

M2 - Architecture et Programmation d'accélérateurs Matériels.

(APM 2022-2023)



TP1

Optimisation des accès mémoire
Utilisation de la mémoire partagée

julien.jaeger@cea.fr

adrien.roussel@cea.fr

L'objectif de ce TP est de comprendre les différentes mémoires existantes d'un GPU et leur utilisation sur un noyau de calcul classique d'algèbre linéaire : le produit matriciel. Nous allons voir à travers cet exemple comment utiliser la mémoire globale, la mémoire partagée et la mémoire unifiée.

Les objectifs de ce TP sont :

- Maitriser la chaîne de compilation CUDA
- Compréhension du concept de global memory coalescing GPU,
- Optimisation des performances avec l'utilisation de la mémoire partagée.

Ce TP est à faire sur deux séances.

I Chaîne de compilation

Pour observer certaines étapes du processus de compilation, compiler le fichier *tp1_check.cu* avec la commande :

nvcc -keep -keep-dir CUDAIMG -c tp1_check.cu

(il faudra avoir créé le répertoire CUDAIMG avant de lancer la commande).

Des fichiers intermédiaires sont créés dans le répertoire CUDAIMG.

Q.1: Quel fichier correspond au PTX ? Lequel correspond au binaire CUDA ?

Q.2: Quelles parties des sources de *tp1_check.cu* ont été conservées pour être compilées en PTX ? **Remarque :** Vous pouvez lire ce paragraphe de la documentation CUDA pour en apprendre plus sur la chaîne de compilation CUDA.

Q.3: À quoi correspond le suffixe ***_30*** des fichiers créés ?

II Multiplication matricielle

Le but de cette partie est d'implémenter une première version de multiplication de matrices sur GPU. La mémoire globale est accessible depuis n'importe quel thread d'une grille sur un GPU. C'est la mémoire qui est allouée/libérée grâce aux fonctions `cudaMalloc` et `cudaFree`. Nous allons dans un premier temps utiliser cette mémoire pour écrire un premier kernel de produit matriciel.

La multiplication de deux matrices est un problème qui se prête bien au système de fonctionnement du GPU. Pour vous aider, un code séquentiel écrit en C est à votre disposition, il s'agit du fichier `dgemm.c`. À partir d'ici, veillez à compléter ce programme afin de déporter les calculs de la multiplication des deux matrices sur un GPU. N'oubliez pas de changer l'extension du fichier (*i.e.* `.c` en `.cu`).

Soient A , B et C trois matrices carrées de tailles $N \times N$, la multiplication de la matrice A par la matrice B stockée dans C peut se définir par l'algorithme suivant :

Listing 1 – Multiplication de matrice CPU en C

```
1 float A[N*N];
2 float B[N*N];
3 float C[N*N];
4
5 for (int i = 0; i < N; i++)
6 {
7     for (int j = 0; j < N; j++)
8     {
9         float tmp = 0.0f
10        for (int k = 0; k < N; k++)
11        {
12            tmp += A[i * N + k] * B[k * N + j];
13        }
14        C[i * N + j] = tmp;
15    }
16 }
```

1 Allocation et transfert de données

Q.4: Lisez l'introduction de la page Wikipedia "Row- and -column major order". Quelle représentation avons nous choisi dans l'extrait de code précédent ? Sauf mention contraire, nous utiliserons cette représentation dans la suite du TP.

Q.5: Dans un premier temps, à l'aide de la fonction `cudaMalloc`, allouer trois vecteurs de tailles $N \times N$ sur le GPU : deux vecteurs en entrée pour les opérandes de la multiplication, et un troisième pour stocker le résultat final.

Ici, nous représentons une matrice sous la forme d'un seul vecteur de taille $N \times N$ comme dans le listing 1. L'intérêt est de stocker les matrices de manière contigüe en mémoire et de ne pas avoir à faire des transferts de tableaux à outrance (par exemple un tableau par ligne si l'on stocke une matrice sous forme d'un vecteur de vecteurs).

Q.6: Les données étant maintenant allouées, transférez les données initialisées par l'hôte (*i.e.* `Host`) vers l'accélérateur (*i.e.* `Device`) en utilisant la fonction `cudaMemcpy`.

2 Configuration du kernel

Q.7: Avant de déporter du calcul sur l'accélérateur, il faut configurer la grille sur laquelle vont s'exécuter l'ensemble des threads CUDA. À l'aide des structures de type `dim3`, configurez une grille contenant des blocs 2D de taille 32x32 (*i.e.* 1024 threads par bloc). Cette grille devra contenir suffisamment de threads afin que chacun d'entre eux puisse être en charge du calcul d'un élément de la matrice résultat. Pour plus de facilités, prenez une taille de matrice N qui soit multiple de 32.

Q.8: L'heure est venue de faire s'exécuter le premier kernel sur un accélérateur de calcul. Écrire un premier kernel, pour l'instant vierge, sur la grille préalablement configurée à la question précédente.

Q.9: En l'état, le noyau s'exécute sur l'accélérateur de calcul mais ne retourne aucun résultat sur la partie hôte. A la suite de l'exécution du kernel, veiller à transférer le résultat qui nous intéresse pour le calcul de la multiplication matricielle.

3 Écriture du kernel

Q.10: Dans le noyau de calcul, à partir des dimensions et des indices relatifs à la grille, calculez l'indice de la ligne et de la colonne de l'élément de C qu'il faudra manipuler. Utilisez la dimension x pour les lignes et la dimension y pour les colonnes.

Q.11: Pour un thread donné, effectuez le calcul de la case de C dans le noyau exécuté par le GPU.

4 Analyse des performances

Q.12: Maintenant que le calcul de multiplication matricielle fonctionne, il nous faut analyser les performances de notre application. Le SDK CUDA fournit des fonctions permettant de calculer le temps total passé dans un noyau de calcul. Utiliser ces fonctions et regardez le temps mis par l'application pour calculer le produit de deux matrices et le transfert du résultat sur la partie hôte.

Comparez ces performances au noyau s'exécutant uniquement sur CPU, de manière séquentiel.

Indice : la fonction à utiliser pour calculer le temps écoulé entre un point A et un point B est la fonction `cudaEventElapsedTime`.

Q.13: Dans cette question, nous considérons la multiplication de deux matrices carrées 4092×4092 de flottants simple précision (FP32). Nous souhaitons estimer une borne inférieure du temps nécessaire pour effectuer la multiplication sur GPU des matrices.

Estimez

- le nombre d'opérations flottantes
- le volume de données minimal V_r à lire (en MB)
- le volume de données minimal V_w à écrire (en MB)

nécessaire pour calculer le produit matricielle.

Quelle est la bande passante mémoire maximale théorique de votre GPU ? Quelle est le débit d'opérations flottantes maximal théorique de votre GPU ? (cf TP0). En utilisant ces deux données et en considérant une implémentation idéale qui ne transfert que le volume de données minimal $V = V_r + V_w$ depuis ou vers la mémoire globale, calculez

- le temps minimal t_{data} pour effectuer les transferts de données
- le temps minimal $t_{compute}$ pour effectuer les calculs

Calculez $\rho = \frac{t_{compute}}{t_{data}}$. Commentez.

Q.14: Calculez le débit d'opérations flottantes de votre kernel. Comparez le au débit d'opération flottante maximal de votre GPU. Commentez.

Q.15: L'objectif de cette question est de décrire à la main le pattern d'accès mémoire du kernel naïf. Dans le kernel naïf, quel volume de données doit charger un thread pour calculer une valeur de C ? Est-ce que les threads accèdent à des données contigües en mémoire ?

En considérant le pire cas où chaque chargement de données génère un cache miss, estimatez le traffic mémoire requis pour effectuer le produit matricielle de 2 matrices 4092×4092 .

III Optimisation du pattern d'accès à la mémoire globale

Dans cette section, nous allons optimiser le kernel de la partie précédente en changeant le pattern d'accès mémoire de manière à ce que le GPU puisse effectuer des accès mémoires "coalescés" ("global memory coalescing" en anglais).

1 Concept de warp

Nous vous rappelons ici le concept de warp. Lors de l'exécution d'un kernel, les blocs de threads sont assignés aux SM. Au sein d'un même bloc, les threads sont regroupés en paquets de 32 threads appelé "warps". Les warps sont assignés à un warp scheduler (il y en plusieurs par SM) qui se charge d'exécuter les instructions sur les coeurs du GPU.

Le groupement des threads en warp se fait par *threadId* global consécutif. Par exemple, pour une grille 2D de blocs 2D, l'indice global *threadId* est (cf TD1) :

$$\begin{aligned} \textit{threadId} = & (\textit{blockIdx.y} \times \textit{gridDim.x} + \textit{blockIdx.x}) \times \textit{blockDim.x} * \textit{blockDim.y} \\ & + \textit{threadIdx.y} \times \textit{blockDim.x} + \textit{threadIdx.x} \end{aligned}$$

Les threads de *threadId* consécutif sont regroupés en warp comme illustré sur la figure suivante.

2 Global memory coalescing

Les accès mémoire séquentiel des threads d'un même warp peuvent être regroupés et exécutés comme un seul accès par le hardware. Cela est appelé le "global memory coalescing".

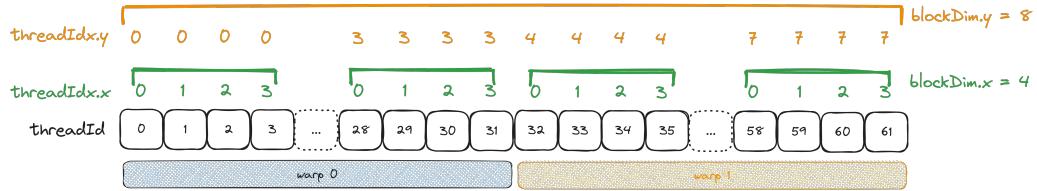


FIGURE 1 – Regroupement des threads en warp. $\text{blockIdx} = (0, 0)$

Le GPU supporte des accès mémoire de taille 32B, 64B et 128B.

Si les accès mémoire des threads d'un même warp sont alignés et contigus en mémoire, alors le GPU sera en mesure de regrouper plusieurs accès en une seule transaction.

Sinon, le GPU effectuera autant d'accès mémoire de taille 32B que nécessaire. Mais cela n'optimise pas l'utilisation de la bande passante mémoire.

La figure suivante illustre un exemple où les accès mémoire contigus des threads d'un même warp sont regroupés, permettant à chaque warp de faire 4 accès mémoire de 32B en utilisant seulement 1 load de 128B.

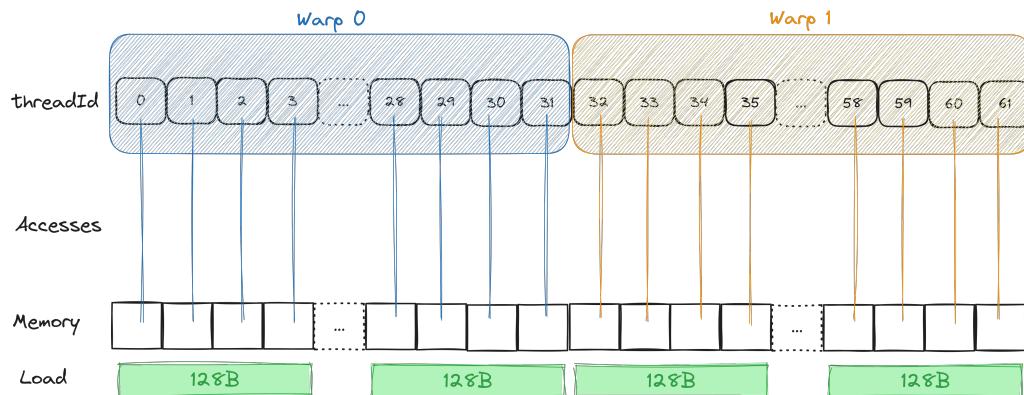


FIGURE 2 – Regroupement des accès à la mémoire globale

3 Accès mémoire du kernel naïf

Vu la représentation de matrices que nous avons choisie, une ligne constitue un ensemble de données contigües en mémoire et une colonne une ensemble de données non contigües en mémoire.

La figure suivante illustre le pattern d'accès mémoire des threads d'un même warp au sein des 3 matrices.

Au sein de A, à chaque tour de boucle, les threads d'un même warp accèdent à une portion de ligne de A. Ces accès ne sont pas contigus et ne peuvent être coalescés.

Au sein de B, à chaque tour de boucle, tous les threads accèdent à la même donnée qui peut être broadcasté à tous les threads du warp.

Au sein de C, les données écrites ne sont pas contigües en mémoire.

Q.16: Nous allons changer le pattern d'accès mémoire du kernel de manière à bénéficier du global memory coalescing. Cela peut être effectué en changeant la valeur de C calculé par un thread. Nous souhaitons que les threads d'un même

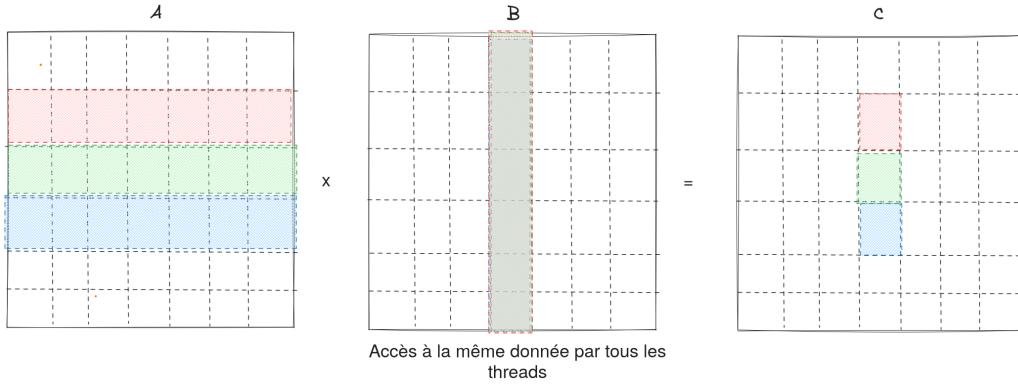


FIGURE 3 – Accès mémoire du kernel naïf

warp s’occupent de calculer les valeurs de C situés sur la même ligne. De cette manière les threads d’un même warp liront à chaque tour de boucle

- la même valeur dans A
 - des valeurs contigües dans B (car situées sur la même ligne)
- Le pattern d'accès mémoire visé est illustré sur la figure suivante.

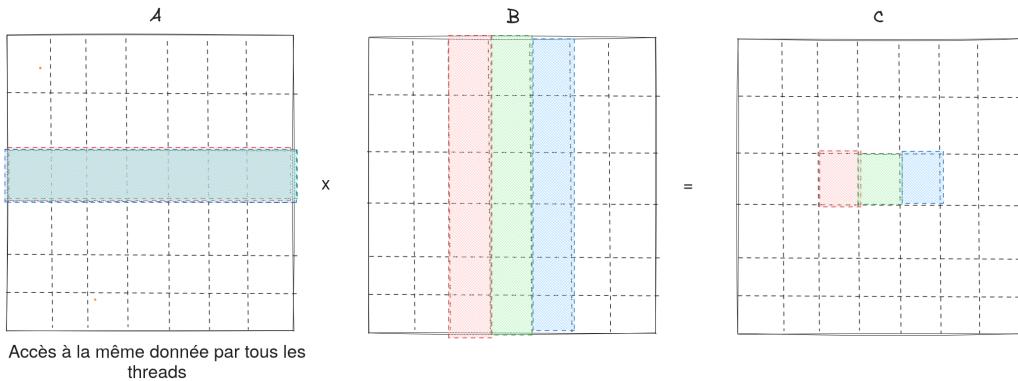


FIGURE 4 – Pattern d'accès mémoire visé

Changez la valeur de C calculé par un thread de manière à obtenir le pattern d'accès mémoire visé.

IV Utilisation de la mémoire partagée

La mémoire partagée avec CUDA favorise la coopération entre les threads d'un même bloc. Lorsque plusieurs threads d'un bloc utilisent les mêmes données depuis la mémoire globale, ces données peuvent être chargées dans un espace mémoire partagé entre tous les threads d'un même bloc : c'est la notion de mémoire partagée avec CUDA.

L'objectif ici est d'améliorer l'implémentation que l'on a faite de la multiplication matricielle en utilisant la mémoire partagée. La configuration de la grille ne change pas (blocs 2D de 32×32 threads ou blocs 1D de 32×32 threads), uniquement le noyau de calcul va changer.

Q.17: Nous allons maintenant travailler sur des blocs de données, que chaque thread va charger depuis la mémoire globale. Ces accès intempestifs proviennent du

chargement des deux matrices d'entrée. Déclarer deux blocs de données partagées à l'intérieur du noyau. Ces blocs 2D auront les mêmes dimensions qu'un bloc de la grille configurée pour l'exécution du noyau CUDA.

Q.18: Dans un second temps, nous allons identifier les données à charger en mémoire partagée pour favoriser la coopération entre les threads d'un même bloc. Chaque thread d'un bloc va charger en mémoire partagée une valeur des deux matrices d'entrée. Comme nous procémons par bloc, écrivez une boucle qui permette, à chaque itération, de charger un élément par thread de chaque matrice d'entrée qui permette de couvrir le chargement de l'ensemble de la matrice.

Q.19: Après avoir chargé les données, nous pouvons maintenant passer à la partie calculatoire. A l'aide des données en mémoire partagée, effectuer le produit matriciel sur le bloc courant et accumuler le résultat dans une variable locale. Stockez la valeur finale dans C à la case correspondante.

Q.20: Exécuter maintenant votre programme et vérifier le résultat par rapport à la version d'avant. À moins que vous n'ayez déjà paré cette éventualité, le résultat n'est pas correct. Pourquoi ? Comment corriger ce problème ?

Q.21: Passons maintenant à l'évaluation des performances. Reportez le temps d'exécution de notre noyau optimisé par rapport au programme non optimisé du TP précédent. Quel est le gain obtenu avec cette nouvelle version ?