Devoir No. 6 et Lab. # 6

PRÉ-REQUIS

* Vous devez avoir complété le Pré-Lab 3 et lu les chapitres 2 à 5 de la référence [2]
* À faire avant de commencer le lab (Voir annexe A)
  1. Ajouter “cl.exe” aux variables d’environnement du système

# PARTIE 1 - MULTIPLICATION MATRICE-MATRICE

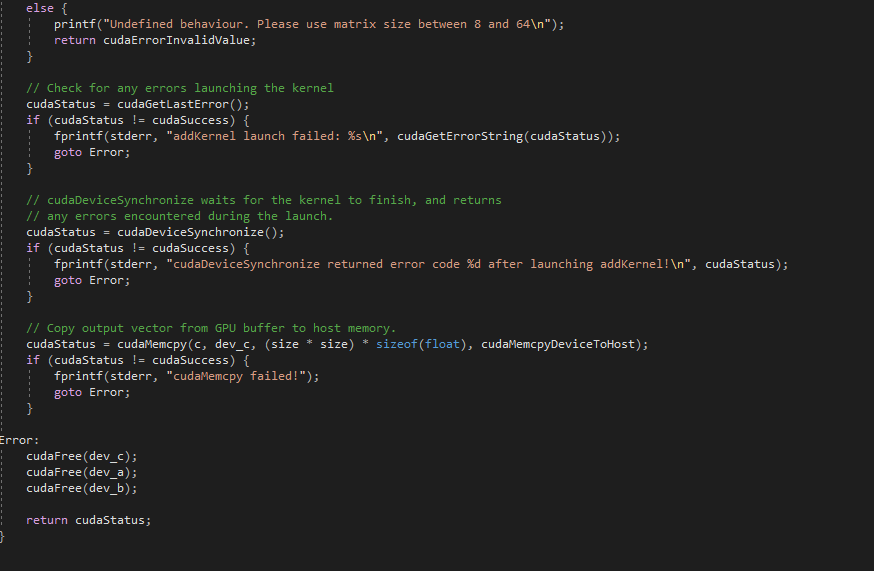
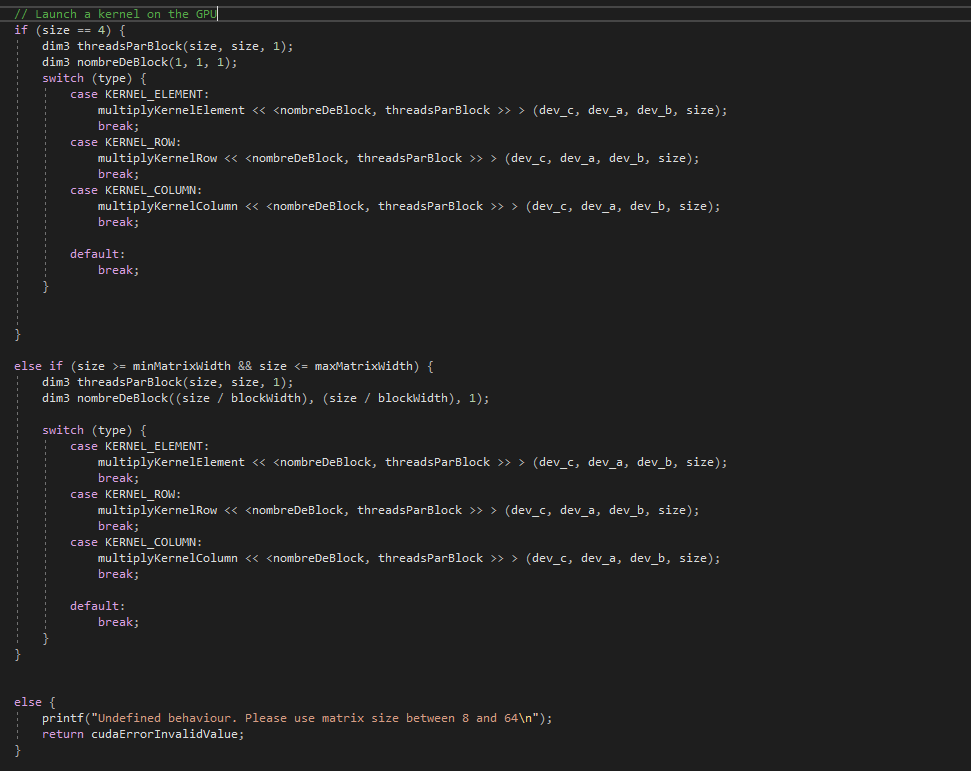
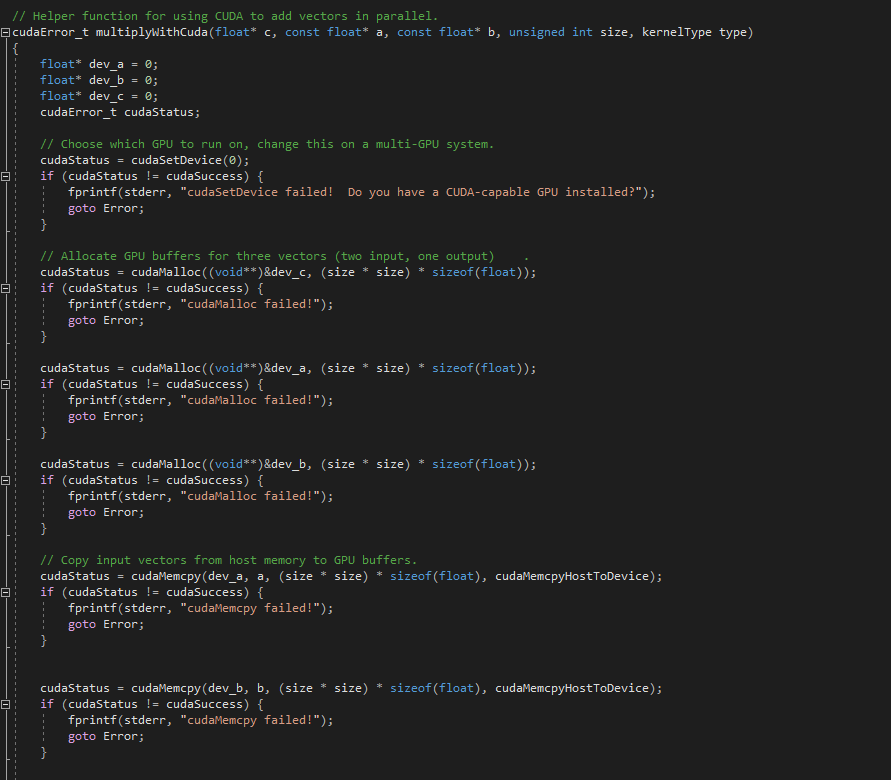
**Objectif :** Écrire une multiplication matricielle C = A x B. Par souci de simplicité, nous allons uniquement utiliser des matrices carrées dont les éléments sont des single-precision floating- point.

#### Méthodologie :

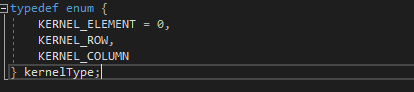
Suivez les instructions ci-dessous :

1. Écrire une fonction “host-stub ”qui allouera la mémoire pour les matrices d’entrées et la matrice de sortie, fera le transfert de ces matrices vers le Device, lancera le kernel, fera le transfert de la matrice de sortie vers l’Host et libèrera la mémoire des matrices d’entrées et de sortie sur le Device. Laissez les configurations d’exécution vide pour cette étape.

Cette fonction a quelque peu changé depuis mon laboratoire #5 :

Comme vous pouvez constater, dépendemment de la grosseur de la matrice entrée, le kernell sera calculé différement. Dans ce programme, je peux faire la multiplicaton matricielle sur des matrices de 4\*4, et tout autre matrice carré qui ont une dimension entre 8 et 64. Si une taille plus grande est rentré, un message est écrit pour indiquer que ce calcul de matrice n’est pas pris en compe dans ce programme.

Pour rendre ce code plus agréable à lire, j’ai créé un enum au début de mon programme pour indiquer les différents type de calcul il est possible de faire avec les threads.



Lorsqu’on appelle la fonction multiplyWithCuda, il est donc important de spécifier quel type d’opération qu’on veut faire tel qu’énoncé dans le cahier dans le laboratoire.

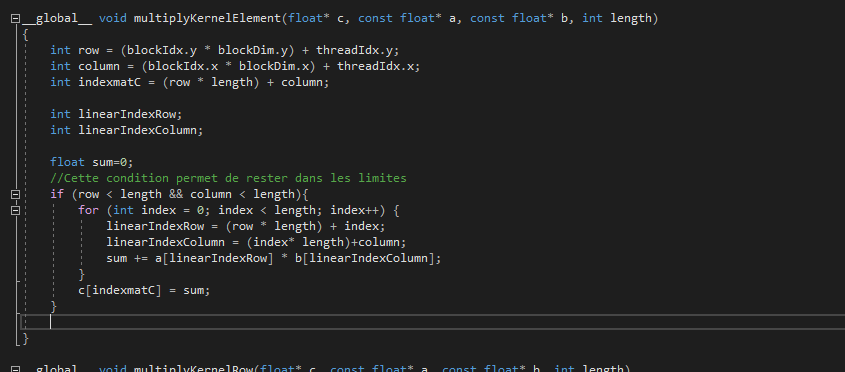
J’ai aussi créé des variables constantes nous permettant de rapidement changer différentes fonctionnalités du programme :



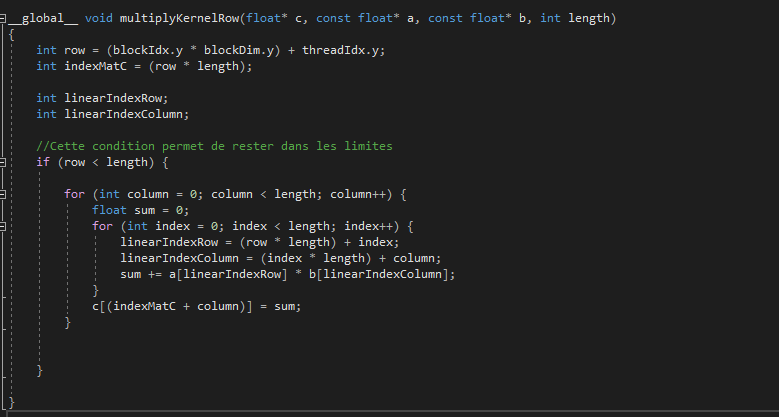
J’ai aussi corrigé l’allocation de la mémoire pour correspondre au float.

## Écrivez un kernel où chaque thread calculera un seul élément de la matrice de sortie. Ajouter

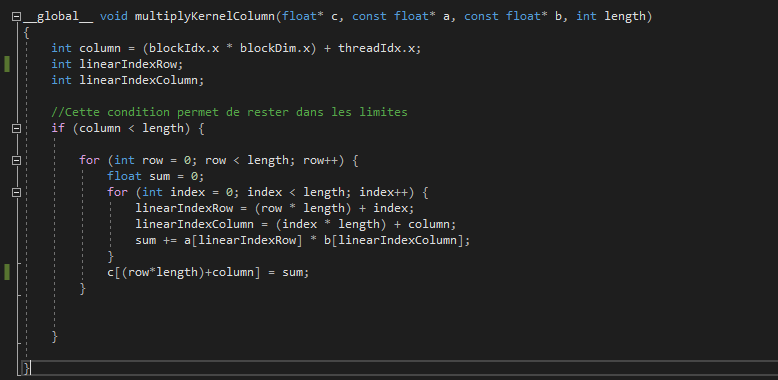
les paramètres d’exécution dans la fonction « Host-Stub » nécessaire à ce kernel.

****Cette fonction permet de calculer un élément de la matrice de sortie. Contrairement au devoir 5, chaque thread doit quand même faire une loop car la matrice de sortie est une somme de toutes les multiplications rangées par colonnes de chaque élément des matrices A et B.

1. **Écrivez un kernel où chaque thread calculera tous les éléments d’une ligne de la matrice de sortie.** Ajouter les paramètres d’exécution dans la fonction « Host-Stub » nécessaire à ce kernel.

Pour faire ceci, il est inévitable de faire une loop imbriqué dans une autre loop, car on doit calculer tous les élélements de la matrice de sortie. En tant que tel, le calcul reste très similaire que la question précédente dans la loop imbriqué, sauf que la première loop va garder la colonne respectif de la matrice de sortie, car on doit calculer une rangée complète dans ce thread.

1. **Écrivez un kernel où chaque thread calculera tous les éléments d’une colonne de la matrice de sortie.** Ajouter les paramètres d’exécution dans la fonction « Host-Stub » nécessaire à ce kernel.

Cette fois-ci, l’implémentation est très similaire que la question précédente, sauf qu’il faut calculer une colonne pour la matrice de sortie. Pour faire ceci, mathématiquement il faut calculer chaque rangée de la matrice A pour une colonne de la matrice B, et ceci donnera une colonne de la matrice C.

1. Analysez les avantages et inconvénients de chaque kernel ci-dessus.

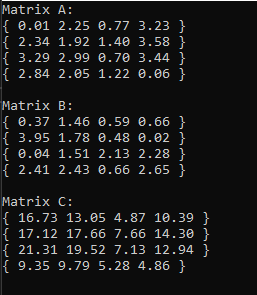
Pour A à E,

1. Validez que le kernel est fonctionnel et effectue le bon calcul en utilisant un simple programme en CUDA C où :

## Les matrices doivent avoir une taille variable 4x4, 8x8, 32x32 et 64x64.

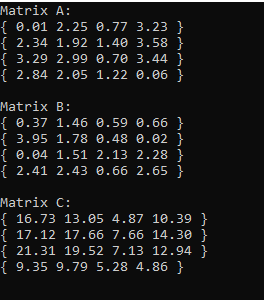
## Matrice 4\*4

## Élément :



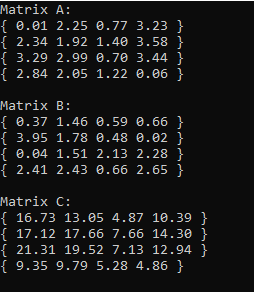
Rangée :

****

****

Colonne :

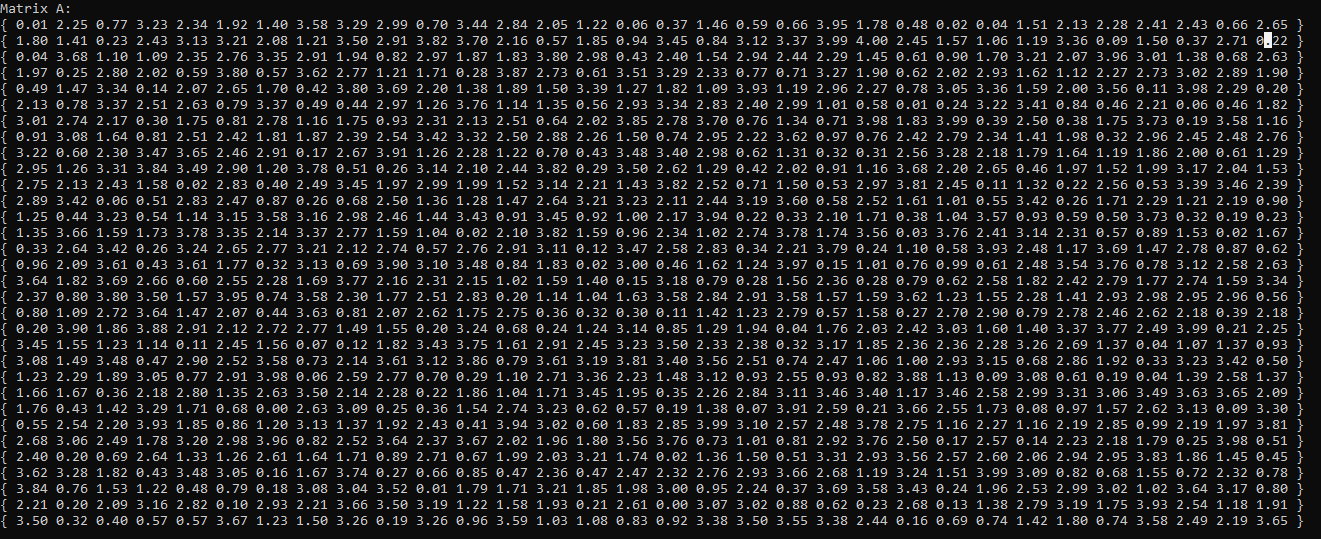


****

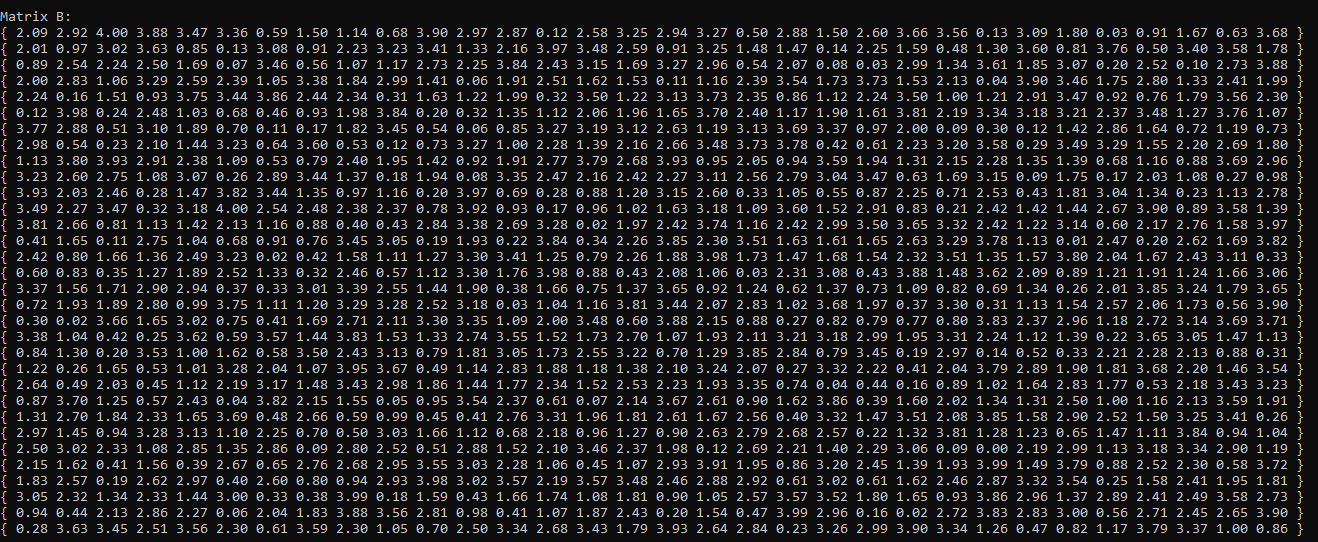
**Matrice 32\*32**

**Élément :**

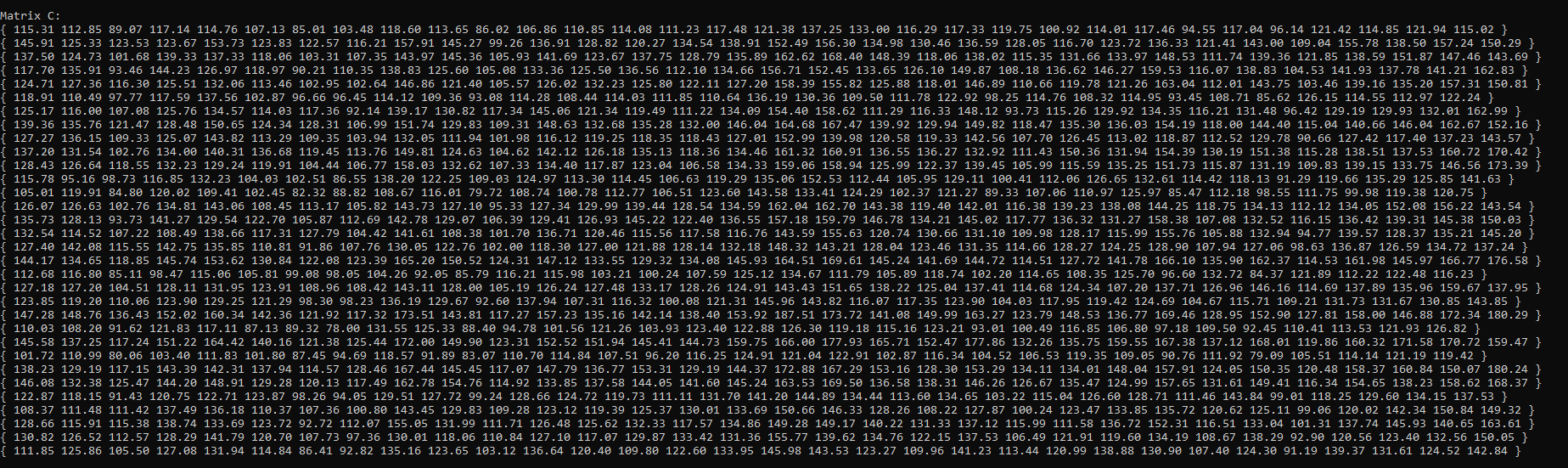
****

Matrice A :****

Matrice B :

****

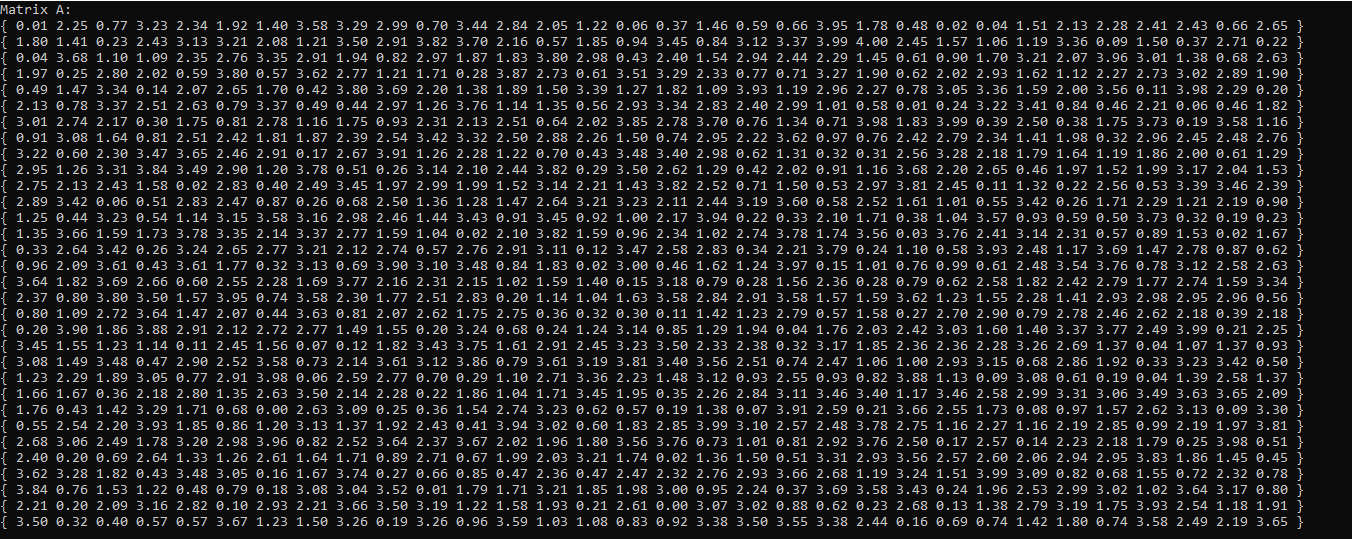
Matrice C :



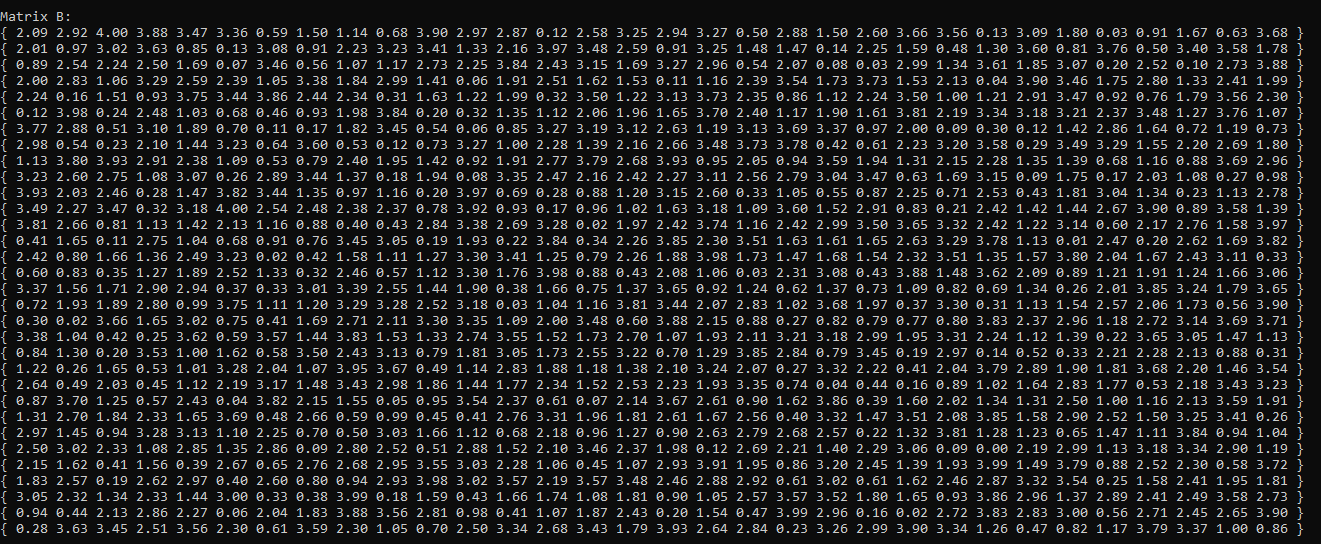
**Rangée :**

****

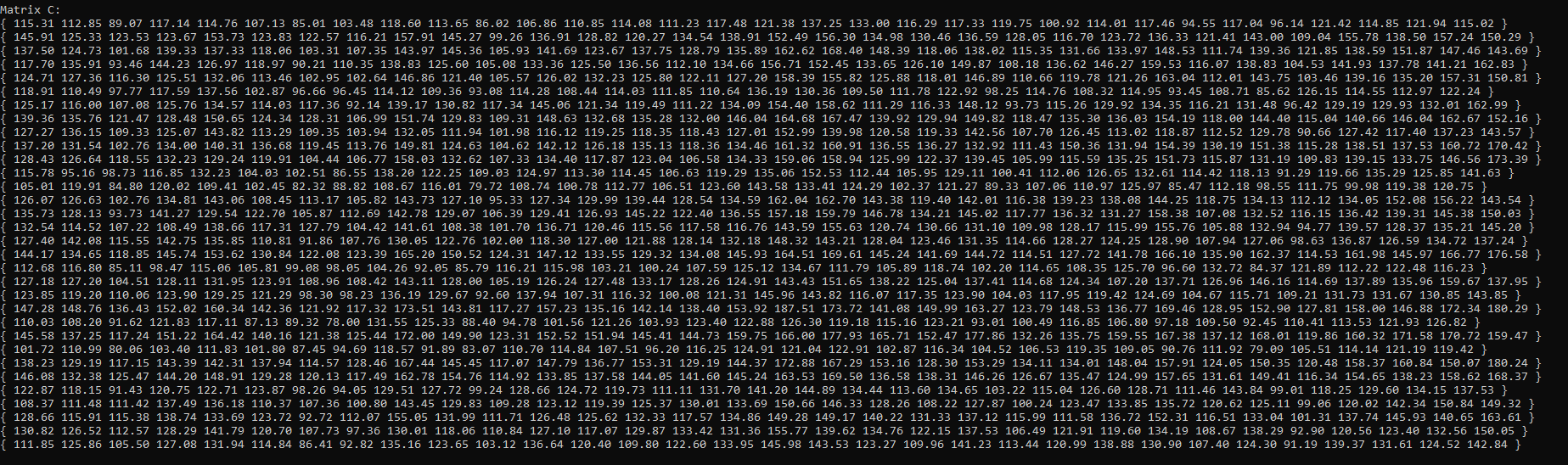
Matrice A :



Matrice B :

****

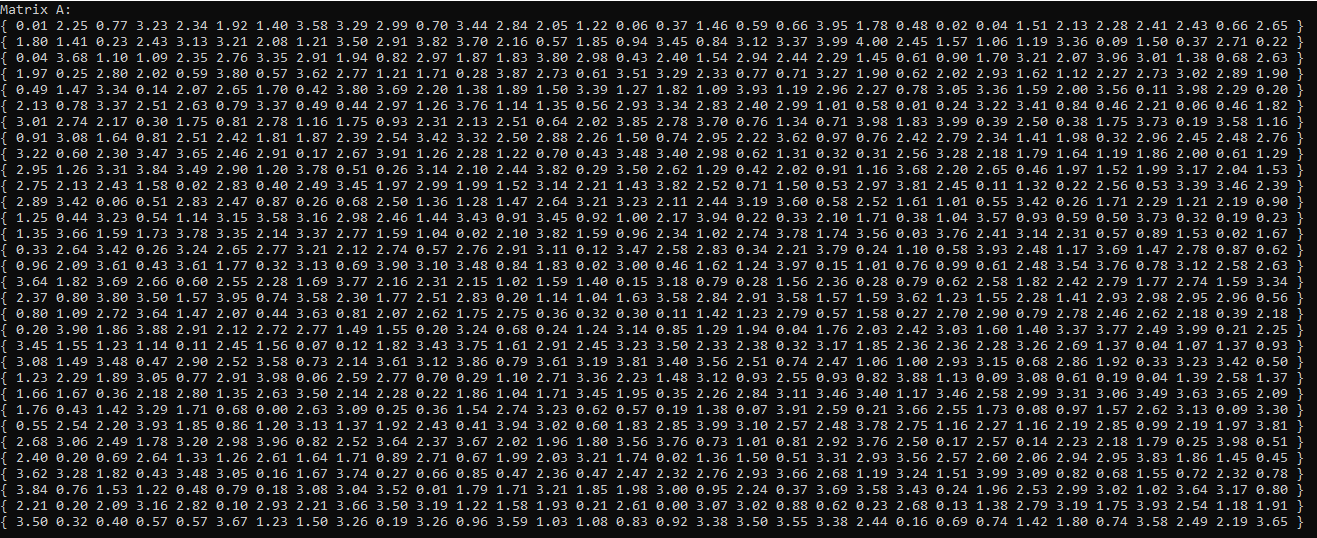
Matrice C :



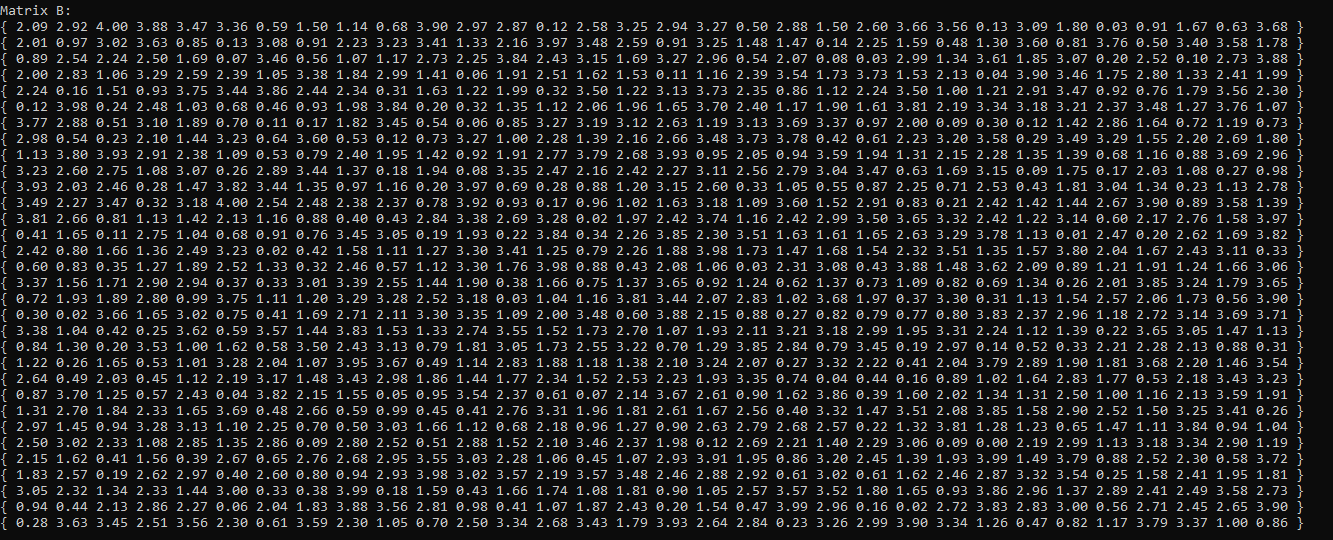
**Colonne :**

****

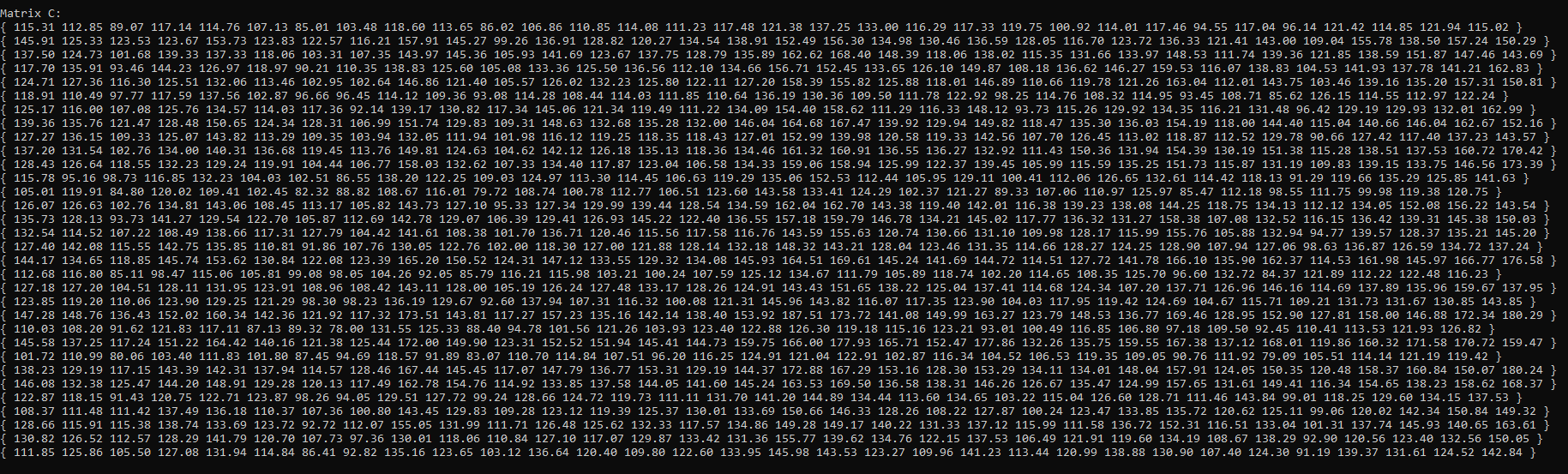
Matrice A :



Matrice B :

****

Matrice C :

****

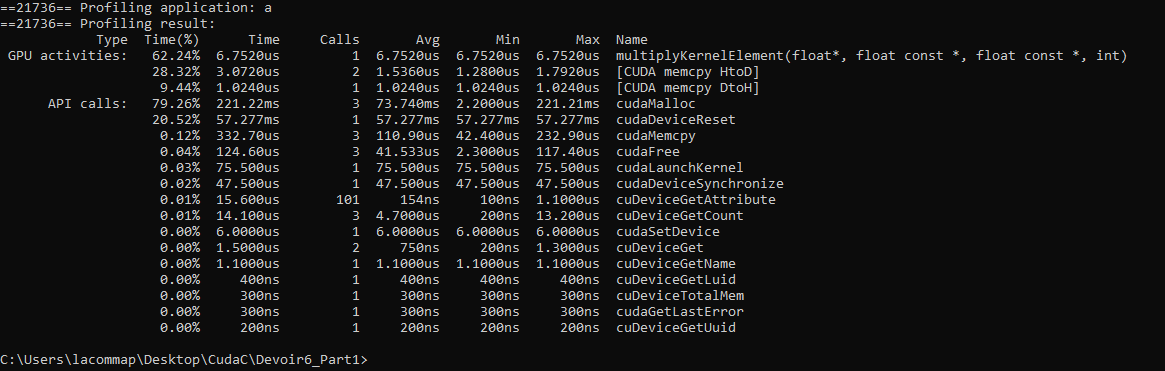
* 1. **Les blocs doivent avoir une taille de 8x8 (à l’exception du cas de matrice 4x4)**

1. Profilez votre kernel et discuter des résultats

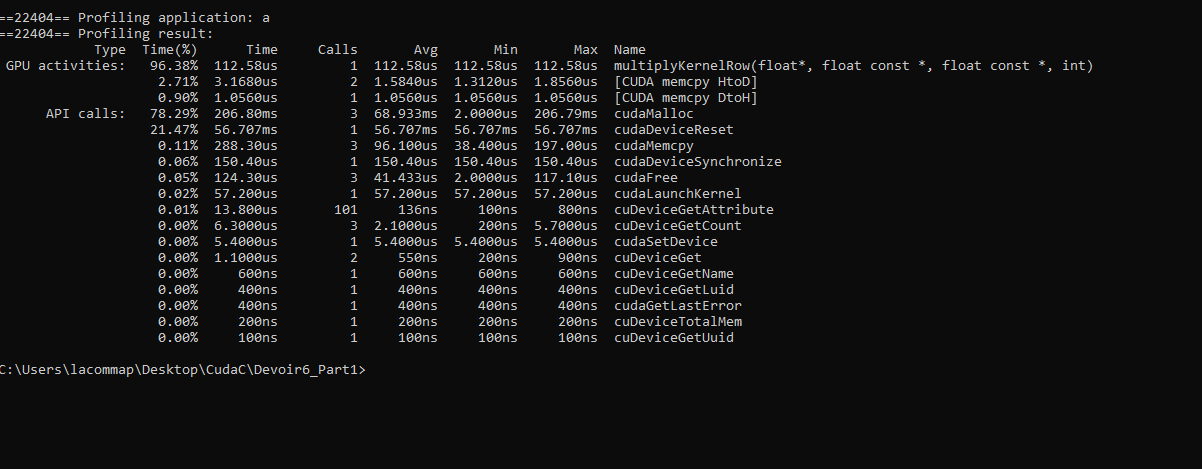
Avec des matrices 32\*32 :

NvProf :

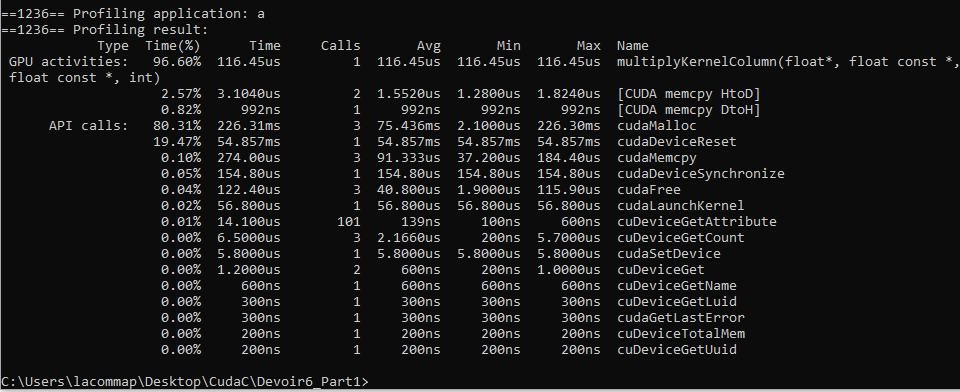
Élément :



Rangée :

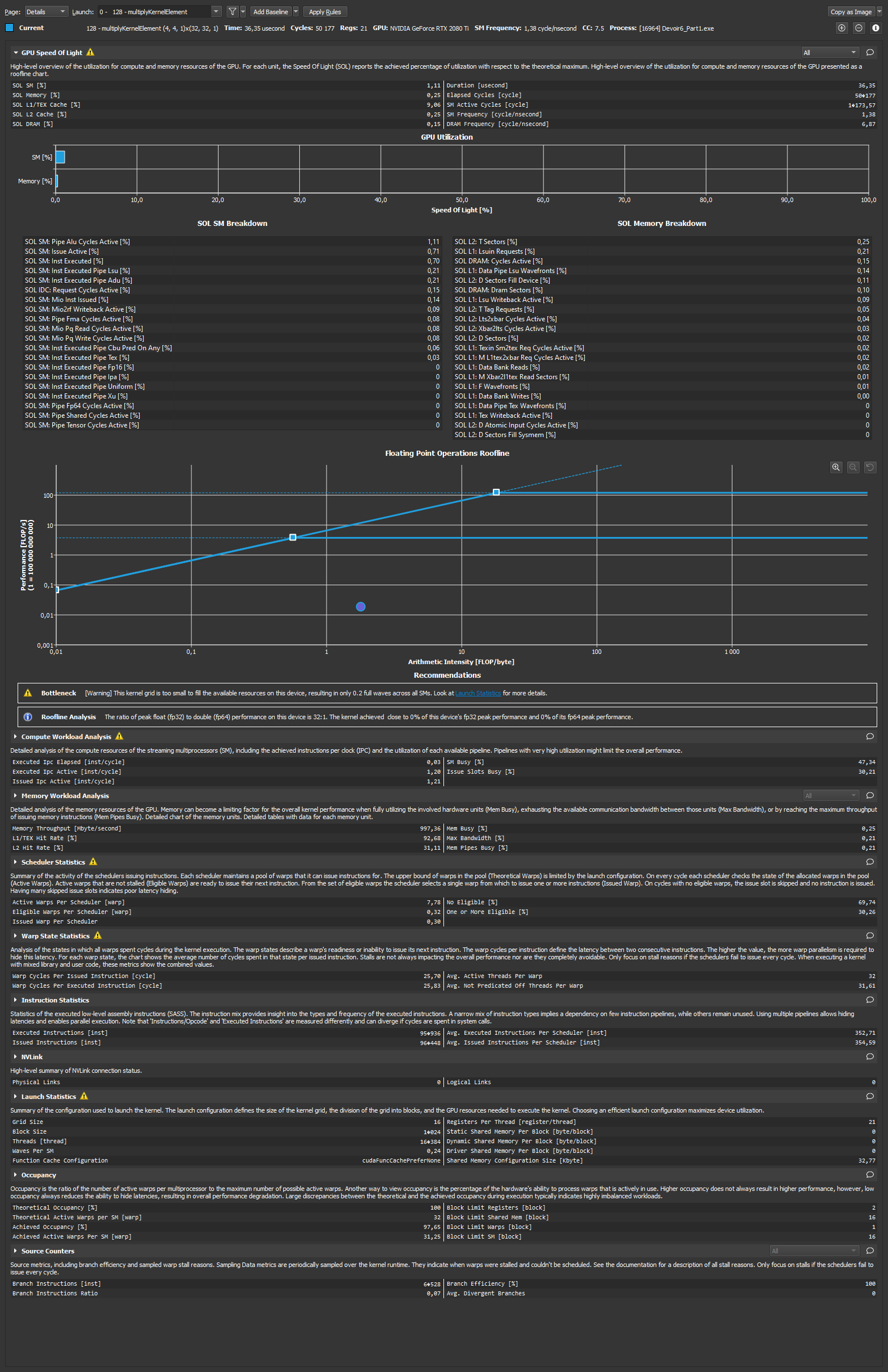


Colonne :

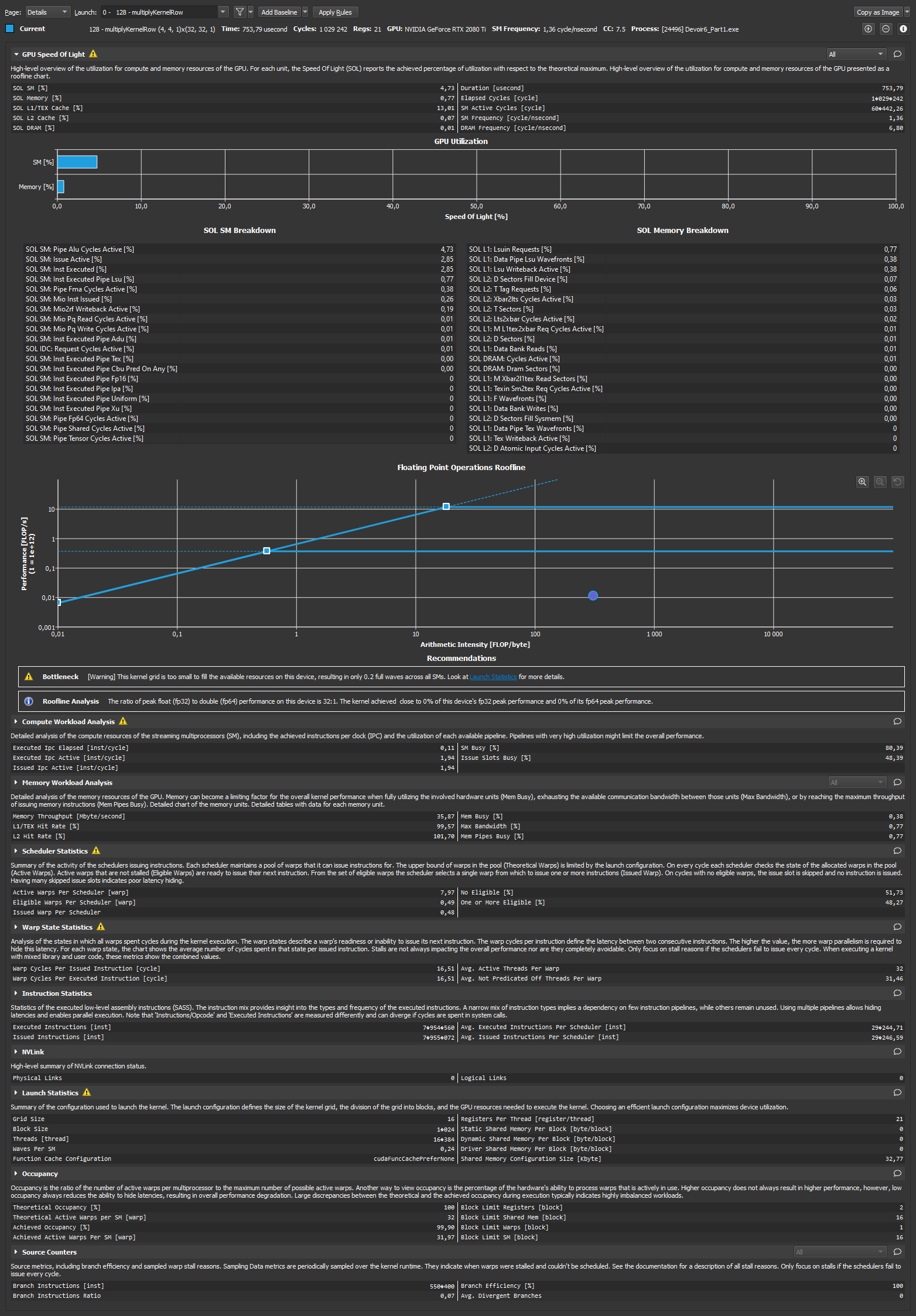


NSight :

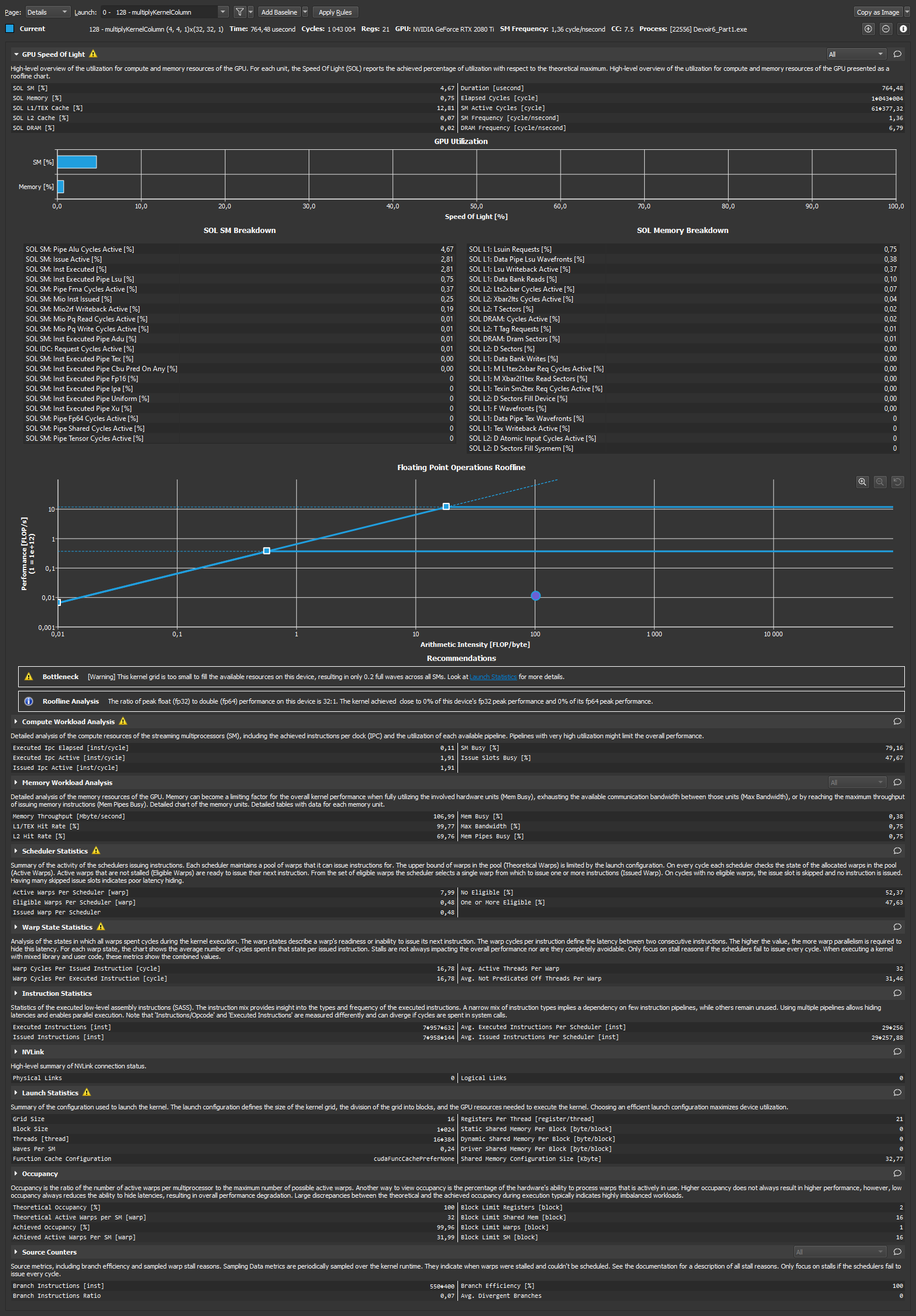
Élément :



Rangée :



Colonne :



# PARTIE 2 – MULTIPLICATION MATRICIELLE OPTIMISÉE

#### Objectifs

Réimplémentez la multiplication matricielle de la PARTIE 1 en utilisant l’algorithme de tiling [chap. 4, 2]. En programmation paralèlle, le tiling force plusieurs threads à se concentrer sur un sous ensemble de données d’entrée lors de chaque phase de sorte que ce sous ensemble de donnée puisse être stocké dans une mémoire spéciale, la shared memory par exemple, ce qui accélère grandement les accès mémoire. L’étudiant doit démontrer qu’avec des tiles de grande taille, les accès à la mémoire globale n’est plus un principal facteur limitant.

D’autre part, la capacité de raisonner en fonction des limitations matérielles en développant une application est un aspect clé de la pensée computationnelle. Les programmeurs CUDA doivent être conscient des limitations de chaque type de mémoire ainsi que des ressources propres à leur GPU puisqu’elles sont étroitement liées à l’implémentation. Une fois les capacités excédées, le GPU limite le nombre de threads pouvant être exécuter dans chaque SM. Les threads sont assignés aux SM sur une base de bloc par bloc. Chaque device CUDA dispose d’une limitation différente par rapport aux ressources disponible dans les SM, autant une limite de blocs par SM et une limite de threads par SM. La première limitation atteinte deviendra le facteur limitant.

Pour chaque Kernel, l’une ou plusieurs de ces limitations peut devenir le facteur limitant pour le nombre

de threads qui sont utilisés simultanément dans le device CUDA.

### MÉTHODOLOGIE

1. Implémentez la multiplication matricielle en suivant les mêmes instructions que pour la PARTIE 1.

Toutefois, l’implémentation doit être faite en utilisant des techniques de tiling.

1. Votre implémentation doit être « transparent scalable » (voir dection 3.4 à 3.6 dans [chap.3, 2]). Pour ce faire, vous devez connaître les propriétés de votre device. L’annexe B fournit un exemple d’une « host-stub function » qui permet d’extraire les propriétés.

#### L’implémentation doit être limitée à des matrices carrées 64x64. Toutefois, la taille des tuiles doit varier entre 2x2, 4x4, 8x8 et 16x16.

1. Validez chaque implementation.
2. Profilez chaque implementation.
3. Discutez de la vitesse d’exécution du kernel versus les ressources utilisées (shared memory,

registres, etc.).

1. Comparez et discutez des résultats par rapport à ceux obtenus à la PARTIE 1.

### REFERENCES

[2] D. Kirk and W.-M. Hwu, *Programming Massively Parallel Processors- A Hands-on Approach*, 3rd Edition, Morgan Kaufmann, 2017. (Elsevier Science & Technology*,* ISBN: 978-0-12-811986-0).