter un kernel CUDA optimisé en utilisant la mémoire partagée pour réduire les accès redondants à la mémoire globale.
- D'intégrer les instructions **__shfl_down_sync** pour tirer parti de la communication intra-warp, réduisant ainsi le besoin de synchronisation explicite.
- De comparer les performances du kernel optimisé avec des approches moins performantes, en mesurant le temps d'exécution global et le coût des transferts mémoire à l'aide de l'outil nvprof.
- De s'assurer que les résultats du kernel sont précis et cohérents avec l'opération mathématique attendue.

Cette tâche met l'accent sur la minimisation des temps d'accès mémoire et sur une utilisation judicieuse des ressources GPU pour accélérer la réduction.

Conception de la solution

Tâche 1 : Transposition de Matrice et Réduction Parallèle

La première étape de cette tâche consiste à développer des noyaux CUDA de base pour effectuer la transposition de matrice et la réduction parallèle, établissant ainsi une base de référence en termes de performances et de comportement mémoire.

Transposition de Matrice

Un noyau CUDA basique sera implémenté pour réaliser la transposition d'une matrice. Chaque thread sera responsable de déplacer un élément spécifique de la matrice source vers sa position transposée dans la matrice destination. Les matrices d'entrée et de sortie seront allouées dans la mémoire globale du GPU en utilisant cudaMalloc et initialisées avec des valeurs générées dynamiquement. Les index de threads seront soigneusement calculés pour garantir que chaque thread accède à une position unique dans la matrice source. Cette implémentation permettra de mesurer les performances de base de la transposition, en établissant une référence pour les optimisations ultérieures.

Réduction Parallèle

Un noyau CUDA de base sera également conçu pour effectuer une réduction parallèle. Chaque thread additionnera deux éléments adjacents du tableau d'entrée, stockant le résultat dans un tableau intermédiaire. Ce processus sera répété de manière itérative, réduisant de moitié le nombre d'éléments à chaque étape, jusqu'à ce qu'un seul élément reste. Les index des threads seront calculés de manière à garantir une distribution équitable des tâches, évitant ainsi les conflits entre threads. Cette implémentation servira à identifier les inefficacités potentielles et à fournir une base de comparaison pour évaluer les améliorations futures.

Tâche 2 : Optimisation avec Mémoire Partagée et Instructions Warp

Cette tâche combine des techniques avancées d'optimisation pour améliorer les performances de la transposition de matrice et de la réduction parallèle. L'accent sera mis sur l'utilisation efficace de la mémoire partagée et des instructions intra-warp pour minimiser les accès à la mémoire globale, réduire les conflits de banque et améliorer l'efficacité globale.

Optimisation pour la Transposition de Matrice

Un schéma de tuilage sera mis en place, où chaque bloc de threads traite une sous-matrice. Les éléments de cette sous-matrice seront chargés dans la mémoire partagée pour réduire les accès coûteux à la mémoire globale. Du padding sera introduit pour ajuster la disposition des éléments dans la mémoire partagée et résoudre les conflits de banque potentiels. En ajustant les indices de threads ainsi que les dimensions des blocs et des grilles, nous viserons à maximiser l'utilisation des threads tout en maintenant des schémas d'accès contigus et optimaux. L'impact des optimisations sera évalué à l'aide de l'outil nvprof pour mesurer les latences et identifier les gains en performance.

Optimisation pour la Réduction Parallèle

Pour la réduction parallèle, les performances seront améliorées en combinant l'utilisation de la mémoire partagée et des instructions intra-warp comme __shfl_down_sync. Les threads d'un même bloc chargeront leurs données dans la mémoire partagée, réduisant ainsi les accès redondants à la mémoire globale. Les opérations intra-warp permettront de réduire davantage la synchronisation explicite entre threads, accélérant les étapes de réduction. La configuration des blocs et des grilles sera optimisée pour équilibrer la charge et maximiser l'utilisation des unités de calcul.

Évaluation des Performances

Des analyses détaillées avec nvprof seront menées pour comparer les versions optimisées et non optimisées. L'objectif est de quantifier les gains obtenus en termes de réduction de latence, d'élimination des conflits de mémoire et d'amélioration de l'utilisation de la bande passante. Ces optimisations garantiront une efficacité accrue des opérations sur GPU, particulièrement pour des jeux de données de grande taille.

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 : Transposition de Matrice

Ce code implémente une transposition de matrice en exploitant le parallélisme offert par CUDA. La taille de la matrice source (N×N) est prédéfinie, et chaque élément de la matrice transposée est calculé indépendamment par un thread. Les indices des blocs et des threads sont utilisés pour déterminer l’élément spécifique de la matrice source à déplacer dans la matrice destination. Les matrices sont allouées et initialisées sur l’hôte (CPU), puis transférées vers la mémoire globale du GPU à l’aide de cudaMemcpy. Une fois les calculs effectués sur le GPU, les résultats sont copiés vers l’hôte pour vérification et affichage. Cette implémentation de base, bien que fonctionnelle, souffre des inconvénients liés aux accès fréquents à la mémoire globale, ce qui limite ses performances. Elle sert néanmoins de référence pour évaluer les améliorations apportées par l'utilisation de la mémoire partagée et du padding.

Tâche 2 : Réduction Parallèle

Ce code implémente une version optimisée de la réduction parallèle en exploitant la mémoire partagée et les instructions intra-warp telles que __shfl_down_sync. Chaque bloc de threads traite un sous-ensemble des éléments du tableau d’entrée, en préchargeant ces éléments dans un tableau en mémoire partagée pour limiter les accès à la mémoire globale. Les threads collaborent ensuite pour effectuer les étapes successives de réduction, réduisant de moitié le nombre d’éléments à chaque itération. Les threads au sein des warps utilisent des opérations intra-warp pour accélérer les calculs sans nécessiter de synchronisation explicite, jusqu’à ce qu’un seul élément reste par bloc. Les résultats partiels sont écrits en mémoire globale et combinés ultérieurement sur l’hôte pour obtenir la valeur finale. Cette approche améliore significativement les performances par rapport à une implémentation naïve, en réduisant la latence et en minimisant les conflits d'accès mémoire.

Évaluation de l'implémentation

Le laboratoire a démontré des améliorations significatives des performances à travers plusieurs tâches utilisant CUDA, notamment la transposition de matrice, la réduction parallèle et les optimisations liées à la mémoire partagée. Chaque implémentation a été évaluée à l'aide de l'outil de profilage "nvprof", permettant d'identifier les goulots d'étranglement et d'évaluer l'impact des optimisations.

1. **Transposition de Matrice**

- **Efficacité** : L'utilisation de la mémoire partagée avec padding a permis d'éliminer les conflits de banque, ce qui a amélioré le temps d'exécution global par rapport à une implémentation naïve. La réduction des conflits mémoire a été validée par le profilage, où le temps consacré au kernel optimisé était marginal comparé aux copies mémoire. L'optimisation a été particulièrement efficace pour des matrices de grande taille, montrant une réduction significative des latences d'accès à la mémoire.

- **Lisibilité :** L'ajout de padding a augmenté la complexité du code, en particulier dans la gestion des dimensions et des indices des blocs. Cependant, des commentaires explicites ont facilité la compréhension du rôle de la mémoire partagée et des ajustements nécessaires pour éviter les conflits.

- **Maintenabilité :** L'approche modulaire de la gestion des dimensions de blocs et de l'utilisation conditionnelle du padding rend le code adaptable à différentes tailles de matrices. Cependant, la spécialisation pour CUDA limite son application directe à d'autres plateformes.

2. **Réduction Parallèle**

- **Efficacité** : L'utilisation de la mémoire partagée combinée aux instructions intra-warp __shfl_down_sync a réduit le besoin de synchronisation explicite et améliorer le parallélisme. Les résultats du profilage ont montré que le kernel optimisé avait un temps d'exécution minimal (36 µs), bien inférieur aux temps des transferts mémoire. L'algorithme a montré une excellente scalabilité pour de grandes tailles de données.

- **Lisibilité :** Les optimisations, comme l'unrolling des boucles et l'utilisation des opérations intra-warp, rendent le code plus difficile à lire sans une compréhension préalable des concepts CUDA. Les commentaires expliquant chaque étape, notamment la réduction dans les warps, ont été essentiels pour clarifier l'implémentation.

- **Maintenabilité :** Le code est bien structuré et permet des ajustements simples, comme la modification de la taille des blocs. Toutefois, les optimisations spécifiques au matériel GPU, comme l'utilisation des warps, limitent sa

portabilité vers d'autres architectures parallèles.

3. Profiling et Gestion des Données

- **Efficacité :** Dans toutes les tâches, les coûts des transferts de données entre l'hôte et le GPU (HtoD et DtoH) ont été identifiés comme des goulots d'étranglement, représentant jusqu'à 95% du temps total pour certaines exécutions. Cela met en évidence la nécessité de minimiser ces transferts, par exemple en regroupant les calculs ou en effectuant plus d'opérations directement sur le GPU.

- **Lisibilité :** Les outils comme nvprof ont fourni des rapports clairs sur les temps d'exécution, permettant d'isoler les sections du code à optimiser. Ces résultats ont été faciles à intégrer dans l'analyse pour justifier les décisions de conception.

- **Maintenabilité :** L'analyse des performances et la modularité du code permettent d'ajouter facilement des optimisations supplémentaires ou de tester des alternatives, comme l'utilisation d'autres techniques de gestion mémoire.

Résultats et Validation

Tâche 1 :

```
[6] !nvprof ./Tache1

==8020== NVPROF is profiling process 8020, command: ./Tache1
==8020== Profiling application: ./Tache1
==8020== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   66.31%  1.6614ms         1  1.6614ms  1.6614ms  1.6614ms  [CUDA memcpy DtoH]
                   31.35%  785.44us         1  785.44us  785.44us  785.44us  [CUDA memcpy HtoD]
                    2.34%  58.688us         1  58.688us  58.688us  58.688us  transposeWithSharedMemory(float*, float*, int, int
      API calls:   97.53%  189.43ms         2  94.713ms  74.266us  189.35ms  cudaMalloc
                    2.09%  4.0564ms         2  2.0282ms  950.42us  3.1060ms  cudaMemcpy
                    0.16%  307.07us         2  153.54us  103.80us  203.28us  cudaFree
                    0.11%  204.58us         1  204.58us  204.58us  204.58us  cudaLaunchKernel
                    0.10%  195.52us       114  1.7150us     151ns  71.307us  cuDeviceGetAttribute
                    0.01%  13.112us         1  13.112us  13.112us  13.112us  cuDeviceGetName
                    0.00%  7.2880us         1  7.2880us  7.2880us  7.2880us  cuDeviceGetPCIBusId
                    0.00%  6.1730us         1  6.1730us  6.1730us  6.1730us  cuDeviceTotalMem
                    0.00%  1.7010us         3     567ns     256ns  1.0660us  cuDeviceGetCount
                    0.00%  1.1450us         2     572ns     303ns     842ns  cuDeviceGet
                    0.00%     615ns         1     615ns     615ns     615ns  cuModuleGetLoadingMode
                    0.00%     250ns         1     250ns     250ns     250ns  cuDeviceGetUuid
```

Tâche 2 :

```
[10] !./Tache2

Résultat final : 524288

[11] !nvprof ./Tache2

==9713== NVPROF is profiling process 9713, command: ./Tache2
Résultat final : 524288
==9713== Profiling application: ./Tache2
==9713== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   95.20%  765.76us         1  765.76us  765.76us  765.76us  [CUDA memcpy HtoD]
                    4.48%  36.064us         1  36.064us  36.064us  36.064us  reduceOptimized(float const *, float*, int)
                    0.31%  2.5280us         1  2.5280us  2.5280us  2.5280us  [CUDA memcpy DtoH]
      API calls:   99.11%  191.53ms         2  95.764ms  77.585us  191.45ms  cudaMalloc
                    0.49%  952.33us         2  476.17us  27.288us  925.04us  cudaMemcpy
                    0.17%  335.42us         2  167.71us  134.59us  200.83us  cudaFree
                    0.11%  208.05us         1  208.05us  208.05us  208.05us  cudaLaunchKernel
                    0.09%  167.80us       114  1.4710us     137ns  83.408us  cuDeviceGetAttribute
                    0.02%  37.881us         1  37.881us  37.881us  37.881us  cudaDeviceSynchronize
                    0.01%  11.438us         1  11.438us  11.438us  11.438us  cuDeviceGetName
                    0.00%  5.1060us         1  5.1060us  5.1060us  5.1060us  cuDeviceGetPCIBusId
                    0.00%  4.3040us         1  4.3040us  4.3040us  4.3040us  cuDeviceTotalMem
                    0.00%  2.6100us         2  1.3050us     183ns  2.4270us  cuDeviceGet
                    0.00%  1.4420us         3     480ns     241ns     929ns  cuDeviceGetCount
                    0.00%     679ns         1     679ns     679ns     679ns  cuModuleGetLoadingMode
                    0.00%     358ns         1     358ns     358ns     358ns  cuDeviceGetUuid
```

Problèmes Rencontrés

1. **Dépendance au Matériel**

   Les optimisations avancées, comme l'utilisation des instructions intra-warp __shfl_down_sync, sont fortement dépendantes des architectures CUDA modernes. Cela limite la compatibilité avec des GPU plus anciens ou d'autres plateformes comme OpenCL ou CPU multicœurs.

   Solution : Fournir une implémentation alternative pour les architectures plus anciennes, utilisant uniquement des synchronisations explicites (__syncthreads) et des accès classiques à la mémoire partagée. De plus, intégrer des options de compilation conditionnelles basées sur la version de CUDA pour maintenir la compatibilité.

2. **Temps de Transfert Mémoire**
   Les transferts entre la mémoire hôte et le GPU représentaient une grande partie du temps total d'exécution, masquant les gains obtenus avec l'optimisation des noyaux CUDA.

   Solution : Réduire les transferts mémoire en regroupant les opérations de calcul GPU pour traiter les données en une seule fois. Exploiter la mémoire unifiée CUDA pour minimiser les copies explicites entre l'hôte et le GPU. Enfin, optimiser les transferts avec des techniques de pipeline pour permettre l'overlap entre calcul et transfert.

3. **Conflits de Banque en Mémoire Partagée**
   Les conflits de banque dans la mémoire partagée ralentissaient les performances, notamment lors de la transposition de matrice. Ce problème survient lorsque plusieurs threads accèdent simultanément à des adresses qui partagent la même banque mémoire.

   Solution : Introduire du padding pour réorganiser la disposition des données dans la mémoire partagée, garantissant que les accès des threads sont répartis sur des banques distinctes. Ajuster dynamiquement la taille du padding en fonction des dimensions du bloc et des spécifications matérielles pour une efficacité maximale.

4. **Optimisation des Accès à la Mémoire Globale**
   Les schémas d'accès non coalescés à la mémoire globale réduisaient les performances, en particulier lors de la réduction parallèle où les accès évoluent à chaque itération.

   Solution : Réorganiser les indices et l'agencement des threads pour aligner les accès mémoire sur les limites de 128 ou 256 octets, permettant des transactions

coalescées. Pour les cas où la coalescence n’est pas directement possible, précharger les données dans la mémoire partagée pour optimiser les accès en mémoire globale.

Conclusion

Au terme de ce laboratoire, nous avons pu approfondir nos connaissances sur les techniques avancées d'optimisation des performances des programmes CUDA et connaître leur importance. En effet, en utilisant les méthodes de padding mémoire et de shuffle de warp dans nos implémentations, nous avons pu obtenir des résultats satisfaisant avec la réduction des conflits de banques mais aussi au niveau de l’optimisation de l’acces a la mémoire globale. Ces techniques nous ont permis avec le profilage de mettre en évidence l'efficacité de la mémoire partagée lorsqu’elle est utilisée avec un alignement précis des données et des techniques adéquates pour l'utiliser à son plein potentiel.

Nous pouvons ainsi en conclure que les techniques d'optimisation des performances des programmes CUDA sont bels et bien efficaces dans le but d'améliorer les performances des applications GPU.

Distribution des tâches

Nom	Tâche assigné
Aziz Tazrout	- 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	- Contribuer dans le rapport : Conception de la solution, Résultats et Validation et Problèmes Rencontrés.
Gbegbe Decaho Jacques	- Travailler sur le code pour Tâche 1 . - 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 1 - Contribuer dans le rapport : Table des matières, Référence, Évaluation de l'implémentation, et Problèmes Rencontrés.
Jean Alexandre Elloh	- Contribuer dans le code pour Tâche 2. - 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 :
  tâche 1 et 2 : lab4_CEG_4536_tache_1_et_2

- Vous pouvez trouver les vidéos pour la présentation et la démonstration, et tout autre document sur le dossier google drive suivant : LAB 4 - CEG 4536