Introduction à la Programmation GPU

TP5 : Multiplication matricielle

Majeure SSIE - EPITA - 2024

L'objectif de ce TP est d'implémenter l'opération de multiplication matricielle $C = A * B$ entre deux matrices d'entier A et B sur GPU. Une matrice est un tableau 2D représenté linéairement par un tableau unidimensionnel. Chaque matrice est stockée en mémoire ligne par ligne (en ordre "row-major"), comme illustré par la Figure 1.

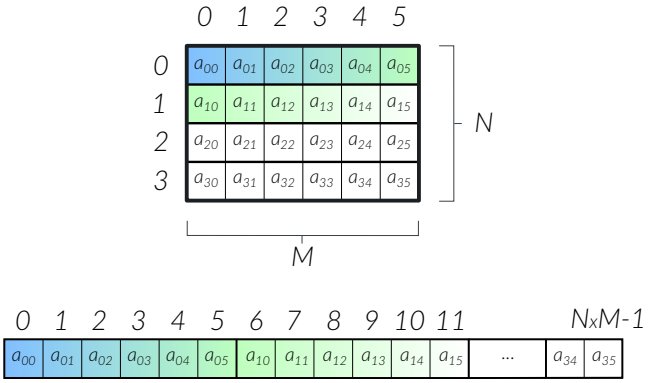

Figure 1: Matrice 2D de taille $N \times M$ stockée dans un tableau 1D ligne par ligne (en ordre "row-major").

Dans ce TP, A est une matrice d'entier de taille $N \times M$ (N lignes et M colonnes), B de taille $M \times P$, et C est la matrice résultante de taille $N \times P$. Les exercices sont à implémenter uniquement dans le fichier `matrix.cu`. Tous les noyaux CUDA sont configurés en 2D avec des blocs de taille `T`×`T` threads (`T=16`).

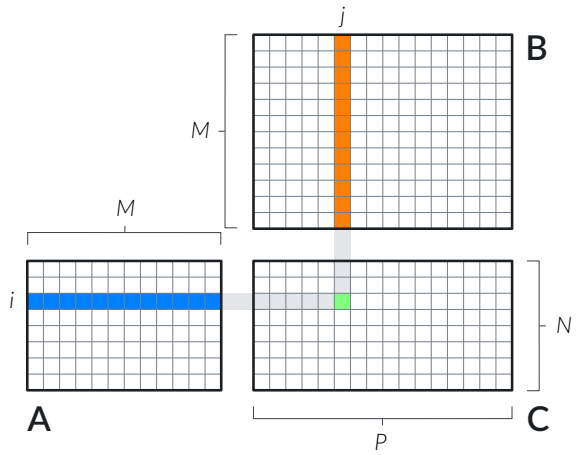

Figure 2: Multiplication matricielle

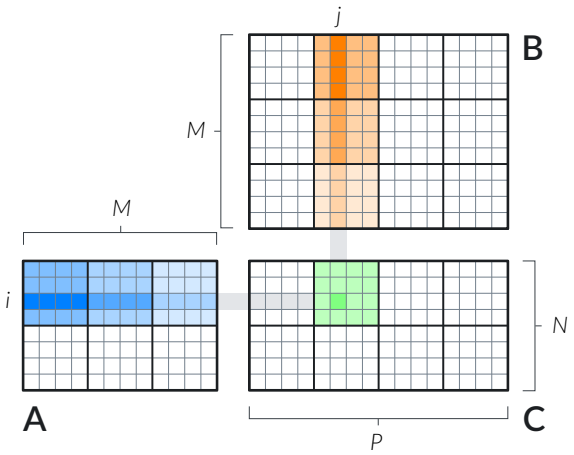

Figure 3: Multiplication matricielle par bloc

1. Multiplication matricielle sur CPU

Dans ce premier exercice, la multiplication matricielle est effectuée séquentiellement, élément par élément, selon l'Équation 1.

$$c_{ij} = \sum_{k=0}^{M-1} a_{ik} * b_{kj}, \qquad i \in (0, N-1), \quad j \in (0, P-1) \tag{1}$$

Comme illustré par la Figure 2, un indice k entre 0 et $M-1$ permets d'additionner dans **une** boucle `for` les M multiplications entre paires d'éléments a_{ik} de A (en bleu) et b_{kj} de B (en orange) afin de calculer l'élément c_{ij} de C (en vert).

➜ **Step 01.** Dans le fichier `matrix.cu`, implémenter la fonction `index1` qui calcule l'indice 1D de l'élément à la ligne `i` et à la colonne `j` selon le stockage de matrice en ligne (*cf* Figure 1).

➜ **Step 02.** Implémenter la fonction `matmul1` qui effectue l'opération de multiplication matricielle sur CPU. Compiler avec la commande `nvcc main.cu matrix.cu -o exec` puis exécuter `./exec`.

2. Multiplication matricielle sur GPU

Dans un premier temps, la multiplication matricielle est effectuée naïvement sur GPU avec un noyau CUDA 2D, où chaque thread calcule un élément c_{ij} selon l'Équation 1 et la Figure 2. Chaque thread calcule la somme des M éléments sans optimisation particulière.

➜ **Step 03.** Implémenter le noyau CUDA `kernel::matmul2` qui calcule la multiplication matricielle en parallèle sur GPU.

➜ **Step 04.** Implémenter la fonction `matmul2` qui gère le lancement du noyau `kernel::matmul2` ainsi que la mémoire du device. Calculer le nombre de bloc nécessaire dans les deux dimensions pour couvrir la matrice C avec des blocs de taille `T`×`T`. Compiler `nvcc main.cu matrix.cu -o exec` puis exécuter `./exec`.

3. Multiplication matricielle par bloc sur GPU

La mémoire partagée (*shared memory*) entre les threads d'un bloc CUDA et le mécanisme de synchronisation des threads d'un bloc permettent d'améliorer les performances de calculs sur GPU. D'après la Figure 3, un bloc CUDA qui calcule une sous-matrice de C (en vert) peut être décomposé en S étapes ($S = 3$ sur la figure) en considérant S paires de sous-matrices de taille `T` × `T` dans A et B. À chaque étape, les sous-matrices de A et B sont chargées dans la *shared memory* par tous les threads d'un bloc CUDA qui sont ensuite synchronisés par `__syncthreads()`, puis une somme partielle de la forme de l'Équation 1 est accumulée dans une variable locale `accu`. Le noyau CUDA `kernel::matmul3` implémente le pseudo-code donné par l'Algorithme 1.

Algorithm 1 `__global__ void kernel::matmul3()`

```
1:  (tx,ty) = (threadIdx.x,threadIdx.y)          ▷ CUDA threads indices in the bloc
2:  (bx,by) = (blocIdx.x,blocIdx.y)              ▷ CUDA bloc indices in the grid
3:  S = ...                                      ▷ number of sub-matrices
4:  accu = 0                                     ▷ accumulated result for one element of C
5:  for s = 0...S-1 do                           ▷ for each pair of sub-matrices in A and B
6:     __shared__ float s_A[T][T];               ▷ sub-matrix (bx,s) in the shared memory
7:     __shared__ float s_B[T][T];               ▷ sub-matrix (s,by) in the shared memory
8:     ...                                       ▷ load sub-matrices in the shared memory
9:     __syncthreads();                          ▷ synchronize threads of the bloc
10:    ...                                       ▷ compute partial sum in accu using s_A and s_B
11:    __syncthreads();                          ▷ synchronize threads of the bloc
12: ...                                          ▷ set result accu in C
```

➜ **Step 05.** Implémenter la fonction `index2` qui calcule l'indice dans le tableau 1D qui correspond aux indices de ligne et colonne (i, j) de la sous-matrice indicée par (bi, bj). Contrairement à la fonction `index1` où les indices (i, j) appartenaient aux intervalles $(0, N-1) \times (0, P-1)$, les indices (i, j) de la sous-matrice appartiennent ici aux intervalles $(0, \mathsf{T}-1) \times (0, \mathsf{T}-1)$. La fonction `index1` peut être appelée par `index2`.

➜ **Step 06.** Implémenter le noyau CUDA `kernel::matmul3` selon l'Algorithme 1. Les tailles de matrice sont supposées proportionnelles à la taille des blocs CUDA et donc à la taille des sous-matrices. Tous les blocs CUDA et les sous-matrices sont supposés carrés.

➜ **Step 07.** Implémenter la fonction `matmul3` de manière similaire à `matmul2`. Compiler `nvcc main.cu matrix.cu -o exec` puis exécuter `./exec`.