

Introduction à la Programmation GPU

TP3 : Map-Reduce

Majeure SSIE - EPITA - 2024

Ce TP a pour objectif d'implémenter le **produit scalaire** entre deux vecteurs sur GPU selon le modèle d'algorithme **Map-Reduce** (ou Transform-Reduce). Le produit scalaire entre deux vecteurs $x, y \in \mathbb{R}^n$ est un réel $z \in \mathbb{R}$ défini par

$$z = \sum_{i=0}^{n-1} x_i y_i.$$

L'opération Map-Reduce illustrée en Figure 3 consiste à

1. transformer chaque élément indépendamment selon le modèle **Map**
2. réduire les éléments transformés en une unique valeur selon le modèle **Reduce**

Le produit scalaire se décompose donc en deux étapes : multiplier chaque paire d'éléments $x_i y_i$, puis sommer tous ces résultats intermédiaires. Sur GPU, un unique noyau CUDA effectue les deux opérations Map et Reduce. Les résultats intermédiaires sont stockés dans la **mémoire partagée** du GPU. Le programme est dit "hétérogène" car la réduction sur le *device* n'est que partielle, le *host* terminant le calcul de la somme finale.

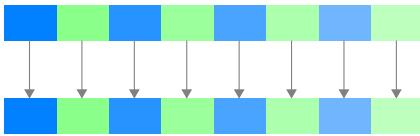


Figure 1: Map

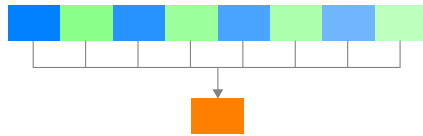


Figure 2: Reduce

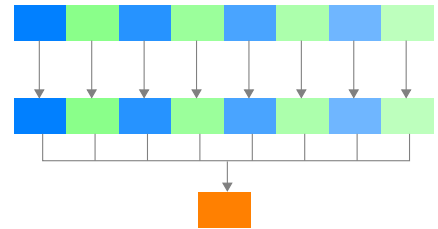


Figure 3: Map-Reduce

Dans chaque exercice, les tableaux ont une taille de **N=1e6** entiers, et le lancement du noyau CUDA est configuré avec **B=128 blocs** de **T=32 threads** par bloc, impliquant une "grid-stride loop" dans l'opération Map (cf TP1).

Mémoire partagée (shared memory)

La mémoire partagée d'un GPU est un espace mémoire accessible par tous les *threads* d'un même *bloc* d'exécution CUDA. Elle se trouve directement sur la puce du GPU ("on-chip") ce qui rend relativement **rapide** l'accès en lecture et écriture depuis un noyau CUDA (comparé aux accès à la mémoire globale).

Les variables et les tableaux statiques déclarés dans un noyau avec le mot clé `__shared__` résident dans la mémoire partagée. La mémoire partagée a comme portée (*scope*) l'exécution du noyau, elle n'est donc **plus accessible après la fin du noyau**. Cet espace mémoire peut être considéré comme un cache programmable dans lequel stocker des résultats intermédiaires ou des données à partager entre *thread* du *bloc*.

1. Produit scalaire simple

Dans cette première version, le noyau CUDA calcule **B** produits scalaires partiels. Dans chaque *bloc*, les **T threads** effectuent une *grid-stride-loop* pour calculer des produits scalaires partiels, puis stockent les résultats dans un *buffer* de la **mémoire partagée** à l'indice de chaque *thread*.

Dans un second temps, après la **synchronisation** de tous les *threads* d'un *bloc*, un unique *thread* calcule la somme des *T* valeurs du *buffer* partagé, puis écrit ce résultat dans *dz* à l'indice du *bloc* correspondant.

Synchronisation au niveau des blocs

La fonction CUDA `__syncthreads()` est une **barrière** qui synchronise les *threads* d'un même *bloc*. Tant que tous les *threads* d'un *bloc* n'ont pas atteint cette ligne de code, les autres *threads* du *bloc* attendent avant de continuer leur exécution. Cette fonction **évite les accès concurrents** (*race conditions*), en s'assurant par exemple que tous les *thread* aient bien écrit dans la mémoire partagée avant de la lire.

→ **Step 01.** Dans le fichier `ex1.cu`, implémenter le noyau CUDA `dot` qui

1. calcule un produit scalaire partiel entre paires d'éléments de `dx` et `dy` à des indices séparés par une *stride*
2. stocke le produit scalaire partiel dans un *buffer* de taille *T* qui réside dans la *shared memory*
3. synchronise tous les *threads* du *bloc* courant
4. si le *thread* courant est le premier *thread* du *bloc* courant, calcule la somme des *T* valeurs du *buffer* partagé et écrit le résultat dans `dz`

→ **Step 02.** Compléter la fonction `main` qui alloue la mémoire nécessaire sur le *device*, transfère les données du *host* au *device*, puis lance le noyau `dot`.

→ **Step 03.** Finaliser la fonction `main` afin de rapatrier les *B* produits scalaires temporaires `dz` sur le *host* et calculer le résultat final. Libérer la mémoire allouée sur le *device* et sur le *host*. Compiler avec la commande `nvcc ex1.cu -o ex1` et exécuter `./ex1`.

2. Produit scalaire optimisé

Un algorithme itératif illustré en Figure 4 existe afin de paralléliser la réduction effectuée par le 1^{er} *thread* du *bloc* à la fin du noyau CUDA précédent. Cette approche exploite mieux le parallélisme du GPU en impliquant plus de *threads* et atteint en général de meilleures performances.

→ **Step 04.** Dans le fichier `ex2.cu`, implémenter le noyau CUDA `dot` qui optimise la réduction. Les points 1, 2 et 3 du noyau CUDA de l'exercice précédent sont similaires. Seul le point 4 diffère car la somme (réduction) est parallèle. Notons que la taille des *blocs* `T=32` est bien une puissance de 2.

→ **Step 05.** Compléter la fonction `main` de manière similaire à l'exercice précédent. Compiler `nvcc ex2.cu -o ex2` et exécuter `./ex2`.

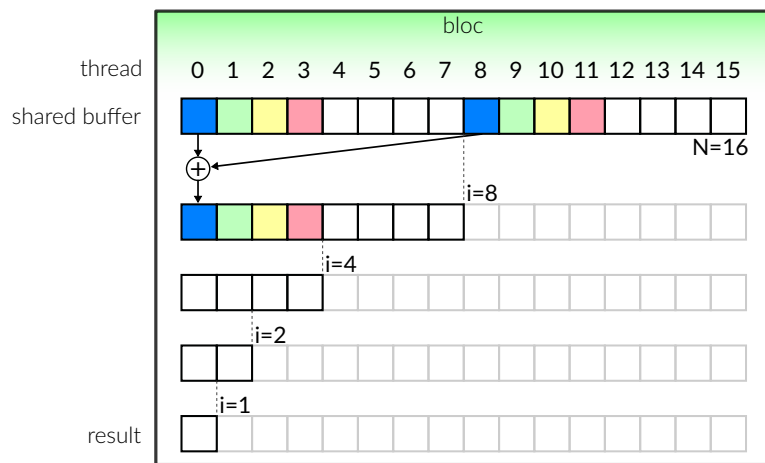


Figure 4: Réduction parallèle dans un *bloc*. Dans cet exemple, 16 *threads* somment les 16 valeurs d'un *buffer* de la mémoire partagée en 4 itérations. A chaque itération, les premiers *i* *threads* effectuent 1 addition de 2 valeurs et écrivent le résultat dans le même *buffer*. A la fin, le premier *thread* écrit le résultat dans un tableau de la mémoire globale. A chaque itération, les *threads* du *bloc* sont synchronisés. Les couleurs bleue, verte, jaune et rouge indiquent les 4 paires de valeurs additionnées par les 4 premiers *threads* lors de la 1^{ère} itération, ainsi que la zone du *buffer* où les résultats sont écrits.