Université d'Ottawa
Faculté de génie

École de science informatique
et de génie électrique

uOttawa
L'Université canadienne
Canada's university

University of Ottawa
Faculty of Engineering

School of Electrical Engineering
and Computer Science

CEG 4536 Architecture des ordinateurs III Automne 2024

Lab4 : Optimisation des kernels CUDA pour une utilisation efficace de la mémoire partagée

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: 25 Novembre, 2 Décembre 2024

Date de la soumission: 3 Décembre 2024

Table of Contents

Table of Figures

Table of Tables

Introduction

Dans le cadre de ce laboratoire, on vise à maîtriser l'utilisation de la mémoire partagée dans les kernels CUDA pour maximiser les performances des programmes GPU. Nous mettrons en œuvre des stratégies d'optimisation, telles que la réduction des conflits de banque, l'application de padding mémoire, et l'exploitation des instructions de shuffle de warp. En s'appuyant sur les concepts clés de la hiérarchie de mémoire CUDA, nous analyserons les gains de performance obtenus par rapport à des implémentations non optimisées grâce à des outils de profilage comme "nvprof".

Objectif

Les objectifs de ce laboratoire sont :

- **Maîtriser et mettre en œuvre les opérations de mémoire partagée** dans les kernels CUDA pour une gestion efficace des données.
- **Améliorer les performances des kernels** en minimisant les conflits de banque et en réduisant les dépendances à la mémoire globale.
- **Étudier l'impact du padding mémoire et des instructions de shuffle de warp** sur les performances des réductions parallèles et des calculs matriciels, tout en optimisant leur implémentation.

Analyse du problème

Ce projet met en évidence les défis liés à l'optimisation de la mémoire partagée sur GPU, où les performances sont souvent limitées par les conflits de banque et les accès inefficaces à la mémoire globale. La transposition de matrice sans gestion adéquate du schéma d'accès peut entraîner des accès concurrents à la même banque mémoire, ralentissant l'exécution. L'ajout de padding permet de contourner ces conflits en dés-alignant les accès. Par ailleurs, les réductions parallèles profitent des instructions de shuffle pour partager les données entre threads d'un même warp sans recourir à la mémoire partagée, réduisant ainsi la latence. L'objectif est d'analyser l'impact de ces optimisations sur les performances en comparant des implémentations avec et sans optimisation, en utilisant des outils de profilage tels que nvprof, afin d'illustrer l'importance du choix des stratégies d'accès mémoire et d'alignement pour exploiter pleinement la hiérarchie de mémoire CUDA.

Conception de la solution

La solution conçue repose sur l'optimisation des kernels CUDA pour la transposition de matrices et la réduction parallèle en exploitant la mémoire partagée et les techniques de gestion d'accès

efficaces. Le kernel de transposition sans padding utilise un tableau partagé 32×32 pour stocker des sous-blocs de la matrice, mais peut provoquer des conflits de banque. Pour remédier à cela, un kernel optimisé introduit un padding d'une colonne supplémentaire, brisant l'alignement des accès concurrents. La réduction parallèle utilise un schéma hiérarchique : les données sont d'abord agrégées dans la mémoire partagée, puis finalisées dans un warp grâce à l'instruction __shfl_down_sync, permettant une réduction efficace sans recours à la mémoire globale. Des timers CUDA mesurent et comparent les temps d'exécution des différentes implémentations pour évaluer les gains en performances, tandis que des vérifications CPU garantissent la validité des résultats GPU.

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

La procédure de résolution suit une approche méthodique pour optimiser la transposition de matrices et la réduction parallèle sur GPU. Dans un premier temps, la matrice est décomposée en blocs de taille fixe 32×32, chaque bloc étant traité indépendamment dans un kernel. Le kernel de transposition sans padding charge les éléments dans la mémoire partagée, synchronise les threads, puis écrit les éléments transposés dans la mémoire globale. Pour éviter les conflits de banque, un deuxième kernel introduit un padding d'une colonne, garantissant des accès désalignés en mémoire partagée. En parallèle, l'algorithme de réduction divise les données en segments traités par chaque bloc de threads. Chaque thread charge un élément dans la mémoire partagée, et la réduction se fait itérativement en combinant les éléments voisins selon un schéma hiérarchique. Une fois la taille des segments réduite à un warp, l'algorithme utilise des instructions de shuffle pour finaliser la réduction, éliminant la synchronisation intra-warp. Les résultats partiels sont ensuite agrégés dans un second passage CPU. Des mesures de performance à l'aide de cudaEvent permettent d'évaluer l'impact des optimisations.

Évaluation d'implémentation

L'implémentation des kernels optimisés a démontré une amélioration notable en termes de performance et d'efficacité mémoire. La transposition de matrice avec padding a réduit les conflits de banque en mémoire partagée, permettant une meilleure parallélisation des accès. Le kernel sans padding, bien que fonctionnel, a subi des ralentissements dus à des accès conflictuels, ce qui a été confirmé par le profilage. L'utilisation des instructions de shuffle dans la réduction parallèle a minimisé les dépendances entre threads, réduisant le besoin de synchronisation explicite et améliorant ainsi le débit. L'évaluation a également montré que la gestion des blocs et des threads était bien équilibrée, maximisant l'utilisation des ressources GPU. En conclusion, les optimisations mises en œuvre, telles que le padding et les instructions de shuffle, ont permis de respecter les objectifs de performance tout en garantissant l'exactitude des résultats.

Résultats et Validation

```
Temps sans padding : 0.209632 ms
Temps avec padding : 0.094400 ms
Temps de réduction optimisée : 0.032704 ms
ReductionSum of GPU : 1024
Reduction of CPU : 1024
```

Figure 1 : sortie d'exécution

```
==3764== NVPROF is profiling process 3764, command: ./lab4_2
Temps sans padding : 0.341536 ms
Temps avec padding : 0.115264 ms
Temps de réduction optimisée : 0.053728 ms
ReductionSum of GPU : 1024
Reduction of CPU : 1024
==3764== Profiling application: ./lab4_2
==3764== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   48.98%  2.7054ms         3  901.81us  1.6640us  2.0390ms  [CUDA memcpy DtoH]
                   47.38%  2.6170ms         2  1.3085ms  1.4720us  2.6156ms  [CUDA memcpy HtoD]
                    2.44%  134.91us         1  134.91us  134.91us  134.91us  matrixTransposeNoPadding(float*, float*, int, int)
                    1.13%  62.207us         1  62.207us  62.207us  62.207us  matrixTransposeWithPadding(float*, float*, int, int)
                    0.08%  4.2560us         1  4.2560us  4.2560us  4.2560us  reduceOptimized(float const *, float*, int)
      API calls:   95.77%  210.37ms         4  52.593ms  18.760us  210.00ms  cudaMalloc
                    3.53%  7.7486ms         5  1.5497ms  24.711us  3.7897ms  cudaMemcpy
                    0.30%  653.01us         4  163.25us  8.4220us  234.86us  cudaFree
                    0.15%  331.99us         3  110.66us  46.148us  236.00us  cudaLaunchKernel
                    0.10%  217.52us       114  1.9080us     185ns  84.327us  cuDeviceGetAttribute
```

Figure 2 : Nvprof

Les résultats obtenus montrent une amélioration significative des performances lorsque le padding est utilisé pour la transposition de matrice, réduisant les conflits de banque en mémoire partagée. Le kernel optimisé avec padding a permis de diminuer le temps d'exécution par rapport à la version sans padding, confirmant l'efficacité de cette technique. La validation des résultats a été effectuée en comparant les matrices transposées générées par les kernels CUDA avec celles obtenues par une transposition CPU, garantissant leur exactitude. Concernant la réduction parallèle, la somme calculée par le GPU est cohérente avec celle obtenue sur le CPU, validant l'implémentation du kernel. Le profilage a également permis d'identifier l'impact des optimisations sur l'utilisation des ressources GPU, confirmant que l'alignement mémoire et l'utilisation des instructions de shuffle réduisent la latence et améliorent l'efficacité globale du calcul parallèle.

Problèmes Rencontrés

Utilisation de la précision simple (float)

Initialement, le code utilisait des variables en simple précision (float) pour la réduction, ce qui a conduit à des erreurs d'accumulation dues à l'arrondi. Ce problème s'est amplifié lorsque le nombre d'éléments à additionner a augmenté, entraînant des dépassements de valeur (overflow). La solution a été de passer à des variables en double précision (double) pour les calculs, ce qui a considérablement amélioré la précision mais a également augmenté légèrement le temps d'exécution.

Conflits de mémoire et dépassements de tampon

Lors de l'exécution, des dépassements d'index ont été observés, notamment lorsque le nombre total d'éléments (n) n'était pas un multiple de la taille des blocs. Ce problème a nécessité l'ajout d'une vérification des limites dans le kernel pour éviter des lectures ou écritures hors limites, ce qui a introduit un coût de synchronisation supplémentaire.

Difficulté avec les métriques de “nvprof”

Un autre problème a été rencontré lors de l'utilisation de l'outil de profilage “nvprof”. La commande pour extraire les métriques nécessaires n’a pas fonctionné comme prévu, et malgré plusieurs tentatives, des données incohérentes ou manquantes ont été obtenues. Cela a compliqué l’analyse des performances et l’identification des goulets d’étranglement.

Validation des résultats

Bien que la réduction GPU ait montré des résultats prometteurs, des incohérences ont été détectées lors de la validation sur le CPU. Ce problème a nécessité une vérification manuelle des valeurs à plusieurs étapes, augmentant le temps de débogage et la complexité de l'implémentation.

Initialisation incorrecte des données

Pendant les tests, certains cas ont montré que les données d’entrée n’étaient pas correctement initialisées, ce qui a introduit des erreurs imprévisibles. Une attention particulière a été portée à l’initialisation des tableaux d'entrée pour garantir que les valeurs soient définies avant de lancer les kernels.

Conclusion

Ce laboratoire nous a permis d’explorer en profondeur l’optimisation des kernels CUDA pour une utilisation efficace de la mémoire partagée, en mettant en pratique des concepts avancés de programmation GPU. Nous avons d’abord implémenté un kernel de transposition de matrice en utilisant la mémoire partagée et en appliquant une stratégie de padding pour éliminer les conflits de banque, ce qui a permis de réduire considérablement les accès non alignés à la mémoire globale. Ensuite, nous avons conçu des kernels de réduction parallèle optimisés grâce à l’utilisation des instructions de shuffle de warp, offrant des performances accrues pour les calculs massivement parallèles.

Ces implémentations ont été rigoureusement profilées à l’aide de “nvprof”, nous permettant d’analyser en détail les statistiques sur les conflits de banque et les schémas d’accès mémoire. Les comparaisons des performances entre les versions optimisées et non optimisées des kernels ont mis en évidence des gains significatifs, démontrant l’impact crucial des alignements mémoire et des stratégies de gestion efficace de la hiérarchie mémoire CUDA.

En conclusion, ce projet nous a non seulement permis de maîtriser les techniques avancées d'optimisation des kernels CUDA, mais il a également renforcé notre compréhension de l'architecture GPU et de son potentiel dans les calculs intensifs. Les compétences acquises ici sont directement applicables dans des contextes où la performance et l'efficacité des calculs massifs sont primordiales.

Distribution de tâches

Nom	Répartition des tâches
Adam Taktek	Travailler sur le code de la tâche du lab
Maro Abdine	Travailler sur le code de la tâche du lab
Christopher Krayem	Écrit le rapport du lab
Simon Marchildon	Écrit le rapport de lab
Reda Bensalah	Écrit le rapport de lab
Chergui Hakim	Démontrer le code de la tâche du lab

Table 1: Distribution de tâche

Références

Appendice

Onedrive LAB4
GitHub