SXS HPC : BE 3 – Calcul sur GPGPU

Auteur : Antoine Brunet <antoine.brunet@onera.fr>
Public : SXS
Date : 2023-2024

Ce BE consiste à implémenter un produit matriciel à l'aide de CUDA, et à analyser la performance de cette implémentation. Le BE se fait en binômes, et sera évalué sur les livrables suivants :

- Un rapport au format `pdf` présentant les réponses aux questions du BE.
- L'ensemble du code source produit.

Ces fichiers seront regroupés dans une archive, qui sera déposée sur LMS avant le 18 février 23h59 CET.

1 Implémentation du produit matricielle

On s'intéresse au calcul du produit de deux matrices carrées de taille $n \times n$:

$$C = A \times B \tag{1}$$

On notera a_{ij} (resp. b_{ij} et c_{ij}) les coefficients de la matrice A (resp. B et C). On a alors :

$$c_{ij} = \sum_k a_{ik} \cdot b_{kj} \tag{2}$$

Pour réaliser ce calcul, nous allons implémenter un kernel CUDA, dans lequel chaque thread va calculer un des coefficients c_{ij}. Vu le caractère bidimensionel du problème, on choisira une grille 2D de blocs de taille `BS` × `BS` threads. Dans cet exercice, on prendra une valeur de `BS` = 16, ce qui donne des blocs d'exécution de taille totale 256. On supposera pour le moment que n est divisible par `BS`.

Dans tout le BE, on utilisera la coordonnée x des blocs et des threads pour indicer les colonnes de la matrice C et la coordonnée y pour indicer ses lignes. On a donc

$$i = \texttt{BS} \times \texttt{blockIdx.y} + \texttt{threadIdx.y} \tag{3}$$

$$j = \texttt{BS} \times \texttt{blockIdx.x} + \texttt{threadIdx.x} \tag{4}$$

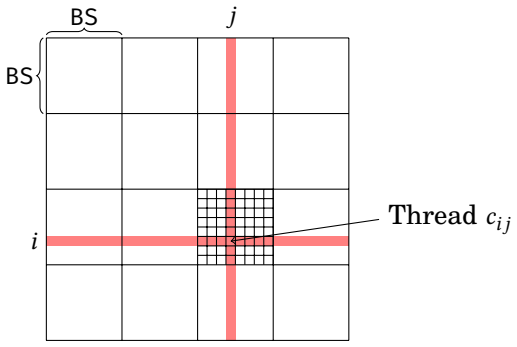

FIGURE 1 – Schéma du découpage du traitement de la matrice C, pour $n = 32$ et `BS` = 8

Pour tester notre implémentation, on réalisera le produit des matrices suivantes :

$$a_{ij} = \begin{cases} 0 & \text{si } i \neq j \\ i & \text{si } i = j \end{cases} \qquad b_{ij} = j \tag{5}$$

Question 1.1 :

Calculer, en fonction de n, le nombre de blocs dans chaque direction dans la grille d'exécution. Compléter la fonction `main` du programme `matmat.cu` avec la valeur calculée.

Question 1.2 :

Ajouter les instructions pour allouer en mémoire les trois matrices A, B, C, en utilisant le type `float` pour les coefficients de chaque matrice.

Question 1.3 :
Écrire un kernel `mat_init` permettant d'initialiser les matrices A, B et C, et ajouter l'appel correspondant dans la fonction `main`.

Question 1.4 :
Compléter le code de vérification de la fonction `main` en recopiant le contenu de la matrice C du GPU vers l'hôte, et en indiquant la valeur attendue pour chaque coefficient.

Question 1.5 :
Écrire un kernel `mat_mat` réalisant le produit matriciel. Dans cet exercice, on n'utilisera uniquement la mémoire globale du GPU, et on se passera de la mémoire partagée.

Question 1.6 :
En fonction de n, évaluer le temps d'exécution du kernel `mat_mat`. Combien d'opérations sont nécessaires pour réaliser le produit matriciel ? (On comptera le nombre total d'additions et de multiplications de flottants) Tracer la vitesse de calcul, en FLOPS, en fonction de n, pour des tailles de matrices allant de 16 à 2048. Commenter.

2 Utilisation de la mémoire partagée

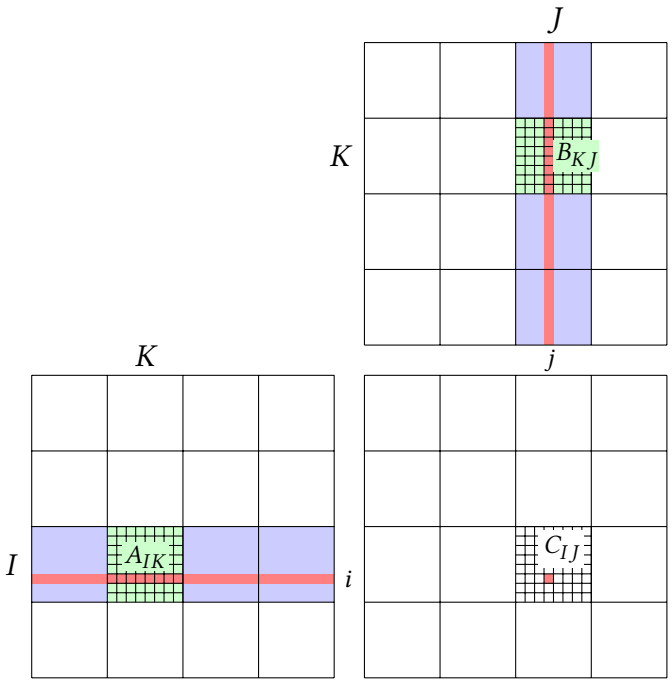

FIGURE 2 – Schéma du produit matriciel par blocs, pour $n = 32$ et $\texttt{BS} = 8$

Nous souhaitons améliorer les performances du programme écrit à la question précédente. Pour mieux identifier les accès mémoires communs à tous les threads d'un même bloc, nous décidons de réécrire le produit matriciel *par blocs*.

Notons A_{IJ} (resp. B_{IJ} et C_{IJ}) les sous-matrices de taille $\texttt{BS} \times \texttt{BS}$ correspondantes aux différents blocs d'exécution. On utilisera les indices majuscules I, J pour indicer les blocs, et les indices minuscules i, j pour indicer les coefficients.

On peut réécrire le produit matriciel par blocs :

$$C_{IJ} = \sum_K A_{IK} \cdot B_{KJ} \tag{6}$$

Dans l'équation 6, toutes les opérations portent sur des sous-matrices de taille $\texttt{BS} \times \texttt{BS}$. On note également que chacun des blocs C_{IJ} correspond à la partie du calcul de C traitée par un des blocs d'exécution du kernel `mat_mat`.

On propose la méthode de calcul suivant pour le calcul du bloc C_{IJ} :

1. Chaque thread initialise une variable locale `somme`.
2. Pour $K \in [0, \texttt{nb\_bloc}[$:
   (a) On charge A_{IK} et B_{KJ} dans des tableaux en mémoire partagée.
   (b) Chaque thread calcule le coefficient du produit $A_{IK} \cdot B_{KJ}$ correspondant à son indice dans le bloc, et incrémente `somme`.
3. Chaque thread vient écrire le résultat correspondant à son indice global dans la matrice C.

Question 2.1 :
Est-ce que cet algorithme peut permettre d'améliorer le temps de calcul ? Pourquoi ?

Question 2.2 :
Écrire un kernel `mat_mat_s` qui implémente cette méthode de calcul. On pourra toujours suppose que `BS` divise n. Décrire les difficultées rencontrées et les choix de développements réalisés.

Question 2.3 :
Mesurer le temps de calcul de ce nouveau kernel et comparer à celui de `mat_mat`, pour différentes tailles de matrices. Discuter.

Question 2.4 :
Pour $n = 2048$, faire varier la taille des blocs `BS` et discuter de l'impact de ce paramètre sur les performances de calcul. Pour quelle taille de bloc avont nous les meilleures performances ?

Question 2.5 : optionnel
Dans le programme `matmat_omp.c`, implémenter le produit matriciel avec OpenMP, puis tracer la vitesse de calcul sur CPU en fonction du nombre de threads alloués. Comparer aux performances obtenues à la question précédente.

License CC BY-NC-SA 3.0