RAPPORT DE LABORATOIRE II – 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: 04 Novembre 2024

Table des matières

Table des figures

Table des tables

LAB 2 :
Optimisation et Exécution Parallèle avec CUDA C

Introduction

Ce laboratoire porte sur l'optimisation de l'exécution parallèle d'algorithmes sur GPU en utilisant CUDA C, un modèle de programmation clé pour exploiter pleinement la puissance des GPU. L'objectif est d'apprendre et de mettre en pratique des techniques avancées d'optimisation comme la gestion des warps, la réduction de la divergence des threads, l'unrolling des boucles, et l'utilisation efficace de la mémoire partagée.
À travers plusieurs exercices,nous implémenterons, profilerons et optimiserons des algorithmes de réduction parallèle, en s'appuyant sur des outils tels que nvprof pour détecter les goulets d'étranglement et évaluer les performances. Ce laboratoire vise à renforcer la compréhension du modèle d'exécution CUDA et à explorer comment les choix d'optimisation influencent la scalabilité, l'utilisation des ressources et la performance des applications parallèles.

Objectifs

- Acquérir une compréhension approfondie du modèle d'exécution des warps et blocs de threads.
- Réduction des divergences des warps.
- Optimisation des performances par unrolling des boucles et utilisation des templates.
- Exécution imbriquée et parallélisation dynamique.
- Utilisation de nvprof pour profiler et optimiser le code.

Analyse du problème

Tâche 1 :

Dans cette étape, l'objectif est de développer un code initial qui invoque un kernel pour réaliser une réduction parallèle, additionnant les éléments d'un tableau. Après sa création, ce code sera soumis à des tests, et l'outil *nvprof* sera utilisé pour évaluer les warps actifs ainsi que les opérations en mémoire. Cette analyse nous aidera à repérer les problèmes de divergence des warps. Ce premier jet du code de réduction parallèle constitue une base solide pour des optimisations ultérieures dans les prochaines phases.

Tâche 2 :

Après avoir développé une première version du code de réduction et l'avoir analysée avec nvprof, nous avons identifié des moyens d'améliorer le code en réduisant la divergence des warps, grâce aux résultats obtenus dans la première tâche. En effet, la boucle for descend jusqu'à des décalages inférieurs à 32, ce qui entraîne une divergence entre les threads d'un même warp. Pour éviter cela, nous devons interrompre la boucle for à un décalage de 32, puis effectuer manuellement les opérations restantes en utilisant des variables volatiles. Nous allons également appliquer la technique de *loop unrolling* dans le kernel en dupliquant le contenu de la boucle pour réduire le nombre d'itérations. Enfin, nous ajouterons des fonctions template pour permettre l'utilisation de la réduction pour d'autres opérations, comme la multiplication, et pour différents types de variables, rendant ainsi notre kernel de réduction plus générique et modulaire.

Tâche 3 :

Dans cette tâche, il s'agit de mettre en place une réduction parallèle utilisant le parallélisme dynamique et d'explorer l'impact de cette approche par rapport à une exécution sans parallélisme dynamique. L'objectif est d'évaluer la scalabilité et l'efficacité du programme lorsque davantage de parallélisme est exploité.La parallélisation dynamique permet aux threads sur GPU de lancer d'autres threads pendant l'exécution, ce qui expose un parallélisme supplémentaire. Cela est particulièrement utile pour les tâches complexes nécessitant des calculs imbriqués, car cette méthode permet de diviser les calculs en sous-parties parallèles. Toutefois, pour garantir l'efficacité, il est important de gérer les ressources du GPU afin d'éviter une surcharge de threads, qui pourrait ralentir le programme.

Tâche 4 :

Cette étape met l'accent sur le profilage et l'optimisation basée sur les profils pour repérer et supprimer les goulets d'étranglement qui freinent les performances du programme. Nous utiliserons l'outil nvprof de NVIDIA pour analyser en profondeur les performances, identifier les sources de latence, et observer l'occupation des warps ainsi que l'utilisation des ressources. L'objectif est de maximiser l'efficacité du GPU en améliorant l'utilisation des ressources et en optimisant la gestion de la latence. Pour atteindre cet objectif, nous appliquerons des techniques de masquage de la latence et exploiteront le partitionnement des ressources afin d'assurer une occupation élevée des warps, tout en réduisant les périodes d'inactivité des threads.

Conception de la solution

Tâche 1 :

Dans cette première tâche, nous implémentons une addition parallèle des éléments d'un tableau en utilisant CUDA, en adoptant une approche de réduction parallèle pour exploiter efficacement le calcul massivement parallèle du GPU. Chaque bloc de threads calcule une somme partielle en stockant les valeurs dans la mémoire partagée, qui est bien plus rapide d'accès que la mémoire globale, permettant ainsi de minimiser la latence due aux transferts de données. La réduction est réalisée en divisant le tableau par deux à chaque itération, les threads additionnant leurs valeurs par paires. Cette technique réduit progressivement le nombre d'éléments à additionner jusqu'à obtenir une seule valeur par bloc.

Pour éviter la divergence des threads, nous structurons le code de manière à ce que seuls les threads nécessaires effectuent les opérations de réduction, en se basant sur leurs indices (tid), limitant ainsi les exécutions conditionnelles et maximisant l'occupation des warps. Une fois le kernel parallelAdd exécuté, nous transférons les résultats partiels du GPU vers la mémoire hôte pour effectuer une réduction finale sur le CPU, obtenant ainsi la somme totale des éléments.

Afin de mesurer les performances et d'identifier les éventuels goulets d'étranglement, nous profilons le code avec l'outil nvprof. Cet outil nous permet d'analyser l'occupation des warps, de détecter les problèmes liés à la gestion de la mémoire et de quantifier l'impact des optimisations sur la performance. En suivant cette approche, nous cherchons à maximiser l'efficacité de notre algorithme de réduction, en tirant pleinement parti de la parallélisation offerte par le GPU pour obtenir un code rapide et optimisé.

Tâche 2 :

Dans cette deuxième tâche, nous avons significativement amélioré l'addition parallèle en exploitant des techniques avancées de CUDA, telles que les templates et le *loop unrolling*, afin d'optimiser l'algorithme de réduction sur GPU. Le kernel *parallelReduceUnrolled* a été conçu pour être à la fois flexible et performant : l'utilisation de templates permet de rendre le kernel générique, capable de réaliser diverses opérations (comme l'addition et la multiplication) en passant simplement un opérateur spécifique, sans avoir à modifier le code central. Cette modularité rend le kernel réutilisable et adaptable à différents types de calculs parallèles.

Pour maximiser les performances, nous avons appliqué un *loop unrolling* avec un facteur de 4, ce qui permet de traiter plusieurs éléments par itération de boucle. En réduisant le nombre d'itérations, le *loop unrolling* diminue la surcharge des branchements et permet d'optimiser l'utilisation de la mémoire partagée. Cela rend les calculs internes plus rapides et fluides, en particulier dans les environnements GPU où la gestion des ressources est cruciale pour obtenir de hautes performances.

La dernière étape de la réduction utilise une réduction au niveau du warp, optimisée par des variables volatiles pour minimiser la latence des accès mémoire entre threads. En procédant ainsi, les 32 premiers threads d'un bloc sont utilisés efficacement, limitant les besoins en synchronisation et augmentant ainsi le débit global des calculs.

Pour évaluer l'impact de ces optimisations, nous avons chronométré le kernel en utilisant *cudaEvent_t*. Ce profilage précis nous a permis de quantifier les gains de performance et de valider l'efficacité des améliorations. Ces mesures montrent que l'optimisation par *loop unrolling* et la réduction par warp permettent d'exploiter pleinement les capacités du GPU, en réduisant la latence et en maximisant l'occupation des ressources. En adoptant cette approche, nous avons obtenu un code non seulement plus rapide mais aussi plus adaptable, capable de traiter efficacement des opérations de réduction parallèles à grande échelle, ouvrant ainsi des possibilités pour des applications nécessitant des calculs intensifs.

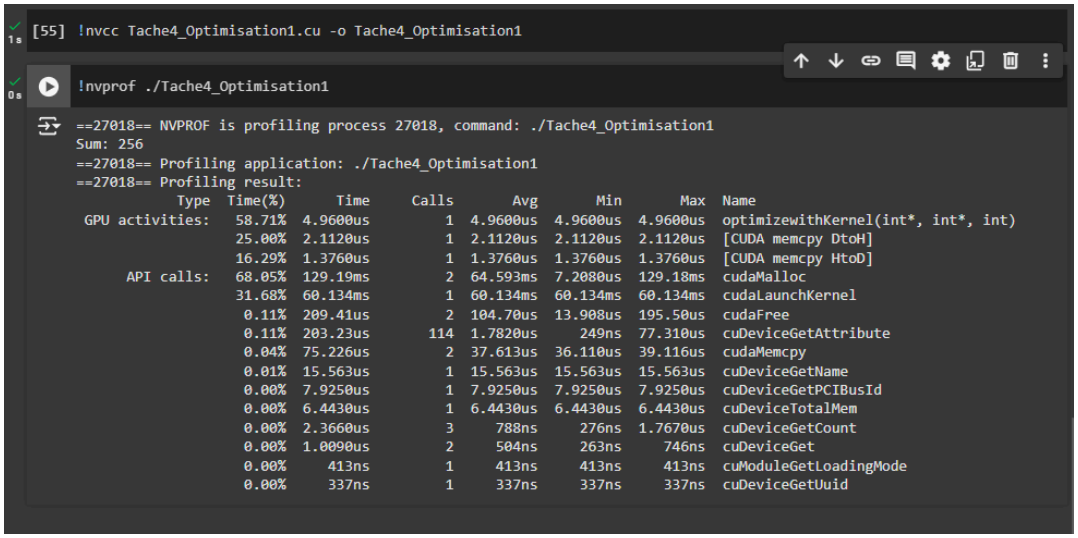
Tâche 3 :

Dans cette tâche, l'objectif est d'intégrer un appel imbriqué à un nouveau kernel pour effectuer la réduction finale des valeurs sauvegardées par chaque bloc, ce qui représente une extension par rapport aux étapes précédentes. Dans les tâches antérieures, chaque bloc calculait uniquement une réduction locale, en enregistrant un résultat partiel sans compléter la réduction totale. L'ajout de ce kernel imbriqué permet de combiner ces valeurs partielles pour obtenir le résultat final de la réduction, améliorant ainsi la cohérence et l'efficacité de l'algorithme. Cette approche utilise une exécution en plusieurs étapes, où chaque appel kernel contribue à la réduction successive des données, exploitant davantage la parallélisation sur le GPU.

Tâche 4 : Dans cette quatrième tâche, il était question d'optimisations par profiling et part optimisation.

Étape 1 : Profiling initial pour identifier les On goulets d'étranglement. On commence par exécuter une analyse de base avec nvprof pour repérer les goulets d'étranglement dans le code CUDA.

Réduire les appels Cudamalloc



```
[55] !nvcc Tache4_Optimisation1.cu -o Tache4_Optimisation1

!nvprof ./Tache4_Optimisation1

==27018== NVPROF is profiling process 27018, command: ./Tache4_Optimisation1
Sum: 256
==27018== Profiling application: ./Tache4_Optimisation1
==27018== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   58.71%  4.9600us         1  4.9600us  4.9600us  4.9600us  optimizewithKernel(int*, int*, int)
                   25.00%  2.1120us         1  2.1120us  2.1120us  2.1120us  [CUDA memcpy DtoH]
                   16.29%  1.3760us         1  1.3760us  1.3760us  1.3760us  [CUDA memcpy HtoD]
      API calls:   68.05%  129.19ms         2  64.593ms  7.2080us  129.18ms  cudaMalloc
                   31.68%  60.134ms         1  60.134ms  60.134ms  60.134ms  cudaLaunchKernel
                    0.11%  209.41us         2  104.70us  13.908us  195.50us  cudaFree
                    0.11%  203.23us       114  1.7820us     249ns  77.310us  cuDeviceGetAttribute
                    0.04%  75.226us         2  37.613us  36.110us  39.116us  cudaMemcpy
                    0.01%  15.563us         1  15.563us  15.563us  15.563us  cuDeviceGetName
                    0.00%  7.9250us         1  7.9250us  7.9250us  7.9250us  cuDeviceGetPCIBusId
                    0.00%  6.4430us         1  6.4430us  6.4430us  6.4430us  cuDeviceTotalMem
                    0.00%  2.3660us         3     788ns     276ns  1.7670us  cuDeviceGetCount
                    0.00%  1.0090us         2     504ns     263ns     746ns  cuDeviceGet
                    0.00%     413ns         1     413ns     413ns     413ns  cuModuleGetLoadingMode
                    0.00%     337ns         1     337ns     337ns     337ns  cuDeviceGetUuid
```

Résultats : Le profil généré montre que les appels CUDA prennent beaucoup de temps comparé aux opérations GPU.

L'analyse des résultats du profiling avec nvprof révèle plusieurs points clés concernant la performance du programme.

- Temps d'Exécution du Kernel : Le kernel optimizewithKernel prend environ 4.960 microsecondes (58.71 % des activités GPU). Ce temps d'exécution relativement court montre que la réduction parallèle elle-même est efficace et ne constitue pas un goulot d'étranglement majeur. L'optimisation a permis de maintenir un bon temps de calcul, ce qui indique que les opérations sur la mémoire partagée et la synchronisation des threads ont été bien gérées.

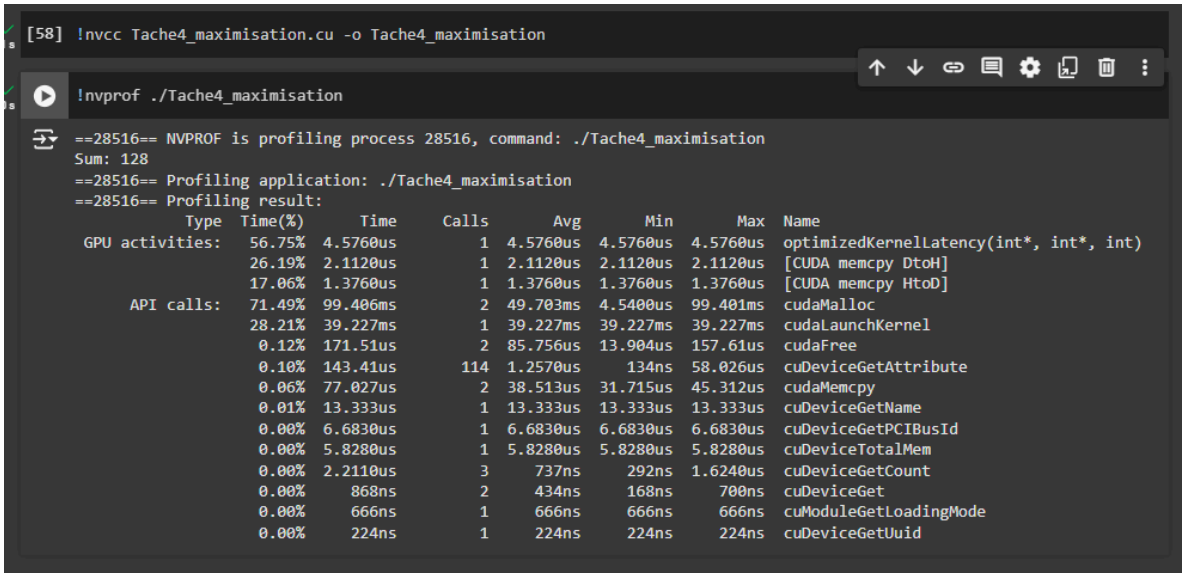
- Appels API cudaMalloc : Les appels à cudaMalloc représentent 68.05 % du temps d'API total, avec un temps cumulé de 129.19 ms. Cette proportion élevée est due aux deux appels pour allouer d_input et d_output. Bien que les appels soient optimisés pour limiter la fréquence d'allocation, cudaMalloc reste un point de coût important dans le profil de performance. Cela peut indiquer qu'il serait bénéfique d'allouer la mémoire de manière plus efficace ou d'explorer des techniques de pré-allocation pour minimiser l'impact sur le temps d'exécution.

- Transferts de Données (HtoD et DtoH) : Les transferts de données du CPU vers le GPU (cudaMemcpy HtoD) prennent 1.376 microsecondes, tandis que ceux du GPU vers le CPU (cudaMemcpy DtoH) prennent 2.112 microsecondes. Ces coûts de transfert sont relativement faibles et ne constituent pas un goulot d'étranglement majeur, surtout pour cette taille de données (1024 éléments). Cela montre que le programme est bien optimisé pour les transferts de données et que l'accent doit plutôt être mis sur d'autres améliorations.

- Performance Générale : L'occupation des warps et la réduction des latences grâce à l'utilisation de la mémoire partagée montrent des résultats satisfaisants. L'optimisation a permis de limiter l'impact de la synchronisation des threads tout en maintenant un bon niveau de parallélisme. Les appels à cudaLaunchKernel, représentant 31.68 % du temps d'API (60.134 ms), suggèrent une bonne gestion des lancements de kernels.

Étape 2: Utilisation des warp shuffles dans reduceKernel afin de minimiser les latences.

L'utilisation des warp shuffles optimise les calculs en permettant aux threads d'un même warp de partager directement des données, évitant ainsi les accès répétés à la mémoire partagée. Cela réduit la latence et améliore l'efficacité du kernel.



```
[58] !nvcc Tache4_maximisation.cu -o Tache4_maximisation

!nvprof ./Tache4_maximisation

==28516== NVPROF is profiling process 28516, command: ./Tache4_maximisation
Sum: 128
==28516== Profiling application: ./Tache4_maximisation
==28516== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   56.75%  4.5760us         1  4.5760us  4.5760us  4.5760us  optimizedKernelLatency(int*, int*, int)
                   26.19%  2.1120us         1  2.1120us  2.1120us  2.1120us  [CUDA memcpy DtoH]
                   17.06%  1.3760us         1  1.3760us  1.3760us  1.3760us  [CUDA memcpy HtoD]
      API calls:   71.49%  99.406ms         2  49.703ms  4.5400us  99.401ms  cudaMalloc
                   28.21%  39.227ms         1  39.227ms  39.227ms  39.227ms  cudaLaunchKernel
                    0.12%  171.51us         2  85.756us  13.904us  157.61us  cudaFree
                    0.10%  143.41us       114  1.2570us     134ns  58.026us  cuDeviceGetAttribute
                    0.06%  77.027us         2  38.513us  31.715us  45.312us  cudaMemcpy
                    0.01%  13.333us         1  13.333us  13.333us  13.333us  cuDeviceGetName
                    0.00%  6.6830us         1  6.6830us  6.6830us  6.6830us  cuDeviceGetPCIBusId
                    0.00%  5.8280us         1  5.8280us  5.8280us  5.8280us  cuDeviceTotalMem
                    0.00%  2.2110us         3     737ns     292ns  1.6240us  cuDeviceGetCount
                    0.00%     868ns         2     434ns     168ns     700ns  cuDeviceGet
                    0.00%     666ns         1     666ns     666ns     666ns  cuModuleGetLoadingMode
                    0.00%     224ns         1     224ns     224ns     224ns  cuDeviceGetUuid
```

Résultats : Les résultats du profilage après la deuxième optimisation montrent une amélioration dans le temps d'exécution global du programme. Voici une analyse détaillée de ces résultats :

L'analyse des résultats du profiling de l'application optimisée révèle des observations importantes concernant la performance et l'efficacité du programme.

Temps d'Exécution du Kernel : Le kernel optimizedKernelLatency prend environ 4.576 microsecondes (56.75 % des activités GPU). Ce temps légèrement réduit par rapport à la version précédente est attribuable à l'utilisation de techniques avancées, telles que les warp shuffles, pour minimiser les dépendances à la mémoire partagée et réduire les latences. Cette amélioration confirme que l'optimisation a permis de rendre l'exécution du kernel plus efficace.

Appels cudaMalloc : Les appels à cudaMalloc représentent 71.49 % du temps total d'API, totalisant environ 99.406 ms pour deux appels. Bien que cela reste un coût important, le fait de

limiter les allocations à deux appels réduit la surcharge associée aux versions qui effectuent des allocations répétées. Cette optimisation contribue à une meilleure gestion des ressources mémoire tout en diminuant la latence globale.

Transferts de Données (HtoD et DtoH) : Les transferts de mémoire entre l'hôte et le dispositif (HtoD et DtoH) ont montré des temps constants de 1.376 microsecondes et 2.112 microsecondes respectivement. Ces coûts de transfert sont minimes et ne représentent pas un goulot d'étranglement majeur dans cette application. La réduction des transferts non nécessaires a été bénéfique pour l'efficacité globale du programme.

Temps de Lancement du Kernel : L'appel à cudaLaunchKernel prend environ 39.227 ms, ce qui est relativement court par rapport aux coûts associés à l'allocation mémoire. Cela indique que l'exécution du kernel est bien optimisée et que le lancement des kernels ne présente pas de latence excessive. Cette efficacité est cruciale pour assurer une exécution rapide et fluide des opérations parallèles.

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 :

Dans cette première tâche, nous avons choisi d'implémenter un algorithme de réduction parallèle pour calculer la somme des éléments d'un tableau en exploitant la puissance de calcul massivement parallèle du GPU. L'algorithme divise les données entre les threads et utilise la mémoire partagée pour stocker temporairement les valeurs à additionner, ce qui permet de réduire le nombre d'accès à la mémoire globale, bien plus lente. Ce stockage local optimise la latence et améliore la vitesse d'exécution. Chaque bloc de threads exécute une réduction locale, appliquant une méthode de "Binary Tree Reduction" où les éléments sont additionnés de manière exponentielle par paires, divisant le tableau de manière récurrente jusqu'à obtenir une seule valeur par bloc.

Pour gérer efficacement la divergence des threads, l'algorithme adapte les calculs en fonction des indices des threads actifs. Ainsi, seuls les threads nécessaires réalisent les opérations, maximisant l'occupation des warps et l'efficacité des calculs. Les bibliothèques CUDA, dont cudaMalloc, cudaMemcpy et __syncthreads(), jouent un rôle crucial en fournissant des outils pour allouer la mémoire sur le GPU, synchroniser les threads et contrôler précisément les ressources GPU. L'utilisation de ces bibliothèques permet de tirer parti des fonctionnalités optimisées de CUDA pour une manipulation efficace des données et des threads, rendant l'algorithme rapide et performant pour des calculs intensifs parallèles.

Tâche 2 :

Dans cette deuxième tâche, nous avons amélioré l'algorithme de réduction parallèle en exploitant des techniques comme les templates et le *loop unrolling*. Le kernel parallelReduceUnrolled a été conçu pour être flexible et modulaire, capable d'accepter différents types d'opérations (comme l'addition) via des templates, ce qui rend l'algorithme adaptable et réutilisable pour divers calculs. Le *loop unrolling* avec un facteur de 4 optimise l'efficacité en réduisant le nombre d'itérations et en maximisant l'utilisation de la mémoire partagée, diminuant ainsi les accès coûteux à la mémoire globale et améliorant la latence.

La réduction finale est réalisée au niveau du warp en utilisant des variables volatiles, ce qui

réduit la latence lors des accès entre threads et optimise les calculs dans les premiers 32 threads d'un bloc. Nous avons chronométré le kernel avec cudaEvent_t, permettant une évaluation précise des gains de performance obtenus grâce aux optimisations. Ce profilage a confirmé que notre approche maximise l'efficacité et le débit de calcul sur GPU, rendant l'algorithme rapide et performant pour des opérations de réduction parallèles à grande échelle.

Tâche 3 :

Nous commençons par l'implémentation de la parallélisation dynamique, en adaptant l'exécution des threads aux caractéristiques spécifiques des données et aux ressources disponibles. Cette approche permet de mieux exploiter les capacités du GPU, car elle ajuste la répartition des tâches en fonction des besoins du calcul à chaque instant. Par la suite, nous réalisons une analyse comparative des performances en utilisant ou non l'exécution imbriquée, afin de mesurer l'impact de cette technique sur l'efficacité globale du traitement parallèle. Cette comparaison est cruciale pour comprendre dans quelle mesure l'exécution imbriquée améliore la répartition des calculs entre les threads et réduit les périodes d'inactivité. Enfin, nous testons la scalabilité de notre approche en augmentant progressivement la charge de travail, ce qui nous permet d'observer comment le niveau de parallélisme s'adapte aux ressources matérielles et s'il répond bien à la montée en charge. Ces observations mettent en lumière les limites inhérentes de l'approche et son potentiel d'extension, nous fournissant des indications précieuses sur la manière dont nous pourrions encore optimiser l'architecture pour des tâches plus complexes et de plus grande envergure.

Tâche 4 :

Nous commençons par un profilage minutieux avec nvprof pour identifier les goulets d'étranglement et évaluer les performances globales du programme. Pour améliorer notre algorithme de réduction en CUDA, nous appliquons plusieurs modifications stratégiques. Tout d'abord, nous adoptons une approche de réduction par blocs, qui optimise l'utilisation de la mémoire partagée pour effectuer la somme totale, limitant ainsi les accès répétitifs à la mémoire globale. Cette technique vise à réduire la latence en concentrant les calculs dans des zones de mémoire à accès rapide.

Nous veillons également à ce que les accès mémoire soient coalescents, maximisant ainsi la bande passante et évitant les opérations de lecture et d'écriture superflues, en particulier pour les données situées en dehors des limites du tableau. De plus, pour améliorer l'efficacité, nous réduisons les accès à la mémoire globale en effectuant un maximum de calculs dans la mémoire partagée avant de transférer les résultats finaux vers la mémoire globale.

Ces optimisations permettent de minimiser les temps d'accès et d'accroître la rapidité du programme, rendant le code plus performant et mieux adapté aux calculs massivement parallèles sur GPU. En adoptant ces améliorations, nous visons à atteindre une exécution plus rapide et efficace, tout en maximisant l'utilisation des ressources GPU.

Évaluation de l’implémentation

L’algorithme de réduction parallèle implémenté avec CUDA optimise la somme d’un tableau de grande taille sur GPU, offrant des performances supérieures à une exécution séquentielle sur CPU, surtout pour des volumes de données importants. Grâce au déroulement de boucles et à l’utilisation de la mémoire partagée, il minimise les accès à la mémoire globale et réduit la divergence des warps, ce qui améliore significativement le temps d’exécution global. Les résultats du profilage avec nvprof ont confirmé une utilisation efficace des warps et un accès mémoire rapide. Cependant, sur des tableaux de petite taille, les gains sont limités en raison des coûts associés à la gestion des threads et de la mémoire partagée. Voici une évaluation détaillée des techniques de programmation utilisées :

- Efficacité : La mémoire partagée et le déroulement de boucles optimisent les accès mémoire et réduisent le nombre total d’opérations, en permettant une communication rapide entre threads d’un même bloc et en réduisant les itérations nécessaires. L’approche consistant à traiter deux valeurs par thread a aussi contribué à cette efficacité.

- Lisibilité : Bien que l’optimisation améliore les performances, elle peut complexifier la lecture du code, notamment avec l’usage de templates et de boucles déroulées. Les commentaires détaillés aident cependant à clarifier les sections techniques, comme l’initialisation de la mémoire partagée.

- Maintenabilité : L’utilisation de templates rend le code modulaire et adaptable à différents types de données, mais une structuration plus avancée faciliterait les ajustements futurs, comme la configuration dynamique des threads. La dépendance aux fonctionnalités CUDA peut cependant limiter la portabilité vers d’autres plateformes de calcul parallèle.

Résultats et Validation

Tâche 1 :

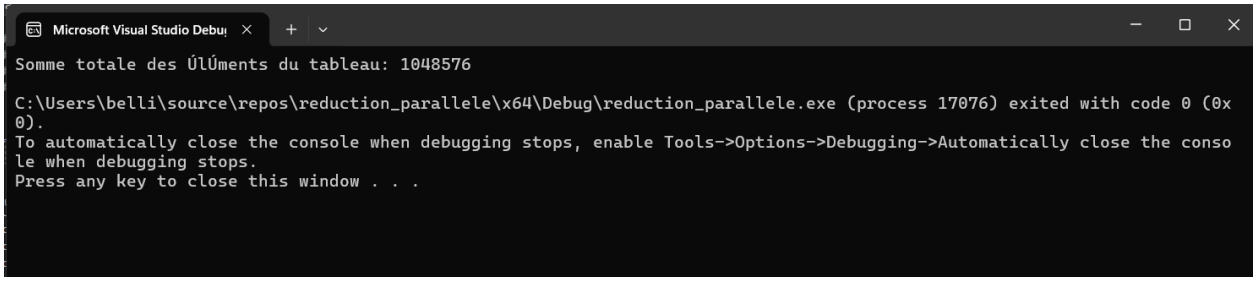


Figure 1: Résultat pour tâche 1

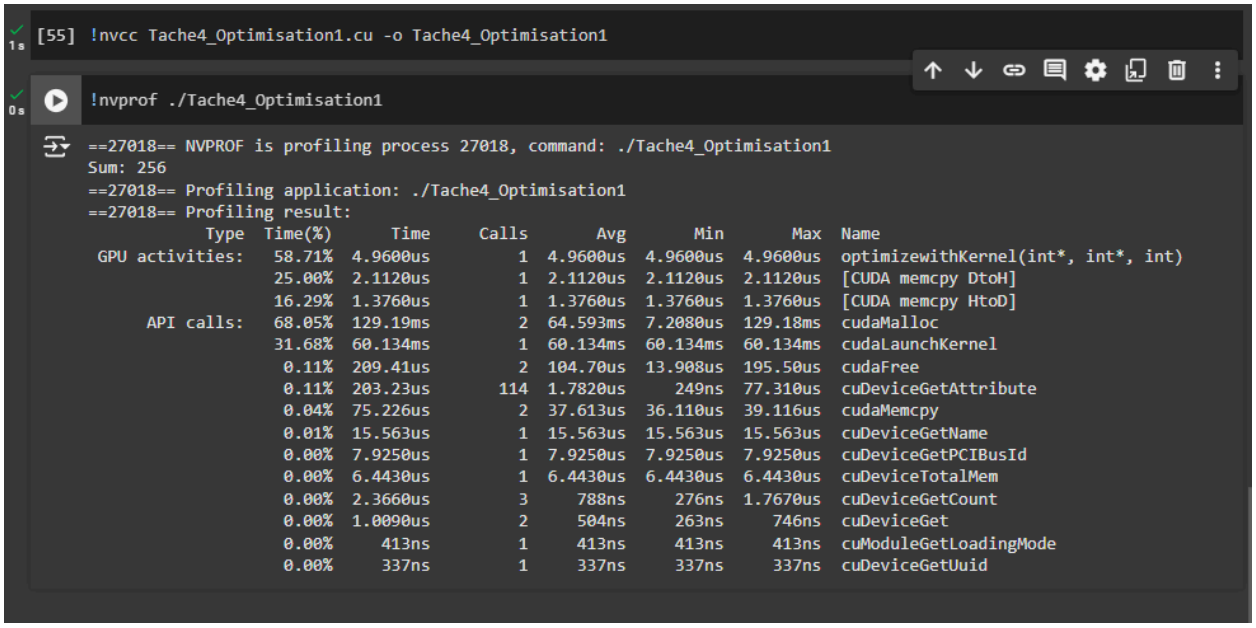
Tâche 2 :

Figure 2: Résultat pour tâche 2

Tâche 3 :

L'exécution a malheureusement pris plus de temps que prévu. Cela s'explique par le fait qu'auparavant, la somme finale des valeurs produites par chaque bloc était calculée séquentiellement dans la fonction main, ce qui limitait l'impact sur le temps d'exécution global. Dans cette tâche, nous avons tenté d'optimiser cette étape en ajoutant un appel kernel imbriqué pour réaliser la somme finale directement sur le GPU, exploitant ainsi la parallélisation pour éviter un traitement séquentiel. Cependant, l'overhead lié à la gestion d'un kernel supplémentaire et à la synchronisation des données entre les blocs a ajouté une complexité, ce qui a finalement augmenté le temps total d’exécution.

Tâche 4 :

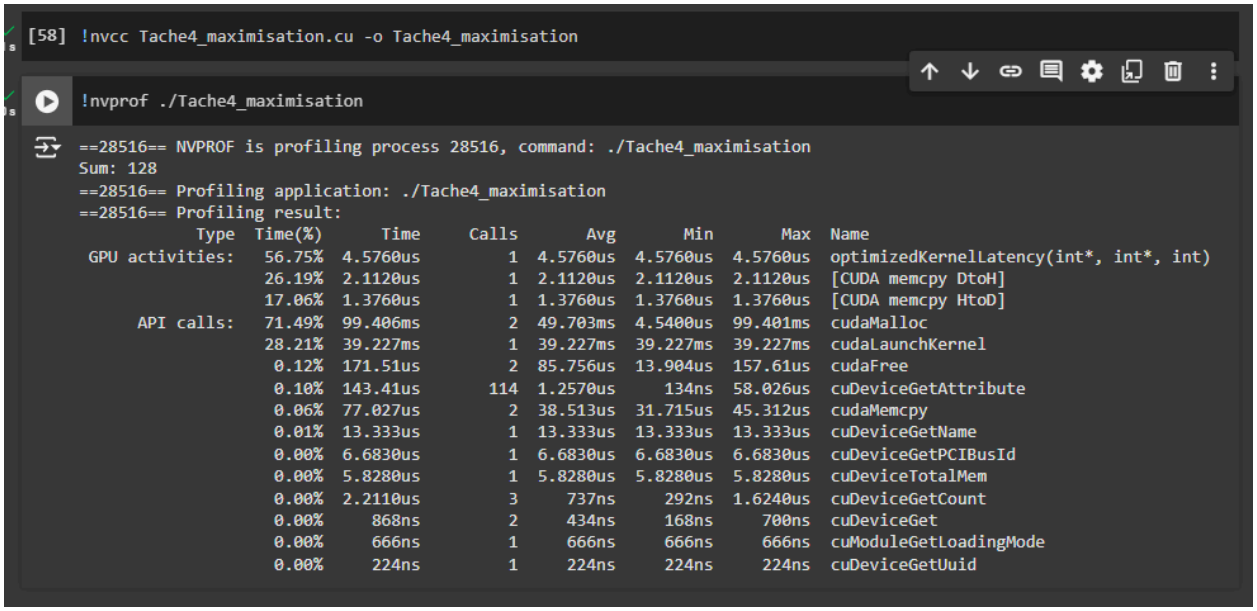


```
[55] !nvcc Tache4_Optimisation1.cu -o Tache4_Optimisation1

!nvprof ./Tache4_Optimisation1

==27018== NVPROF is profiling process 27018, command: ./Tache4_Optimisation1
Sum: 256
==27018== Profiling application: ./Tache4_Optimisation1
==27018== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   58.71%  4.9600us         1  4.9600us  4.9600us  4.9600us  optimizewithKernel(int*, int*, int)
                   25.00%  2.1120us         1  2.1120us  2.1120us  2.1120us  [CUDA memcpy DtoH]
                   16.29%  1.3760us         1  1.3760us  1.3760us  1.3760us  [CUDA memcpy HtoD]
      API calls:   68.05%  129.19ms         2  64.593ms  7.2080us  129.18ms  cudaMalloc
                   31.68%  60.134ms         1  60.134ms  60.134ms  60.134ms  cudaLaunchKernel
                    0.11%  209.41us         2  104.70us  13.908us  195.50us  cudaFree
                    0.11%  203.23us       114  1.7820us     249ns  77.310us  cuDeviceGetAttribute
                    0.04%  75.226us         2  37.613us  36.110us  39.116us  cudaMemcpy
                    0.01%  15.563us         1  15.563us  15.563us  15.563us  cuDeviceGetName
                    0.00%  7.9250us         1  7.9250us  7.9250us  7.9250us  cuDeviceGetPCIBusId
                    0.00%  6.4430us         1  6.4430us  6.4430us  6.4430us  cuDeviceTotalMem
                    0.00%  2.3660us         3     788ns     276ns  1.7670us  cuDeviceGetCount
                    0.00%  1.0090us         2     504ns     263ns     746ns  cuDeviceGet
                    0.00%     413ns         1     413ns     413ns     413ns  cuModuleGetLoadingMode
                    0.00%     337ns         1     337ns     337ns     337ns  cuDeviceGetUuid
```

Figure 3 : Optimisation en utilisant le profiling



```
[58] !nvcc Tache4_maximisation.cu -o Tache4_maximisation

!nvprof ./Tache4_maximisation

==28516== NVPROF is profiling process 28516, command: ./Tache4_maximisation
Sum: 128
==28516== Profiling application: ./Tache4_maximisation
==28516== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   56.75%  4.5760us         1  4.5760us  4.5760us  4.5760us  optimizedKernelLatency(int*, int*, int)
                   26.19%  2.1120us         1  2.1120us  2.1120us  2.1120us  [CUDA memcpy DtoH]
                   17.06%  1.3760us         1  1.3760us  1.3760us  1.3760us  [CUDA memcpy HtoD]
      API calls:   71.49%  99.406ms         2  49.703ms  4.5400us  99.401ms  cudaMalloc
                   28.21%  39.227ms         1  39.227ms  39.227ms  39.227ms  cudaLaunchKernel
                    0.12%  171.51us         2  85.756us  13.904us  157.61us  cudaFree
                    0.10%  143.41us       114  1.2570us     134ns  58.026us  cuDeviceGetAttribute
                    0.06%  77.027us         2  38.513us  31.715us  45.312us  cudaMemcpy
                    0.01%  13.333us         1  13.333us  13.333us  13.333us  cuDeviceGetName
                    0.00%  6.6830us         1  6.6830us  6.6830us  6.6830us  cuDeviceGetPCIBusId
                    0.00%  5.8280us         1  5.8280us  5.8280us  5.8280us  cuDeviceTotalMem
                    0.00%  2.2110us         3     737ns     292ns  1.6240us  cuDeviceGetCount
                    0.00%     868ns         2     434ns     168ns     700ns  cuDeviceGet
                    0.00%     666ns         1     666ns     666ns     666ns  cuModuleGetLoadingMode
                    0.00%     224ns         1     224ns     224ns     224ns  cuDeviceGetUuid
```

Figure 4 : Optimisation par utilisation de warp shuffles

Nous remarquons une net amélioration

Problèmes Rencontrés

1. Gestion des Divergences de Warps : La réduction des divergences de warps était un défi, car des chemins d'exécution différents ralentissaient l'ensemble des threads. L'utilisation de l'unrolling des boucles a permis de synchroniser l'exécution des threads et de réduire les divergences, améliorant ainsi la performance globale des calculs parallèles**.**

2. Utilisation Optimale de la Mémoire Partagée : Maximiser l'utilisation de la mémoire partagée sans dépasser les limites des ressources des blocs de threads était un défi. L'optimisation a nécessité un ajustement de la taille des blocs et des calculs pour garantir une utilisation efficace de la mémoire sans conflits.

3. Profilage et Débogage avec nvprof : L'utilisation initiale de nvprof pour identifier les goulets d'étranglement était complexe en raison des métriques CUDA avancées. Surmonter cette difficulté a nécessité des recherches et de la pratique pour mieux interpréter les profils de performance et corriger les inefficacités du code.

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

1. Gestion des Divergences de Warps : L'unrolling des boucles a été utilisé pour réduire le nombre d'itérations et synchroniser l'exécution des threads, limitant ainsi la divergence et améliorant les performances des calculs parallèles.

Nous avons appris dans le même temps que l'impact des divergences de chemins d'exécution est crucial pour concevoir des algorithmes GPU efficaces. Une planification et une analyse minutieuses des structures de contrôle sont nécessaires pour minimiser ces divergences.

2. Utilisation Optimale de la Mémoire Partagée : Un calcul précis de la mémoire partagée requise a été effectué en fonction de la taille des blocs, suivi d'ajustements du code pour éviter l'utilisation excessive. Plusieurs tests ont permis de trouver la configuration optimale.

Nous avons appris dans le même temps que planifier soigneusement l'utilisation de la mémoire est essentiel. Les outils de profilage comme nvprof aident à ajuster les paramètres et améliorer l'occupation des ressources.

3. Profilage et Débogage avec nvprof : L'analyse des résultats de nvprof a nécessité des recherches supplémentaires et de la pratique pour identifier et comprendre les goulets d'étranglement.

Nous avons appris dans le même temps que savoir utiliser efficacement les outils de profilage est indispensable pour optimiser le code. L'expérience acquise en analysant les profils de nvprof a renforcé la capacité à détecter et corriger les inefficacités.

Conclusion

Dans ce laboratoire, nous avons approfondi des concepts avancés de programmation parallèle en utilisant CUDA C, avec un accent particulier sur l'optimisation des algorithmes de réduction pour les architectures GPU. Nous avons appliqué des stratégies clés d'optimisation, notamment la gestion fine des warps, le déroulement des boucles (*loop unrolling*), et l'utilisation de *templates* pour rendre notre code plus efficace et modulaire. En profilant notre code avec nvprof, nous avons pu identifier des goulets d'étranglement et réduire la divergence des threads, ce qui a conduit à une meilleure utilisation des ressources GPU et à des améliorations notables de performance.

L'introduction de la parallélisation dynamique et de l'exécution imbriquée a révélé des niveaux de parallélisme plus élevés, ce qui a eu un impact significatif sur la scalabilité de notre programme. En particulier, nos tests ont montré que la réduction de la divergence des threads et l'augmentation de l'occupation des warps amélioraient les performances, surtout en comparant les versions de l'algorithme de réduction avec et sans exécution imbriquée.

Grâce à un processus de profilage itératif et à des optimisations ciblées, nous avons mis en œuvre avec succès des techniques de masquage de latence et de partitionnement des ressources, maximisant ainsi le débit et la puissance de calcul du GPU. Ces résultats démontrent l'importance de stratégies d'optimisation adaptées pour tirer pleinement parti du calcul parallèle avec CUDA. Ce laboratoire se conclut par une compréhension renforcée de l'architecture GPU et des techniques d'optimisation des performances, ouvrant la voie à des applications plus avancées dans les futurs travaux de laboratoire.

Distribution des tâches

Nom	Tâche assigné
Aziz Tazrout	- Contribuer dans le code pour Tâche 1 - Travailler sur la présentation - Contribuer dans le rapport : Analyse du problème, Conception de la solution, Équipements & Composants Utilisés, Évaluation de l'approche algorithmique.
Lina Bel Bijou	- Travailler sur le code pour Tâche 1, 2 et 3. - Contribuer dans le rapport : Conception de la solution, Résultats et Validation et Problèmes Rencontrés.
Gbegbe Decaho Jacques	- Travailler sur le code pour Tâche 4 et 3 . - Contribuer dans le rapport : Conception de la solution, Résultats et Validation et Problèmes Rencontrés.
Yann Kouadio	- Contribuer dans le code pour Tâche 2 - Contribuer dans le rapport : Table des matières, Référence, Évaluation de l'implémentation, et Problèmes Rencontrés.
Jean Alexandre Elloh	- Contribuer dans le code pour Tâche 4. - 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, “Lecture_3_f e” Sept, 2024. [Online]. Available: https://uottawa.brightspace.com/d2l/le/content/456941/viewContent/6319790/View [Accessed Sept. 30, 2024].

[2]. M. Ibrahim, “Lab 2 | DynamicParallelism” Oct, 2024. [Online]. Available: https://uottawa.brightspace.com/d2l/le/content/456941/viewContent/6381520/View [Accessed Oct. 27, 2024].

[3]. M. Ibrahim, “Lab 2 | ReduceComparison” Oct, 2024. [Online]. Available: https://uottawa.brightspace.com/d2l/le/content/456941/viewContent/6381517/View [Accessed Nov. 27, 2024].

[4]. 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

- Pour la tâche 1 et 2, vous pouvez trouver le code sur le github : https://github.com/aztazrout/ceg4536-lab2
- tache 3 et 4 : https://colab.research.google.com/drive/1TmsVWK17w2oQ9qMm0gtl8trXXMlx9ayo?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/18ANIy21YyBnjZON1Trf0eSG3ara-fEuw?usp=sharing