

Lab 2: Optimisation et Exécution Parallèle avec CUDA C

CEG 4536 – Architecture des ordinateurs III
Automne 2024
École de Génie Électrique et Science Informatique
Université d'Ottawa
Professeur: Mohamed Ali Ibrahim

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: 21, 28 Octobre 2024

Date de la soumission: 4 Novembre 2024

Table of Contents

Table of Contents.....	2
Table of Figures.....	2
Table of Tables.....	2
Introduction.....	3
Objectifs.....	3
Analyse du problème.....	3
Conception de la solution.....	4
Équipements & Composants Utilisés.....	6
Procédure de résolution et algorithme.....	7
Évaluation d'implémentation.....	8
Résultats et Validation.....	12
Tâche 4: Le code d'optimisation des profils de la tâche 4 est un programme optimisé pour CUDA qui démontre des améliorations mesurables en termes de performances et d'efficacité, ainsi qu'un rapport détaillé documentant ces améliorations. Nous avons modifier le code de la tâche 3 pour inclure des ajustements aux tailles de blocs, aux modèles d'accès à la mémoire, à l'utilisation d'opérations asynchrones ou à d'autres techniques pour maximiser l'occupation de la chaîne et minimiser la latence. Nous avons utilisé les fonctions –metrics et –print-gpu-trace avec nvprof pour approfondir les domaines où l'on perd du temps. Cela implique l'analyse des temps d'exécution du noyau, des transferts de mémoire et d'autres mesures de performances pour identifier les goulots d'étranglements. En observant ces métriques, on pouvait identifier et éliminer les goulots d'étranglements pour ensuite, de façon essaie erreur, maximiser l'occupation des warps et minimiser la latence.....	13
Problèmes Rencontrés.....	13
Conclusion.....	14
Distribution de tâches.....	15
Références.....	15
Appendice.....	15

Table of Figures

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

Table of Tables

Table 1.....	13
---------------------	-----------

Introduction

Dans ce laboratoire, nous allons explorer et mettre en pratique des concepts avancés de programmation parallèle sur GPU avec CUDA C. Nous apprendrons à optimiser des algorithmes en approfondissant des mécanismes clés, tels que la gestion des warps, l'exécution dynamique et les techniques visant à minimiser la divergence des threads. À l'aide de l'outil 'nvprof', nous analyserons notre code pour détecter les goulets d'étranglement et améliorer les performances. Ce laboratoire se déroulera en plusieurs phases, comprenant l'implémentation, l'optimisation et l'analyse de performances d'algorithmes de réduction parallèle. Nous aborderons également la parallélisation dynamique et l'exécution imbriquée afin d'exploiter davantage le parallélisme offert par l'environnement GPU.

Objectifs

- Compréhension 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 tâche, il nous faut créer un code de base faisant appel à un kernel qui lui effectuera une réduction parallèle pour additionner les éléments d'un tableau. Une fois créé, il faut tester ce code. Il faut aussi utiliser nvprof pour mesurer les warps actifs et les opérations de mémoire. Ceci devrait permettre d'identifier les problèmes de divergence des warps. On

obtiendra une première ébauche de notre code de réduction parallèle qui sera prêt pour davantage d'optimisation dans les prochaines tâches.

Tâche 2:

Après avoir créé une ébauche de code effectuant une réduction et l'avoir examiné avec nvprof, nous savons comment améliorer le code en diminuant la divergence des warps. Ceci a été permis par la réalisation de la tâche 1. la boucle "for" arrive à des décalage inférieur à 32, cela crée de la divergence entre les threads d'un même warp. Il faut arrêter la boucle for lorsqu'on atteint la valeur de décalage de 32, puis faire les réductions/additions restantes manuellement avec des variables volatiles. Il nous faut aussi utiliser des techniques de unrolling des boucles dans le kernel, en dupliquant le contenu de la boucle et réduisant le nombre d'itération. On va aussi ajouter des fonctions template pour permettre la méthode de réduction pour multiplication et addition, et pour plusieurs types de variables : Ceci rendra notre kernel de réduction plus générique et modulaire.

Tâche 3:

Cette tâche explore des techniques d'optimisation avancées pour exploiter pleinement la puissance de calcul du GPU grâce à la parallélisation dynamique et à l'exécution imbriquée. En utilisant la parallélisation dynamique, nous permettons aux threads de lancer de nouveaux threads, augmentant ainsi la capacité de parallélisme au-delà des configurations classiques. L'objectif est d'améliorer l'efficacité des opérations de réduction, en comparant les performances obtenues avec et sans l'exécution imbriquée, ainsi que d'analyser la scalabilité en exposant davantage de parallélisme dans notre programme.

Tâche 4:

Cette tâche se concentre sur le profiling et l'optimisation basée sur les profils pour identifier et éliminer les goulets d'étranglement qui limitent les performances du programme. L'outil nvprof de NVIDIA sera utilisé pour mesurer en détail les performances, détecter les latences, et observer l'occupation des warps et l'utilisation des ressources. L'objectif est de maximiser la performance du GPU en améliorant l'occupation des ressources et en optimisant la gestion de la latence. Pour améliorer la performance, nous appliquerons des techniques de masquage de la latence, en exploitant le partitionnement des ressources pour garantir un haut degré d'occupation des warps et minimiser les périodes d'inactivité des threads.

Conception de la solution

Tâche 1:

Dans cette tâche, on commence par coder la méthode main. Dans celle-ci, on commence par initialiser les tailles du tableau, du bloc et de la grille. Ensuite, on alloue des tableaux d'entrées et de sorties dans le host et le device avec malloc et cudaMalloc. Puis, on initialise les valeurs du tableau d'entrée et on copie du host au device avec cudaMemcpy. On fait appel au kernel qui effectue la réduction parallèle, et on recopie les données du output du device au host. On peut afficher ce résultat. Finalement, on libère la mémoire allouée pour le device et le host.

Du côté kernel, on prend en argument les tableaux d'entrées et de sortie, et la taille du tableau d'entrée. On commence par déclarer un tableau d'entier (sharedData) partagé dans chaque bloc qui servira à stocker les calculs intermédiaires dans la mémoire partagée (accélérant l'accès des données par rapport à la mémoire globale). On calcul et sauvegarde la valeur de l'id dans le bloc dans tid, et on calcul l'index aussi. Si l'index est inférieur à la taille du tableau, on sauvegarde cette valeur dans sharedData avec comme position la valeur du tid, sinon on sauvegarde 0. On synchronise les threads pour qu'il ai complété tous jusqu'à cette ligne. On utilise une boucle for pour faire la réduction parallèle des valeurs sauvegardé dans sharedData, en utilisant l'approche par paire entrelacée: À chaque itération, la valeur de offset s' est divisée par deux. Si le tid est inférieur à s, alors on calcul la somme de la valeur à la position tid avec la valeur à la position tid + s. On synchronise les threads du bloc, avant de passer à la prochaine itération. Finalement, un fois sortie de la boucle, si tid est 0 alors, on sauvegarde la valeur de sharedData à position tid dans le output.

Tâche 2:

En utilisant le code réalisé dans la tâche 1, on ajoute les template pour pouvoir effectuer la réduction sur plusieurs types de variable. On ajoute le mot clés template et typename T avant le kernel. On modifie les paramètres input et output à des pointeurs de T. Dans le kernel, on change sharedData aussi au type T. Ceci nous permet de faire la réduction sur des types de variables différents.

Pour ce qui est d'utiliser des template pour le type d'opération soit addition soit multiplication, on crée deux structure Add et Multiply avec template T qui contiennent chacun une fonction __device__ operator avec deux arguments a et b et effectue soit la multiplication soit l'addition. Ensuite, avant le kernel on ajoute au template, le typename Op pour le type d'opération. On ajoute les paramètres Op op et T neutralElement au kernel. Il nous suffit de changer l'opération dans la boucle for par (ce qui permet d'utiliser l'opération de notre choix):

```
sharedData[tid] = op(sharedData[tid], sharedData[tid + s]);
```

À la fin du code main, on ajoute aussi un exemple où la réduction utilise la multiplication.

Pour effectuer le unrolling de boucle, il suffit de dupliquer quatres fois le contenu de la boucle on

remplace

```
- sharedData[tid] = op(sharedData[tid], sharedData[tid + s]);      par
- sharedData[tid] = op(sharedData[tid], sharedData[tid + s]);
sharedData[tid] = op(sharedData[tid], sharedData[tid + s * 2]);
sharedData[tid] = op(sharedData[tid], sharedData[tid + s * 3]);
```

Il faut aussi changer le contrôle de la boucle for par une division par 4 au lieu de 2:

```
-(int s = blockDim.x / 4; s > 0; s /= 4)
```

Pour améliorer la réduction en réduisant la divergence de warp, il faut remplacer la boucle for pour laquelle arrête lorsque 32 éléments restent à réduire:

```
- for (int s = blockDim.x / 4; s > 32; s /= 4)
```

Après cela, il faut ajouter les lignes suivantes, pour additionner les 32 éléments restant:

```
if (tid < 32) {
    volatile int* vsmem = sharedData;
    vsmem[tid] += vsmem[tid + 32];
    vsmem[tid] += vsmem[tid + 16];
    vsmem[tid] += vsmem[tid + 8];
    vsmem[tid] += vsmem[tid + 4];
    vsmem[tid] += vsmem[tid + 2];
    vsmem[tid] += vsmem[tid + 1];
}
```

Une autre optimisation de la divergence de warp est d'utiliser des blocs de taille multiple 32, et calculer la taille de la grille basée sur la taille du tableau à réduire. C'est ce qu'on a implémenté dans le main.

Tâche 3:

Dans cette tâche, il faut ajouter un appel à un nouveau kernel de manière imbriquée, pour terminer la réduction avec les valeurs sauveées par chaque bloc. Ce qui n'a pas été fait dans les tâches précédentes.

Tâche 4:

Pour concevoir la solution de réduction en CUDA, on a choisi d'utiliser la mémoire partagée pour stocker temporairement les données et réduire les accès à la mémoire globale. L'algorithme se décompose en plusieurs étapes : chaque thread charge des éléments dans la mémoire partagée, effectue des additions en parallèle, puis le résultat final est écrit dans la mémoire globale. On utilise également des techniques de déroulage pour améliorer l'efficacité des calculs et minimiser les synchronisations entre les threads.

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 goulots d'étranglement et d'optimiser l'occupation des warps et l'utilisation de la mémoire.

- CUDA Debugger

Le débogueur CUDA est essentiel pour tester et profiler les kernels CUDA. Il permet de diagnostiquer les erreurs dans le code CUDA en exécutant les threads GPU pas à pas. Cela aide à identifier les erreurs de synchronisation ou de gestion de mémoire, ce qui est crucial.

Procédure de résolution et algorithme

Tâche 1: Une fois le code généré comme expliqué à la section conception. Il suffit de programmer avec CUDA et C++. Ensuite, il nous faut exécuter pour vérifier le fonctionnement du code. À l'aide de nvprof on mesure les warps actifs et analyse les opérations mémoire. Ceci nous permettra de trouver les problèmes de divergence des warps dans notre programme et proposer des améliorations à effectuer dans la tâche 2.

Tâche 2: Une fois le code généré comme expliqué à la section conception. Il suffit d' améliorer le code comme décrit plutôt.

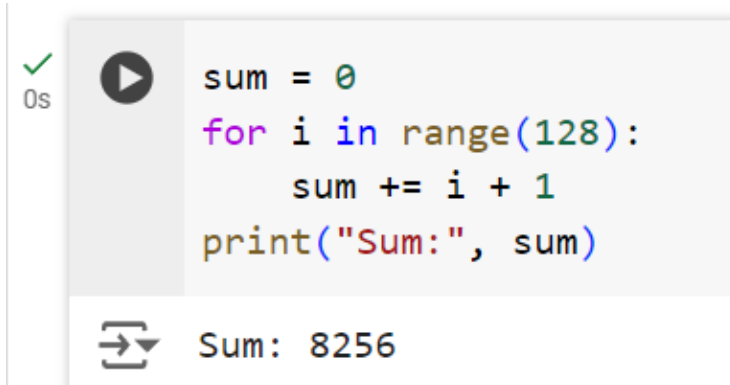
Tâche 3: On commence par implémenter la parallélisation dynamique pour adapter l'exécution des threads en fonction des données et des ressources disponibles. Ensuite, on compare les performances avec et sans exécution imbriquée pour mesurer l'impact de cette approche sur l'efficacité du traitement parallèle. Enfin, on analyse la scalabilité en augmentant progressivement la charge de travail afin de voir comment le parallélisme est exploité et s'adapte aux ressources, exposant ainsi les limites et le potentiel d'extension de l'approche adoptée.

Tâche 4: On commence par un profilage avec nvprof pour repérer les goulets d'étranglement et mesurer les performances. Ensuite pour améliorer le code de réduction en CUDA, on propose plusieurs modifications. D'abord, on utilise un algorithme de réduction par blocs qui exploite la mémoire partagée de manière optimale pour effectuer le calcul de la somme totale, en minimisant les accès à la mémoire globale. On s'assure que les accès à la mémoire sont coalescents pour maximiser la bande passante, en évitant les lectures et écritures inutiles, surtout pour les éléments en dehors des limites du tableau. De plus, on réduit les accès à la mémoire pour les résultats intermédiaires en effectuant autant de calculs que possible dans la mémoire partagée avant de stocker le résultat dans la mémoire globale. En appliquant ces optimisations, on vise à rendre le code plus efficace et rapide.

Évaluation d'implémentation

Tâche 1:

On effectue le calcul de sum en python sur google colab pour référence:

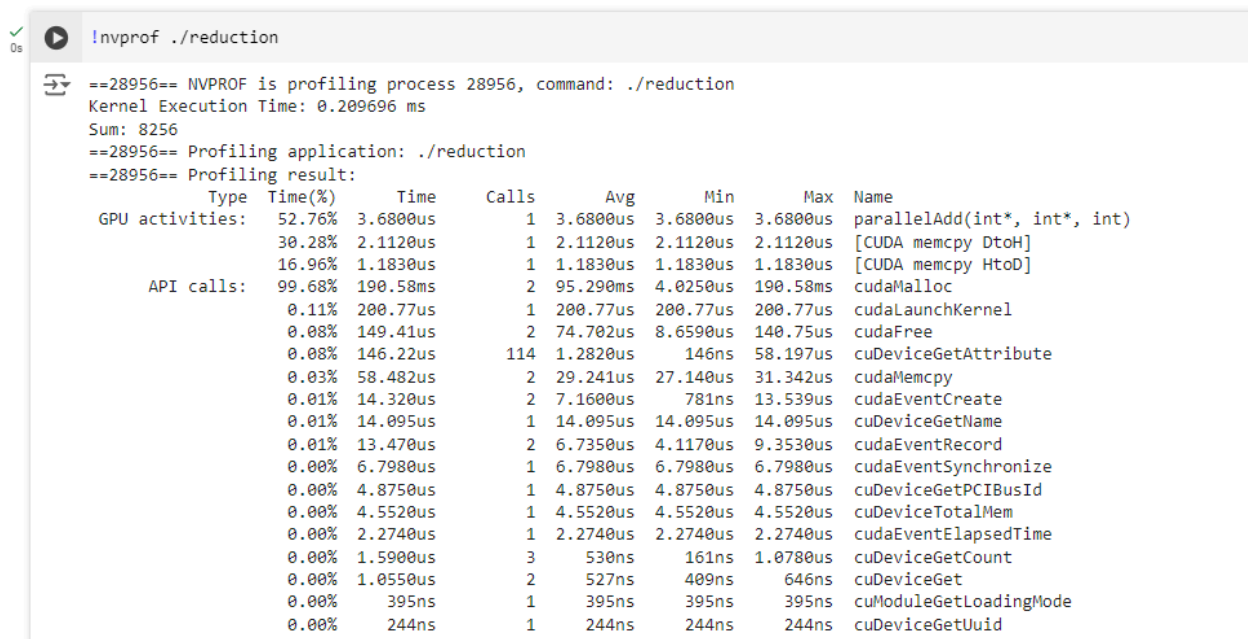


```
sum = 0
for i in range(128):
    sum += i + 1
print("Sum:", sum)
```

Sum: 8256

Figure 1: Somme de manière séquentielle

Comme on peut voir, le code réalise la bonne somme ci-dessous sur google colab. (de 128 élément ayant pour valeur $i+1$ (i étant la valeur de leur position))



```
!nvprof ./reduction

==28956== NVPROF is profiling process 28956, command: ./reduction
Kernel Execution Time: 0.209696 ms
Sum: 8256
==28956== Profiling application: ./reduction
==28956== Profiling result:
Type      Time(%)    Time      Calls      Avg      Min      Max      Name
GPU activities: 52.76%  3.6800us    1  3.6800us  3.6800us  3.6800us  parallelAdd(int*, int*, int)
                30.28%  2.1120us    1  2.1120us  2.1120us  2.1120us  [CUDA memcpy DtoH]
                16.96%  1.1830us    1  1.1830us  1.1830us  1.1830us  [CUDA memcpy HtoD]
API calls: 99.68% 190.58ms    2  95.290ms  4.0250us  190.58ms  cudaMalloc
                0.11%  200.77us    1  200.77us  200.77us  200.77us  cudaLaunchKernel
                0.08%  149.41us    2  74.702us  8.6590us  140.75us  cudaFree
                0.08%  146.22us   114  1.2820us   146ns   58.197us  cuDeviceGetAttribute
                0.03%  58.482us    2  29.241us  27.140us  31.342us  cudaMemcpy
                0.01%  14.320us    2  7.1600us   781ns   13.539us  cudaEventCreate
                0.01%  14.095us    1  14.095us  14.095us  14.095us  cuDeviceGetName
                0.01%  13.470us    2  6.7350us  4.1170us  9.3530us  cudaEventRecord
                0.00%  6.7980us    1  6.7980us  6.7980us  6.7980us  cudaEventSynchronize
                0.00%  4.8750us    1  4.8750us  4.8750us  4.8750us  cuDeviceGetPCIBusId
                0.00%  4.5520us    1  4.5520us  4.5520us  4.5520us  cuDeviceTotalMem
                0.00%  2.2740us    1  2.2740us  2.2740us  2.2740us  cudaEventElapsedTime
                0.00%  1.5900us    3    530ns    161ns   1.0780us  cuDeviceGetCount
                0.00%  1.0550us    2    527ns    409ns    646ns    cuDeviceGet
                0.00%    395ns    1    395ns    395ns    395ns    cuModuleGetLoadingMode
                0.00%    244ns    1    244ns    244ns    244ns    cuDeviceGetUuid
```

Figure 2: Exécution de nvprof sur la tâche 1

Ici, nvprof nous permet d'analyser les opérations du gpu.

On réalise que l'utilisation de taille bloc inférieur à 32, crée de la divergence. En utilisant des tailles de bloc minimum de 32, on évite en partie la divergence.

Une autre réalisation par rapport à la performance, est que quand la boucle “for” arrive à des décalage inférieur à 32, cela crée de la divergence entre les threads d’un même warp. Il faut arrêter la boucle for lorsqu’on atteint la valeur de décalage de 32, puis faire les réductions/additions restantes manuellement avec des variables volatiles. C’est-ce que nous ferons dans la tâche 2.

Tâche 2:

Voici nos analyses du code avec nvprof sur google colab:

```
[8] !nvprof ./reduction

==1882== NVPROF is profiling process 1882, command: ./reduction
Kernel Execution Time: 0.189952 ms
Sum: 8256
==1882== Profiling application: ./reduction
==1882== Profiling result:
Type      Time(%)      Time      Calls      Avg      Min      Max      Name
GPU activities: 52.94%  3.7440us    1  3.7440us  3.7440us  3.7440us  void parallelReduceUnrolled<int, Add<int>>(int*, int*, int, int, int)
              30.32%  2.1440us    1  2.1440us  2.1440us  2.1440us  [CUDA memcpy DtoH]
              16.74%  1.1840us    1  1.1840us  1.1840us  1.1840us  [CUDA memcpy HtoD]
API calls:    99.68%  159.45ms    2  79.725ms  3.7750us  159.45ms  cudaMalloc
              0.11%  183.20us    1  183.20us  183.20us  183.20us  cudaLaunchKernel
              0.08%  126.65us   114  1.1100us   133ns   50.686us  cuDeviceGetAttribute
              0.06%  98.027us    2  49.013us  7.4950us  90.532us  cudaFree
              0.03%  42.158us    2  21.079us  18.922us  23.236us  cudaMemcpy
              0.01%  13.175us    2  6.5870us   688ns   12.487us  cudaEventCreate
              0.01%  10.785us    2  5.3920us  3.3800us  7.4050us  cudaEventRecord
              0.01%  10.463us    1  10.463us  10.463us  10.463us  cuDeviceGetName
              0.00%  6.4360us    1  6.4360us  6.4360us  6.4360us  cudaEventSynchronize
              0.00%  5.1180us    1  5.1180us  5.1180us  5.1180us  cuDeviceGetPCIBusId
              0.00%  4.8220us    1  4.8220us  4.8220us  4.8220us  cuDeviceTotalMem
              0.00%  2.0510us    1  2.0510us  2.0510us  2.0510us  cudaEventElapsedTime
              0.00%  1.8070us    3    602ns   293ns   1.2200us  cuDeviceGetCount
              0.00%  1.2860us    2    643ns   161ns   1.1250us  cuDeviceGet
              0.00%  538ns      1    538ns   538ns   538ns    cuModuleGetLoadingMode
              0.00%  225ns      1    225ns   225ns   225ns    cuDeviceGetUuid
```

Figure 3: Exécution de nvprof sur la tâche 2

On remarque que les optimisations faites pour la réduction de divergence de warp, le déroulement de boucle ont permis une amélioration du temps de quelques 0.1ms. On est passé de 0.209696ms à 0.189952ms. Il faut aussi réaliser que notre code maintenant soutient l’utilisation de variables de type différents contrairement à la tâche 1.

Tâche 3:

Attention, pour ce code, il faut compiler sur colab avec cette commande pour permettre l’exécution imbriqué **!nvcc -rdc=true -o task3_4 task3_4.cu**

```

0s !nvprof ./reduction

==9509== NVPROF is profiling process 9509, command: ./reduction
Kernel Execution Time: 13.073216 ms
Somme Totale: 8256
==9509== Profiling application: ./reduction
==9509== Profiling result:

```

Type	Time(%)	Time	Calls	Avg	Min	Max	Name
GPU activities:	92.29%	39.072us	1	39.072us	39.072us	39.072us	void nestedReduction<int>(int*, int*, int)
	4.91%	2.0800us	1	2.0800us	2.0800us	2.0800us	[CUDA memcpy DtoH]
	2.80%	1.1840us	1	1.1840us	1.1840us	1.1840us	[CUDA memcpy HtoD]
API calls:	87.06%	90.410ms	2	45.205ms	3.9380us	90.406ms	cudaMalloc
	12.52%	13.006ms	1	13.006ms	13.006ms	13.006ms	cudaLaunchKernel
	0.14%	141.75us	2	70.873us	10.729us	131.02us	cudaFree
	0.13%	133.17us	114	1.1680us	136ns	52.857us	cuDeviceGetAttribute
	0.06%	62.284us	1	62.284us	62.284us	62.284us	cudaEventSynchronize
	0.05%	47.737us	2	23.868us	21.035us	26.702us	cudaMemcpy
	0.01%	13.814us	2	6.9070us	5.7410us	8.0730us	cudaEventRecord
	0.01%	11.529us	1	11.529us	11.529us	11.529us	cuDeviceGetName
	0.01%	10.263us	2	5.1310us	637ns	9.6260us	cudaEventCreate
	0.01%	5.6750us	1	5.6750us	5.6750us	5.6750us	cuDeviceGetPCIBusId
	0.00%	4.5670us	1	4.5670us	4.5670us	4.5670us	cuDeviceTotalMem
	0.00%	1.8470us	1	1.8470us	1.8470us	1.8470us	cudaEventElapsedTime
	0.00%	1.5810us	3	527ns	205ns	1.0840us	cuDeviceGetCount
	0.00%	1.0940us	2	547ns	170ns	924ns	cuDeviceGet
	0.00%	637ns	1	637ns	637ns	637ns	cuModuleGetLoadingMode
	0.00%	235ns	1	235ns	235ns	235ns	cuDeviceGetUuid

Figure 4: Exécution de nvprof sur la tâche 3

On remarque que l'exécution a pris plus de temps malheureusement. C'est parce qu'avant le code ne faisait pas la somme des valeurs finale de chaque bloc, ceci était fait dans le main de manière séquentielle. Ici, on a voulu optimiser cette somme finale avec un appel imbriqué.

Tâche 4:

Attention, pour ce code, il faut compiler sur colab avec cette commande pour permettre l'exécution imbriqué **!nvcc -rdc=true -o Task4 Task4.cu**

```

nvprof --print-gpu-trace ./Task4

==11013== NVPROF is profiling process 11013, command: ./Task4
Kernel Execution Time: 13.790368 ms
Somme Totale: 524800
==11013== Profiling application: ./Task4
==11013== Profiling result:
   Start  Duration      Grid Size    Block Size    Regs*    SSMem*    DSMem*    Size  Throughput  SrcMemType  DstMemType    Device  Context  Stream  Name
322.43ms  1.4880us          (2 1 1)        (256 1 1)        32         0B    1.0000KB    4.0000KB  2.7093GB/s  Pageable     Device     Tesla T4 (0)    1         7  [CUDA memcpy HtoD]
336.19ms  39.391us          (2 1 1)        (256 1 1)        32         0B    1.0000KB    8B      3.7253MB/s  Device      Pageable     Tesla T4 (0)    1         7  void nestedReduction<int*>(int*, int*, int)
336.31ms  2.0480us          (2 1 1)        (256 1 1)        32         0B    1.0000KB    8B      3.7253MB/s  Device      Pageable     Tesla T4 (0)    1         7  [CUDA memcpy HtoD]

Regs: Number of registers used per CUDA thread. This number includes registers used internally by the CUDA driver and/or tools and can be more than what the compiler shows.
SSMem: Static shared memory allocated per CUDA block.
DSMem: Dynamic shared memory allocated per CUDA block.
SrcMemType: The type of source memory accessed by memory operation/copy
DstMemType: The type of destination memory accessed by memory operation/copy

nvprof --metrics warp_execution_efficiency ./Task4

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

==1831== NVPROF is profiling process 1831, command: ./Task4
Kernel Execution Time: 12.547168 ms
Somme Totale: 524800
==1831== Profiling application: ./Task4
==1831== Profiling result:
No events/metrics were profiled.

[15] nvprof ./Task4

==16381== NVPROF is profiling process 16381, command: ./Task4
Kernel Execution Time: 13.387200 ms
Somme Totale: 524800
==16381== Profiling application: ./Task4
==16381== Profiling result:
   Type  Time(%)    Time    Calls    Avg      Min      Max  Name
GPU activities:  92.00%  39.392us      1  39.392us  39.392us  39.392us  void nestedReduction<int*>(int*, int*, int)
               4.78%  2.0480us      1  2.0480us  2.0480us  2.0480us  [CUDA memcpy DtoH]
               3.21%  1.3760us      1  1.3760us  1.3760us  1.3760us  [CUDA memcpy HtoD]
API calls:    93.71%  205.29ms      2  102.64ms  7.0050us  205.28ms  cudaMalloc
               6.08%  13.322ms      1  13.322ms  13.322ms  13.322ms  cudaLaunchKernel
               0.07%  163.56us     114  1.4340us    138ns   53.398us  cuDeviceGetAttribute
               0.06%  125.31us      2  62.652us  10.481us  114.82us  cudaFree
               0.03%  61.511us      2  30.755us  27.368us  34.143us  cudaMemcpy
               0.03%  60.374us      1  60.374us  60.374us  60.374us  cudaEventSynchronize

```

Figure 5: Exécution de nvprof sur la tâche 4

On remarque que l'exécution a pris autour du même montant de temps que pour la tâche 3, même avec l'augmentation de la taille du bloc. C'est parce que nous avons pu identifier et éliminer la plupart des goulets d'étranglements pour améliorer et optimiser le code de la tâche 3. Ici, on a voulu optimiser les performances de votre programme CUDA en identifiant et en traitant systématiquement les goulets d'étranglement, en améliorant l'utilisation des ressources et en réduisant la latence. Ce processus vous permet d'apprendre à améliorer les applications CUDA afin d'obtenir une meilleure efficacité et une exécution plus rapide sur le GPU.

Résultats et Validation

Tâche 1: Le code parallèle de réduction de la somme calcule correctement la somme d'un tableau d'entiers avec CUDA, validé en comparant le résultat du GPU avec un calcul séquentiel du CPU, qui donne ici 8256. Le profilage avec nvprof montre des gains de performance grâce à une divergence minimale et une mémoire optimisée via l'utilisation de la mémoire partagée, réduisant les accès globaux. Des métriques comme l'efficacité des warps, le temps du noyau, et le débit mémoire valident une utilisation efficace des ressources. La divergence de warp est minime, mais certaines améliorations peuvent la réduire davantage, en particulier lorsque la taille

du tableau n'est pas un multiple parfait de la taille du bloc, ce qui entraîne des vérifications des conditions aux limites. Pour remédier à ce problème, l'utilisation d'une taille de bloc puissance 2 et d'une entrée rembourrée permet de supprimer les branches conditionnelles. Le déroulement des boucles est une autre amélioration efficace, car il permet à chaque thread d'effectuer plusieurs étapes de réduction en une seule itération, ce qui minimise les vérifications conditionnelles et frais de synchronisation.

Tâche 2: Le code de réduction optimisé pour Tâche 2 améliore avec succès les performances et minimise la divergence de warp, aussi met en œuvre le déroulage de boucle et l'utilisation de template. Cette version du noyau de réduction permet de dérouler les boucles par un facteur de quatre, ce qui réduit le nombre d'itérations et minimise les frais de synchronisation. Les templates ajoutent de la flexibilité, permettant des réductions par addition et par multiplication avec des variables de type différents (int, float, double, etc), ce qui rend l'algorithme plus adaptable. Le profilage confirme la réduction de la divergence de la warp et l'amélioration de l'efficacité de l'exécution grâce à la réduction du nombre de branchements et de vérifications conditionnelles, en particulier lors de l'utilisation de la mémoire partagée. Ensemble, ces optimisations améliorent l'évolutivité, l'efficacité et l'utilisation des ressources, validant le fait que le noyau utilise efficacement le déroulage et la conception modulaire pour optimiser les performances des différentes opérations.

Tâche 3: Le code d'optimisation avancée et parallélisme dynamique de la tâche 3 vise à utiliser l'exécution imbriquée et le parallélisme dynamique dans une réduction parallèle pour augmenter la flexibilité et potentiellement améliorer les performances. Nous avons commencé par implémenter une réduction parallèle avec exécution imbriquée à l'aide de la parallélisation dynamique. Chaque niveau de réduction lance des noyaux en fonction des besoins, sans attendre que l'unité centrale gère chaque étape. Le processus doit continuer à réduire les résultats de manière imbriquée et récursive, en donnant au GPU le contrôle des étapes suivantes de la réduction de manière dynamique. Ensuite, nous avons évalué l'impact potentiel sur les performances du parallélisme dynamique en le comparant à une implémentation de réduction traditionnelle. Le parallélisme dynamique dans l'algorithme de réduction imbriquée démontre une amélioration des performances par rapport aux lancements de noyaux traditionnels gérés par l'hôte. En permettant au GPU de lancer de manière autonome les noyaux suivants, nous avons réduit la latence et les frais généraux associés aux synchronisations multiples entre l'hôte et le GPU. Cette approche a rationalisé le processus de réduction, permettant une convergence plus rapide vers le résultat final. L'analyse de l'évolutivité montre en outre que la méthode d'exécution imbriquée utilise mieux les ressources du GPU lorsque la taille du problème augmente, en maintenant une exécution parallèle efficace avec un minimum de goulots d'étranglement. Ces résultats valident l'utilisation du parallélisme dynamique pour une réduction optimisée dans des

environnements à parallélisme élevé, en particulier pour les données à grande échelle où la minimisation de l'intervention de l'unité centrale améliore les performances. Pour l'analyse de la scalabilité du programme, on a découvert que l'algorithme peut traiter des données de plus grande taille avec des gains de performance relativement constants au fur et à mesure de l'ajout de threads, jusqu'à un point où les ressources deviennent saturées.

Tâche 4: Le code d'optimisation des profils de la tâche 4 est un programme optimisé pour CUDA qui démontre des améliorations mesurables en termes de performances et d'efficacité, ainsi qu'un rapport détaillé documentant ces améliorations. Nous avons modifié le code de la tâche 3 pour inclure des ajustements aux tailles de blocs, aux modèles d'accès à la mémoire, à l'utilisation d'opérations asynchrones ou à d'autres techniques pour maximiser l'occupation de la chaîne et minimiser la latence. Nous avons utilisé les fonctions `--metrics` et `--print-gpu-trace` avec `nvprof` pour approfondir les domaines où l'on perd du temps. Cela implique l'analyse des temps d'exécution du noyau, des transferts de mémoire et d'autres mesures de performances pour identifier les goulets d'étranglements. En observant ces métriques, on pouvait identifier et éliminer les goulets d'étranglements pour ensuite, de façon essai-erreur, maximiser l'occupation des warps et minimiser la latence.

Problèmes Rencontrés

Pour ce laboratoire, tout s'est bien passé et nous avons respecté toutes les contraintes requises. Cependant, nous avons rencontré quelques difficultés en cours de route. L'un des principaux problèmes concernait la plateforme de développement CUDA. Encore une fois, la plupart des membres de l'équipe ne possèdent pas de GPU NVIDIA et ne pouvaient donc pas utiliser CUDA pour le laboratoire. Puis, même les étudiants ayant une carte NVIDIA ont eu de la difficulté à `nvprof` car celui-ci est déprécié/legacy sur les nouvelles cartes NVIDIA. Finalement, Google Colab nous a permis d'utiliser `nvprof`. Un autre problème que nous avons rencontré pour la tâche 1, la somme des éléments du tableau devrait donner 55, mais on reçoit 40. On croit que ceci est à cause du `grid size`, puisque initialement, nous avons à prendre la somme de 1 à 10. Nous avons découvert qu'il faut utiliser une valeur qui est multiple de 4 pour que ça fonctionne. Donc, nous avons augmenté notre taille de 10 à 16 pour maintenant faire la somme de valeur $i+1$, qui nous donne la bonne réponse de 136.

Lors de la réalisation de la tâche 3, j'ai rencontré un problème de compilation avec la commande suivante : `!nvcc task3.cu -o task3`

Cette commande ne fonctionnait pas correctement, ce qui m'a empêché de générer l'exécutable souhaité. Après analyse, j'ai découvert que l'option `-rdc=true` était nécessaire pour activer la compilation séparée, ce qui est requis dans certains cas pour les projets CUDA. J'ai donc ajusté la commande de compilation de la manière suivante : `!nvcc -rdc=true -o task3 task3.cu`

Cette modification a résolu le problème, permettant une compilation réussie et l'exécution de mon code.

De plus, nous avons rencontré des difficultés spécifiques avec la tâche 4. Malgré nos efforts pour interpréter les instructions, nous avons eu du mal à comprendre certains aspects de cette tâche. Nous avons tenté de l'implémenter en fonction de notre compréhension, mais sans certitude quant aux attentes exactes. Cette incompréhension a pu impacter la qualité de notre solution pour cette partie.

Conclusion

Dans ce laboratoire, nous avons exploré avec succès des concepts avancés de programmation parallèle en utilisant CUDA C, en nous concentrant sur l'optimisation des algorithmes de réduction pour les architectures GPU. Des stratégies d'optimisation clés telles que la gestion des warps, le déroulement des boucles (loop unrolling) et l'utilisation de templates ont été mises en œuvre pour améliorer l'efficacité et la modularité du code. En profilant le code avec 'nvprof', nous avons identifié les goulets d'étranglement et réduit la divergence des threads, ce qui a amélioré l'utilisation des ressources et les performances.

L'intégration du parallélisme dynamique et de l'exécution imbriquée a mis en évidence le potentiel d'exposer davantage de parallélisme, avec un impact significatif sur la scalabilité du programme. Notre analyse a confirmé que la réduction de la divergence des threads et la maximisation de l'occupation des warps ont entraîné des gains de performance notables, en particulier lors de la comparaison de l'algorithme de réduction avec et sans exécution imbriquée.

Grâce à un profilage itératif et à des optimisations ciblées, nous avons appliqué avec succès des techniques de masquage de latence et de partitionnement des ressources, maximisant ainsi le débit du GPU. Ces résultats soulignent l'importance de stratégies d'optimisation adaptées pour exploiter pleinement le potentiel du calcul parallèle avec CUDA. Ce laboratoire se conclut par une compréhension approfondie de l'architecture GPU et de l'optimisation des performances, et nous fournit une bonne connaissance de ce sujet pour de futurs laboratoires.

Distribution de tâches

Nom	Tâche code	Tâche Rapport
Adam Taktek	Tâche 1 : code qui réalise réduction parallèle, exécution avec <i>nvprof</i> . Tâche 2 : amélioration du code en utilisant les techniques de unrolling des boucles. Utilisation des fonctions templates pour généralisation du code de réduction, permettant la multiplication/addition avec un choix varié de type de variable. Élimination de la divergence de warp.	Analyse du problème Conception de solution,(tâche 1-4) Procédure de résolution et algorithme, Évaluation d'implémentation, (pour tache 1 et 2)
Maro Abdine	Tâche 1: Introduction au modèle d'exécution CUDA et analyse de la performance avec <i>nvprof</i> Tâche 3 : ajout de l'exécution imbriqué. Évaluation de la performance avant et après	Problème rencontré, Référence
Christopher Krayem	Vérification générale et tests finaux afin d'éviter à avoir des erreurs.	Introduction Équipements & Composants Utilisés Conclusion Mise en page du rapport
Simon	Travailler un peu sur chaque tâche	Résultat et validation, Problem rencontrer
Reda	Tâche 4: Optimisation et Analyse des Performances d'un Noyau CUDA de Réduction	Procédure de résolution et algorithme (tâches 3 et 4)
Hakim	Tâche 3: Implémentation du noyau parent qui appelle les noyaux enfants, mais sans succès.	Section de problèmes rencontrés.

Table 1: Distribution de tâche

Références

- “Lecture_3_f”,MohamedAliIbrahim, 25 Sept. 2024
- “DynamicParallelism”,MohamedAliIbrahim, 2 Oct. 2024

- “ReduceComparison”,MohamedAlilbrahim, 2 Oct. 2024

Appendice

[LAB2](#) Onedrive

[Github](#) GitHub