PRÉREQUIS

Devoir No. 5 et Lab. # 5

* Vous devez avoir complété le Pré-Lab 3
* Avoir lu les chapitres 3 et 4 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

Faire les exercices suivants à la fin du chapitre 2 de la référence [2] :

1. 2.1
2. 2.2
3. 2.3
4. 2.5
5. 2.6
6. 2.7
7. 2.8

# PARTIE 2

## Exercice 1

L’addition matricielle prend deux matrices A et B en entrées et produit une matrice C en sortie. Chaque élément de la matrice de sortie C est la somme des éléments correspondants des matrices d’entrées A et B, i.e., C[*i*][*j*] = A[*i*][*j*] + B[*i*][*j*]. Par souci de simplicité, nous allons uniquement utiliser des matrices carrées dont les éléments sont des single-precision floating-point. Écrivez un kernel qui effectue l’addition matricielle et la fonction « Host-Stub » qui peut être appeler avec 4 paramètres : pointeur vers la matrice de sortie C, pointeur vers la première matrice d’entrée A, pointeur vers la seconde matrice d’entrée B et la dimension des matrices. Suivez les instructions suivantes :

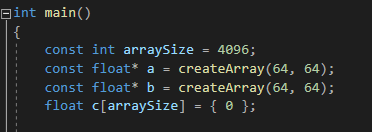
## Pour des matrices de taille 64x64 (Dimensions des blocs au choix des étudiants)

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.

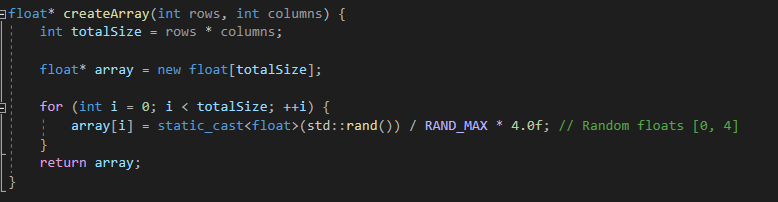
Pour faire la question A, heureusement, un projet de base dans Visual Studio fait presque tout ce qui est demandé.

Allocation de mémoire des matrices du host :

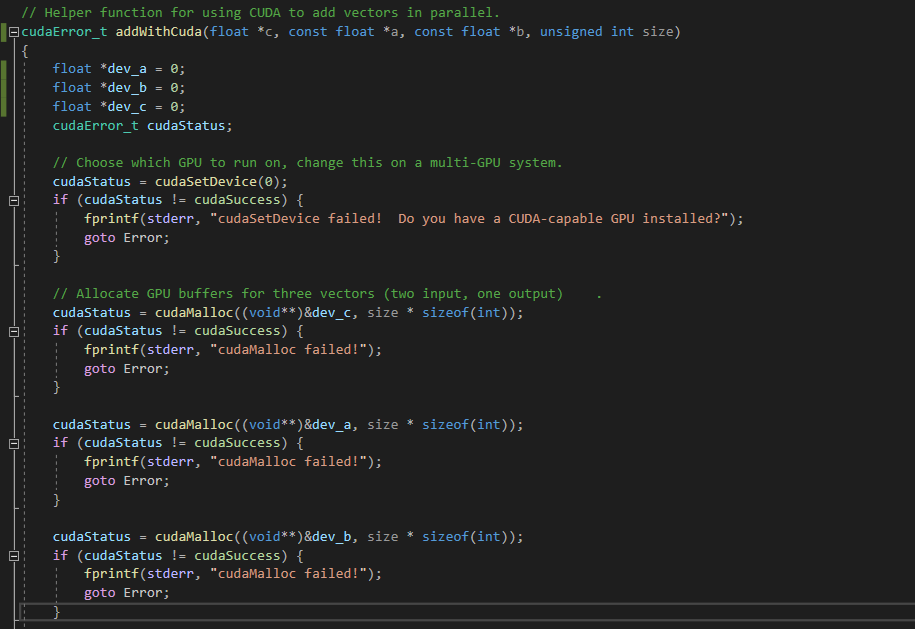
Dans la fonction main(), il y a l’intiailisation des matrices du host. J’ai créé une fonction createArray() par m’aider à générer des très grosses matrices.



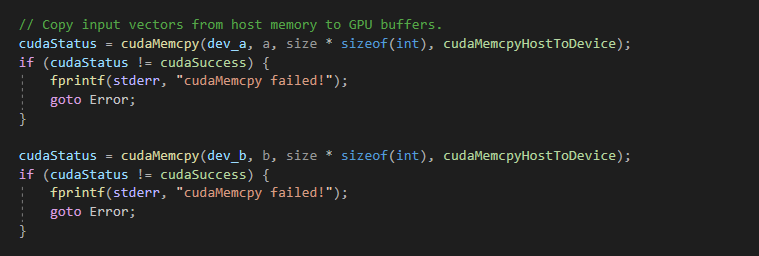
Voici ma function:

Allocation de mémoire des matrices du device :

Par la suite, encore une fois dans la fonction main() fournit par Visual Studio, on appelle la fonction addWithCuda(), celle-ci va allouer la mémoire nécessaire pour chaque matrice:

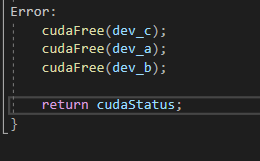


Transfert de ces matrices vers le Device :

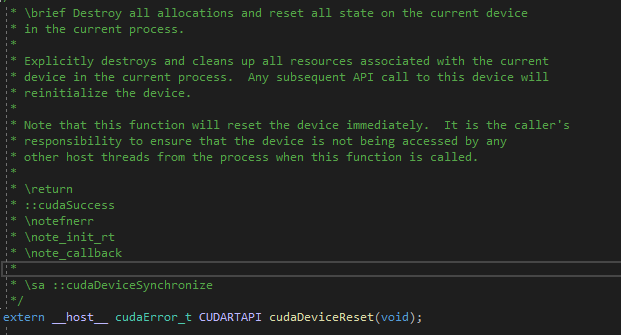


Libération de la mémoire du device :

Dans le cas où nous avons une erreur, la mémoire dans le device est libéré ici :

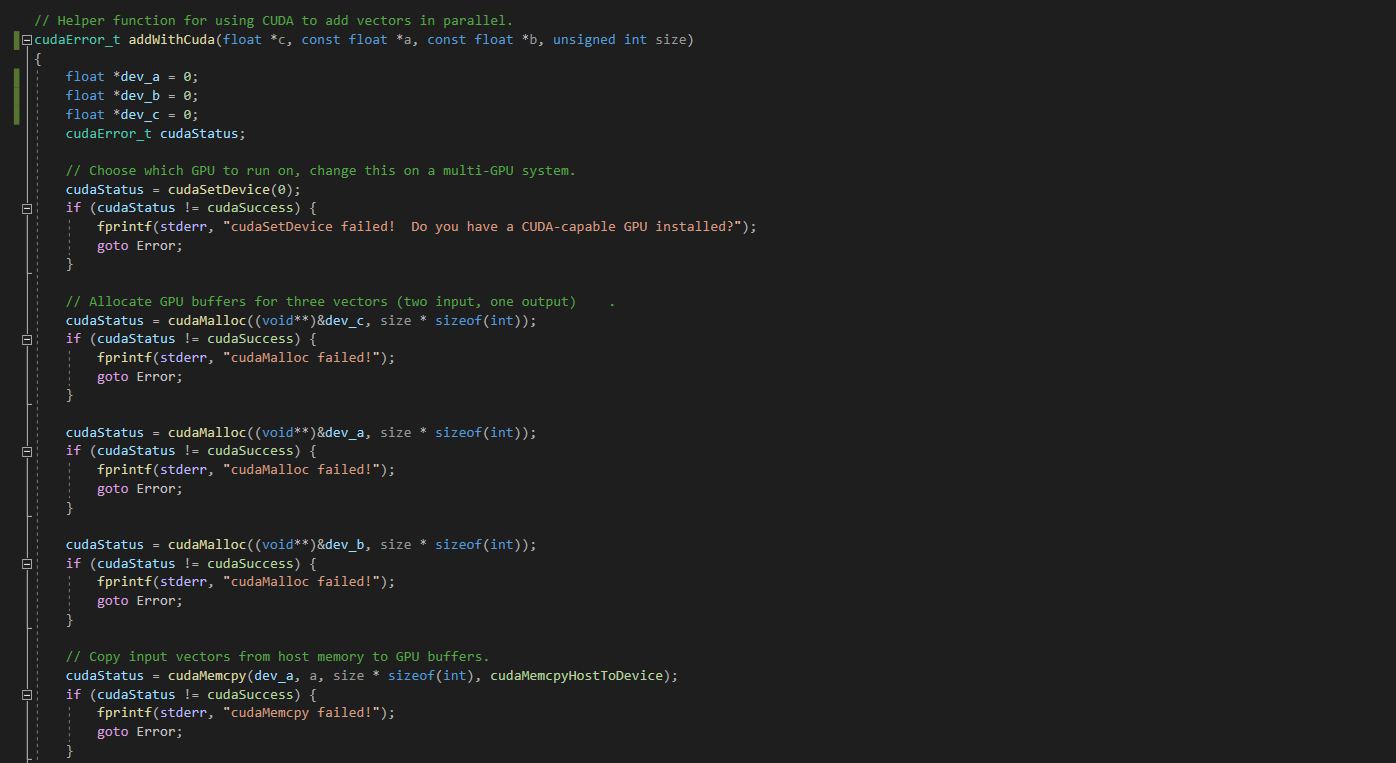


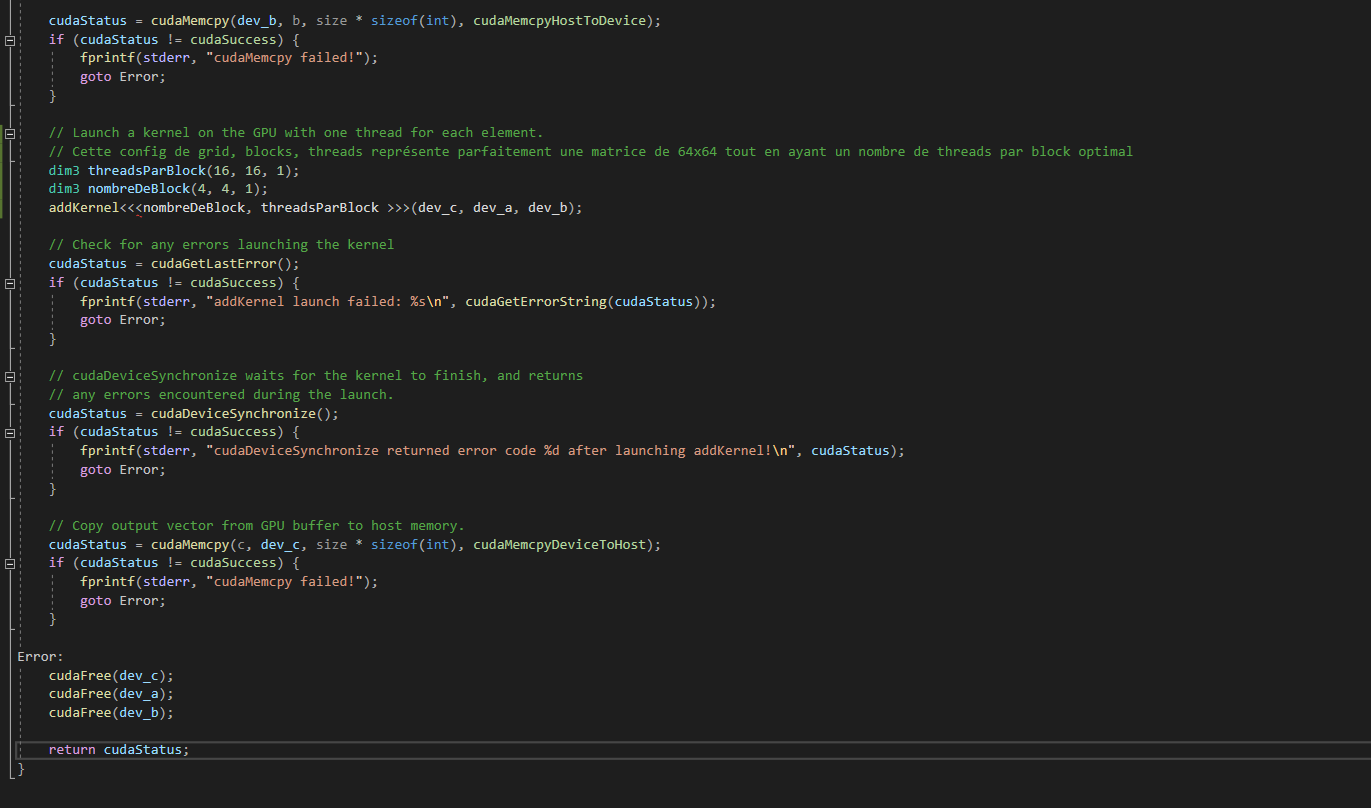
Sinon, si tout c’est bien déroulé alors dans le main(), la fonction cudaDeviceReset va tout détruire et nettoyer dans le device tel que indiqué dans le header de la fonction :



1. **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.

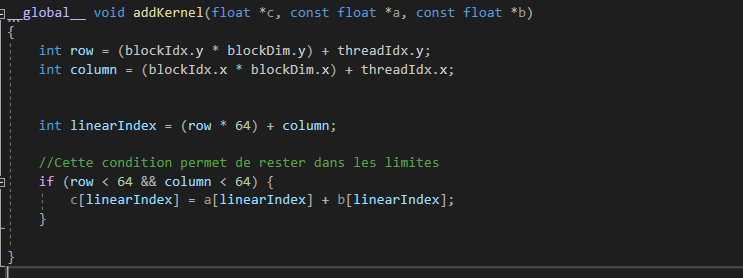
Voici la host-stub function :



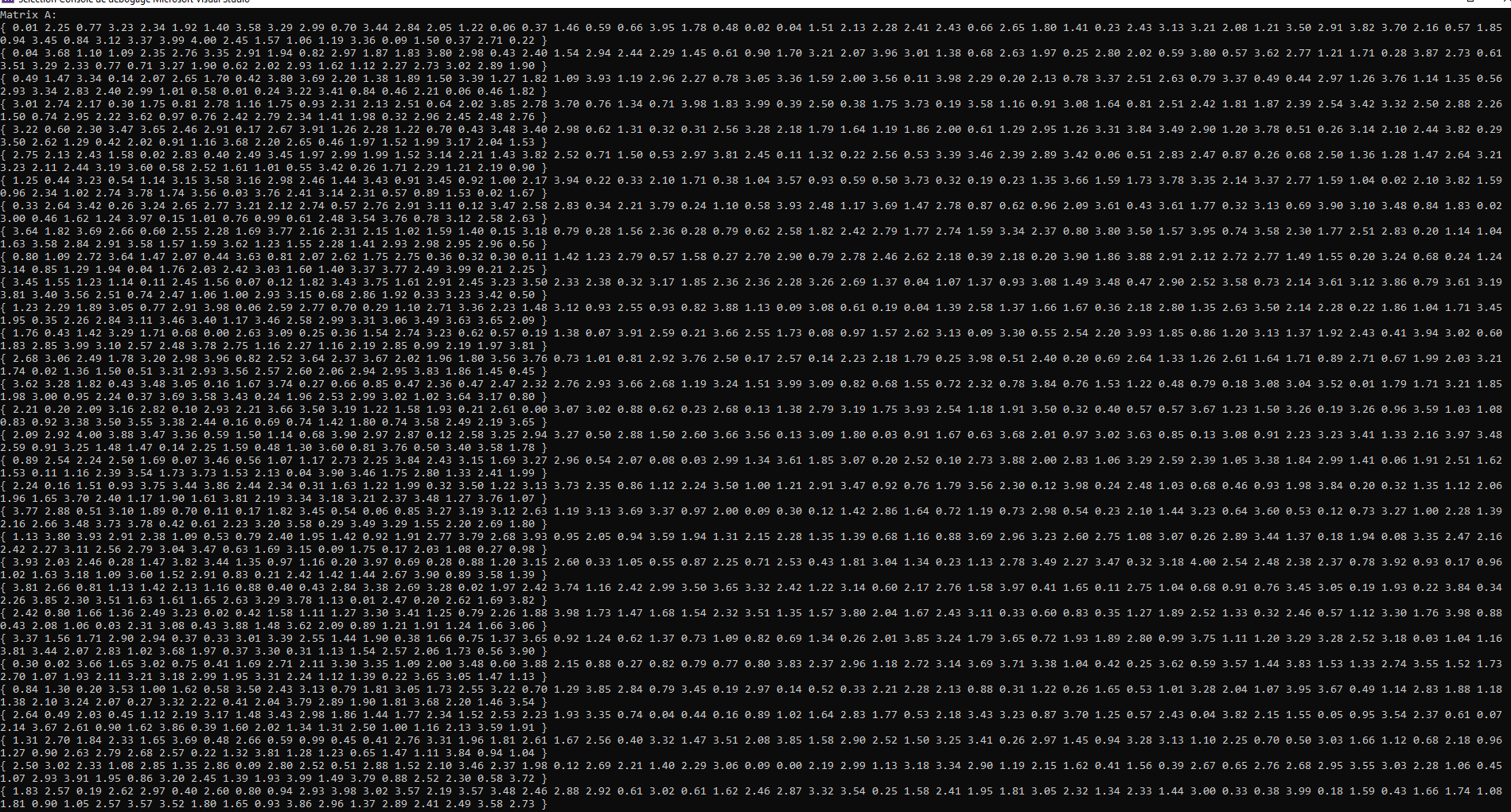


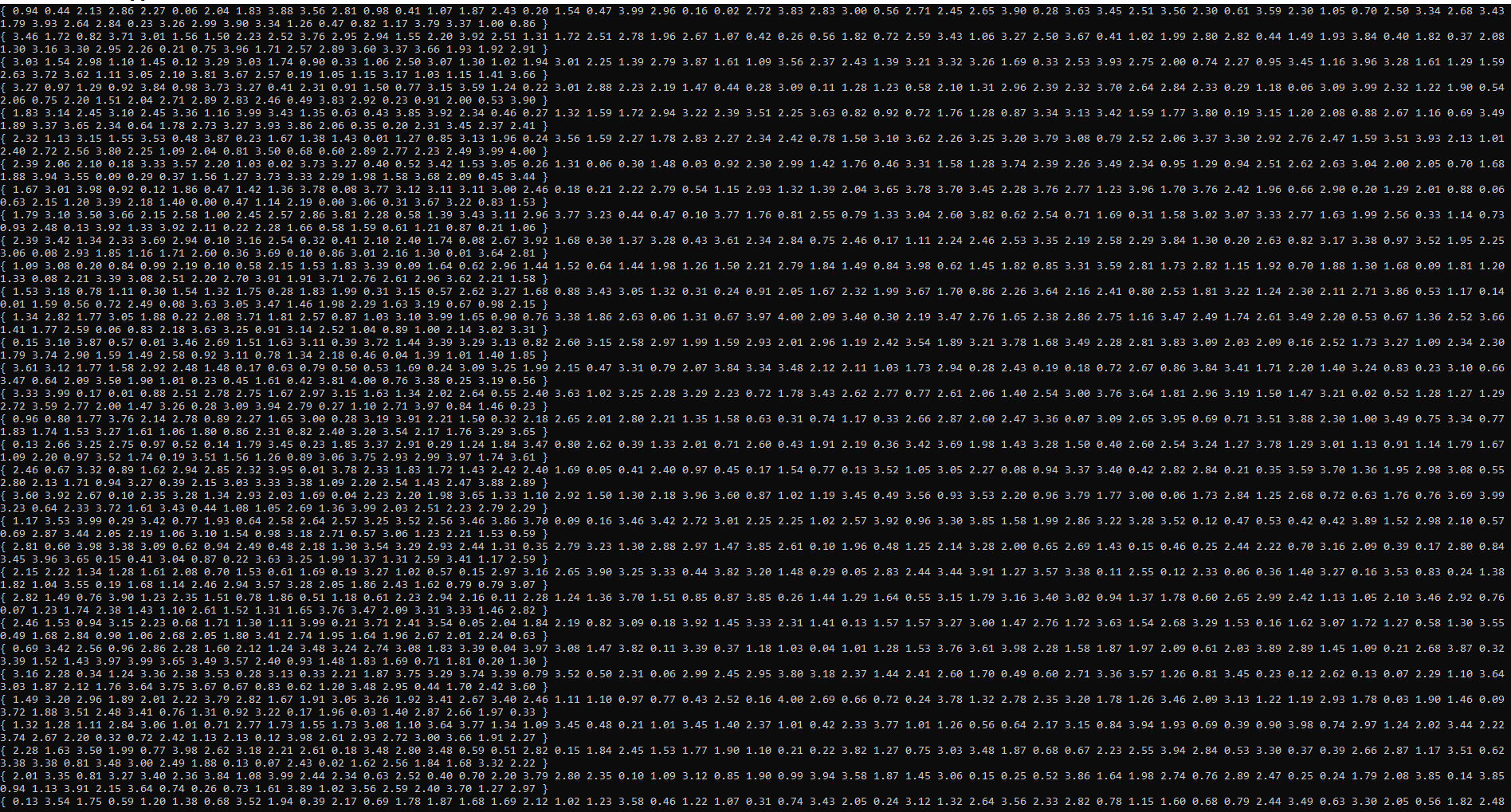
Comme expliqué en commentaire, cette configuration permet de retrouver une organization de la matrice de taille 64 par 64.

Voici la fonction de Kernel:



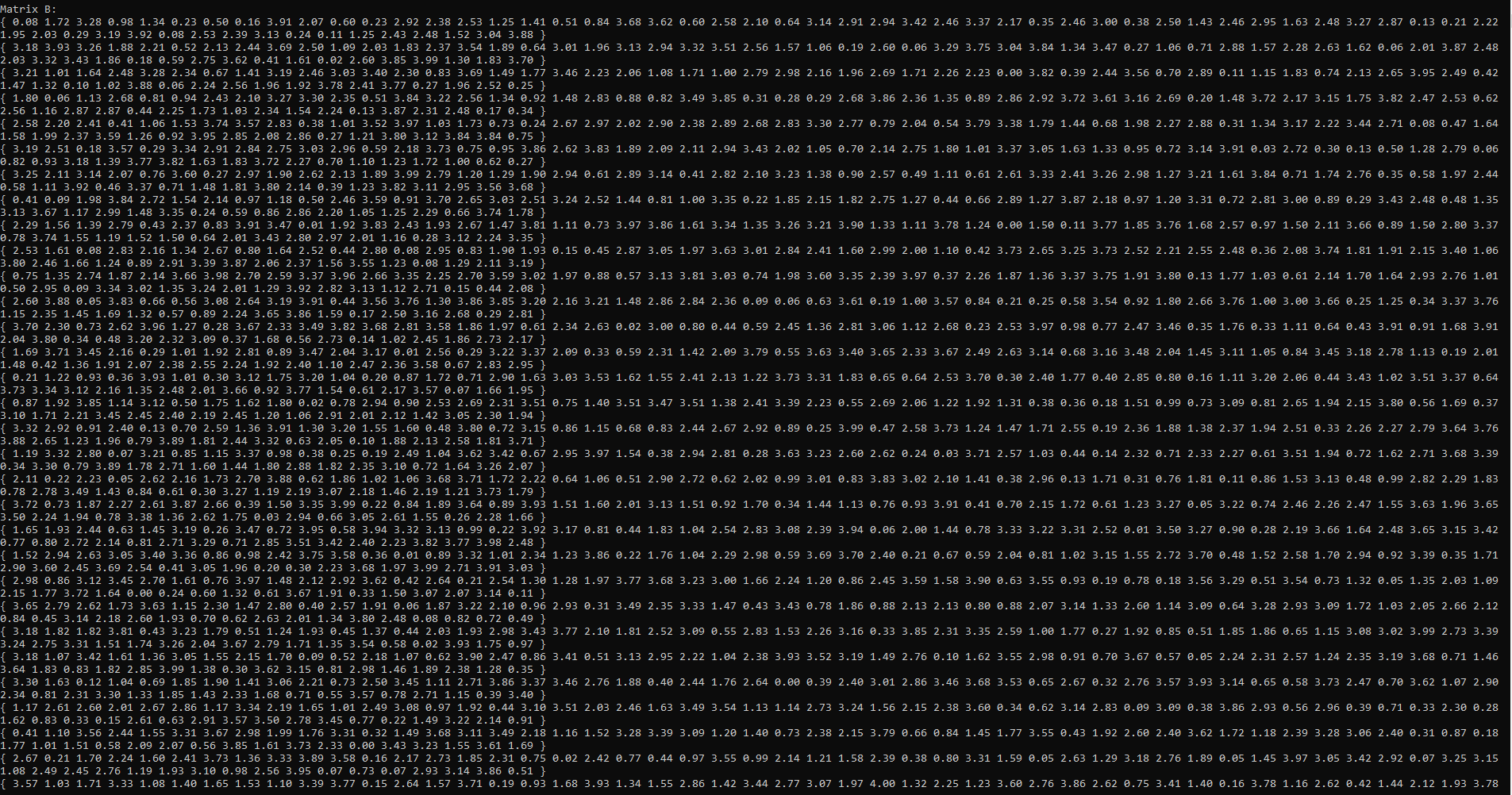
Voici la matrice A où chaque rangé est délimité par des accolades :

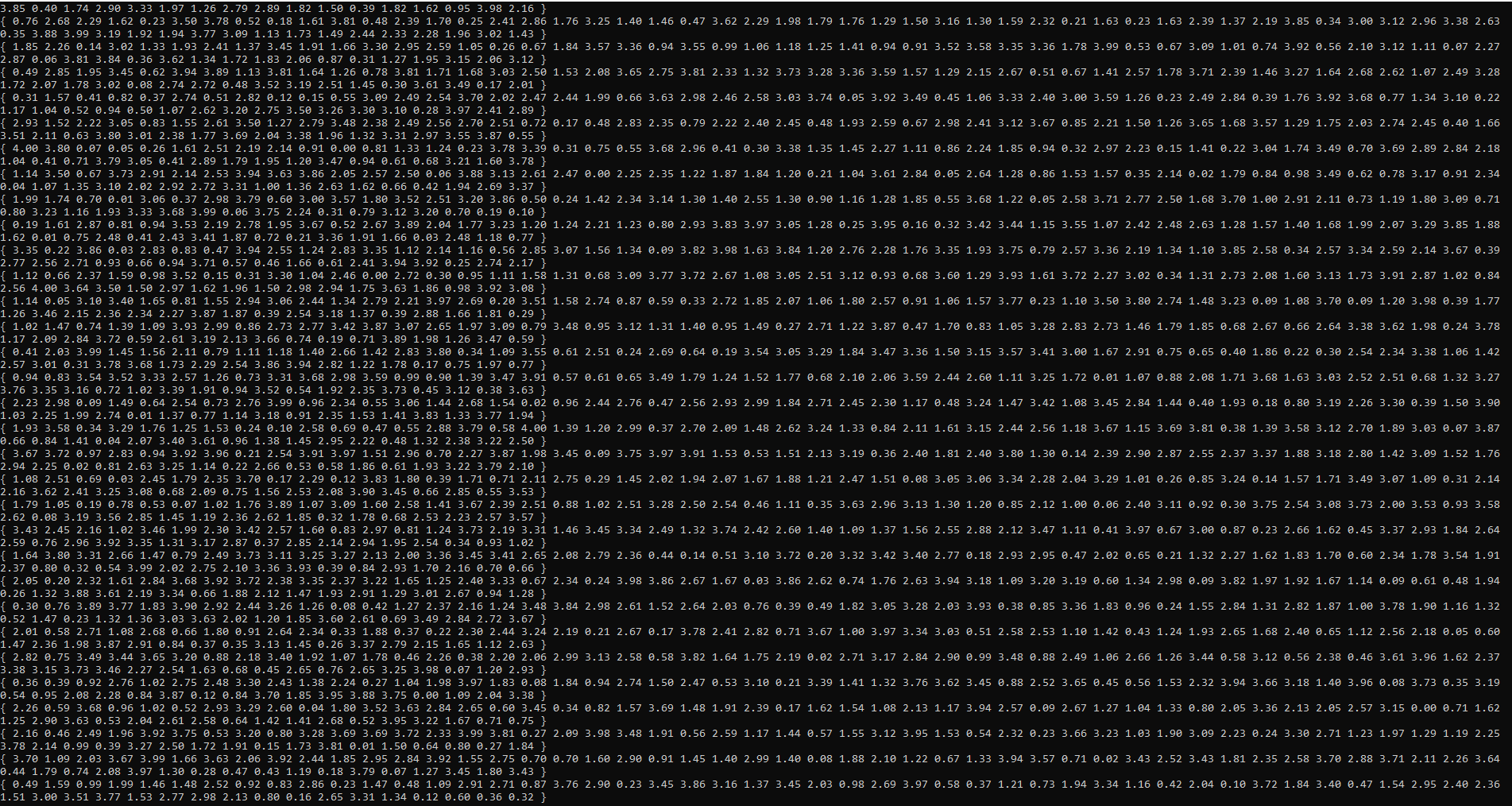


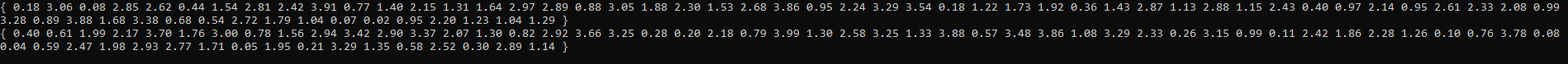




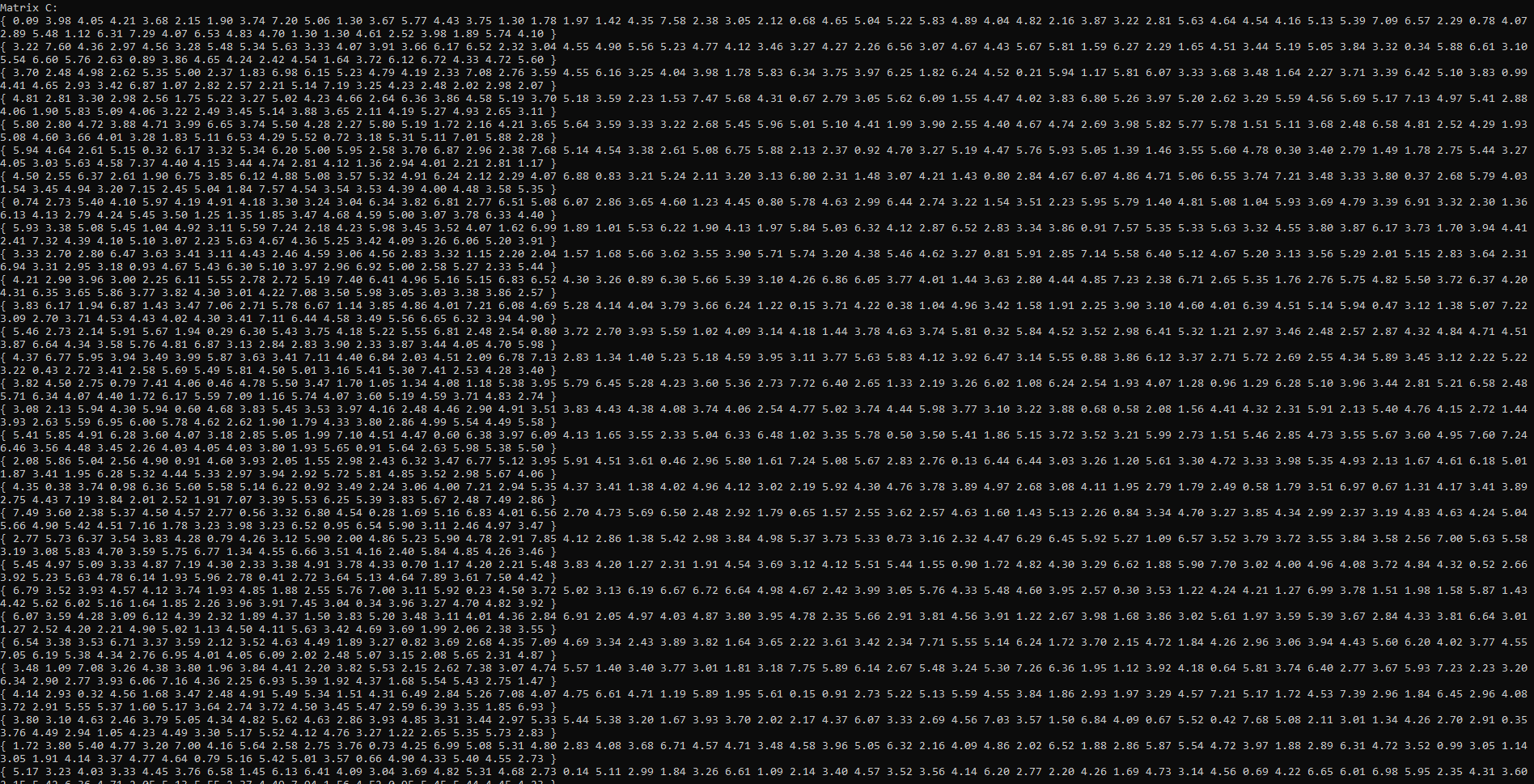
Voici la matrice B où chaque rangé est délimité par des accolades :

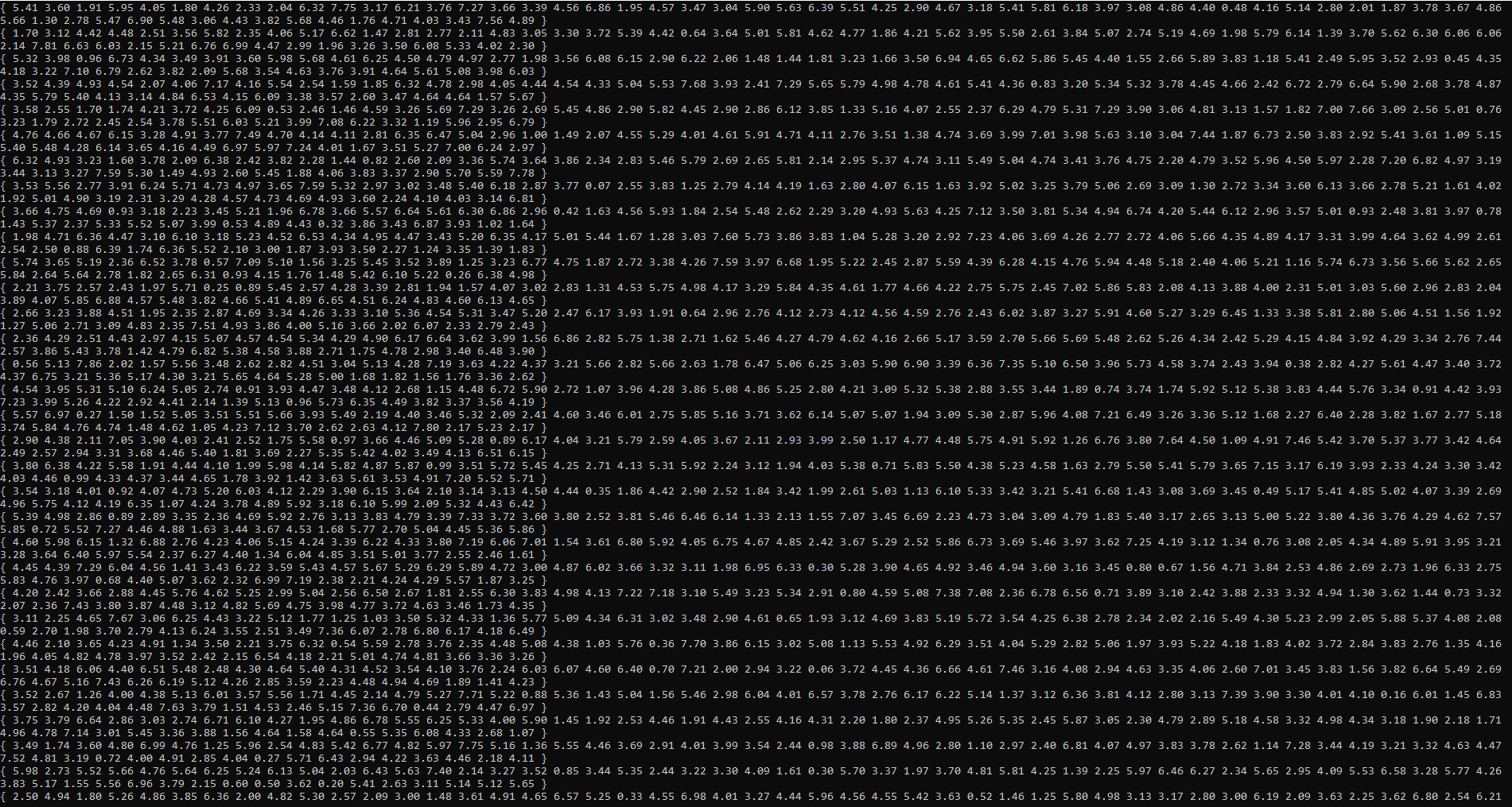


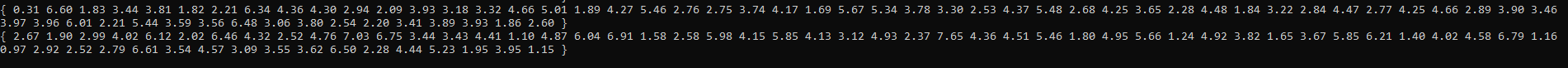




Voici la matrice C où chaque rangé est délimité par des accolades :



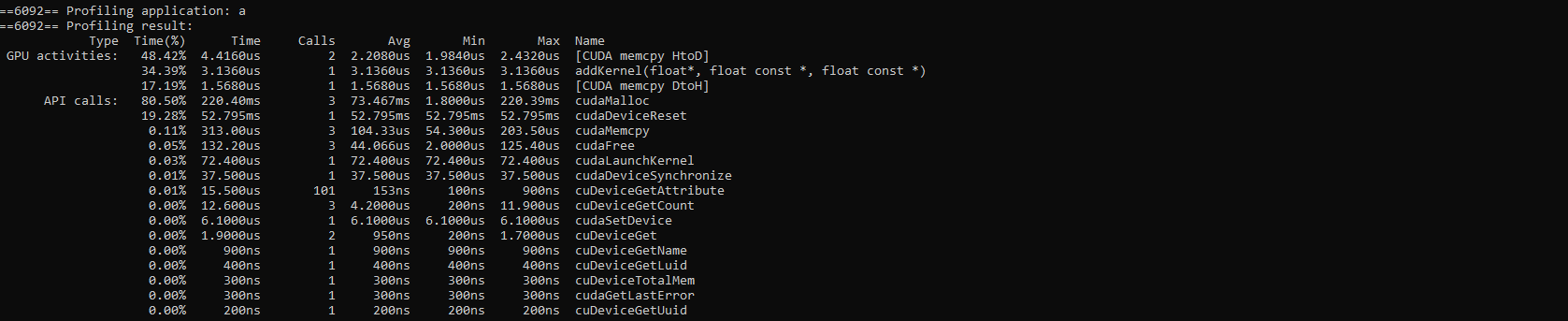




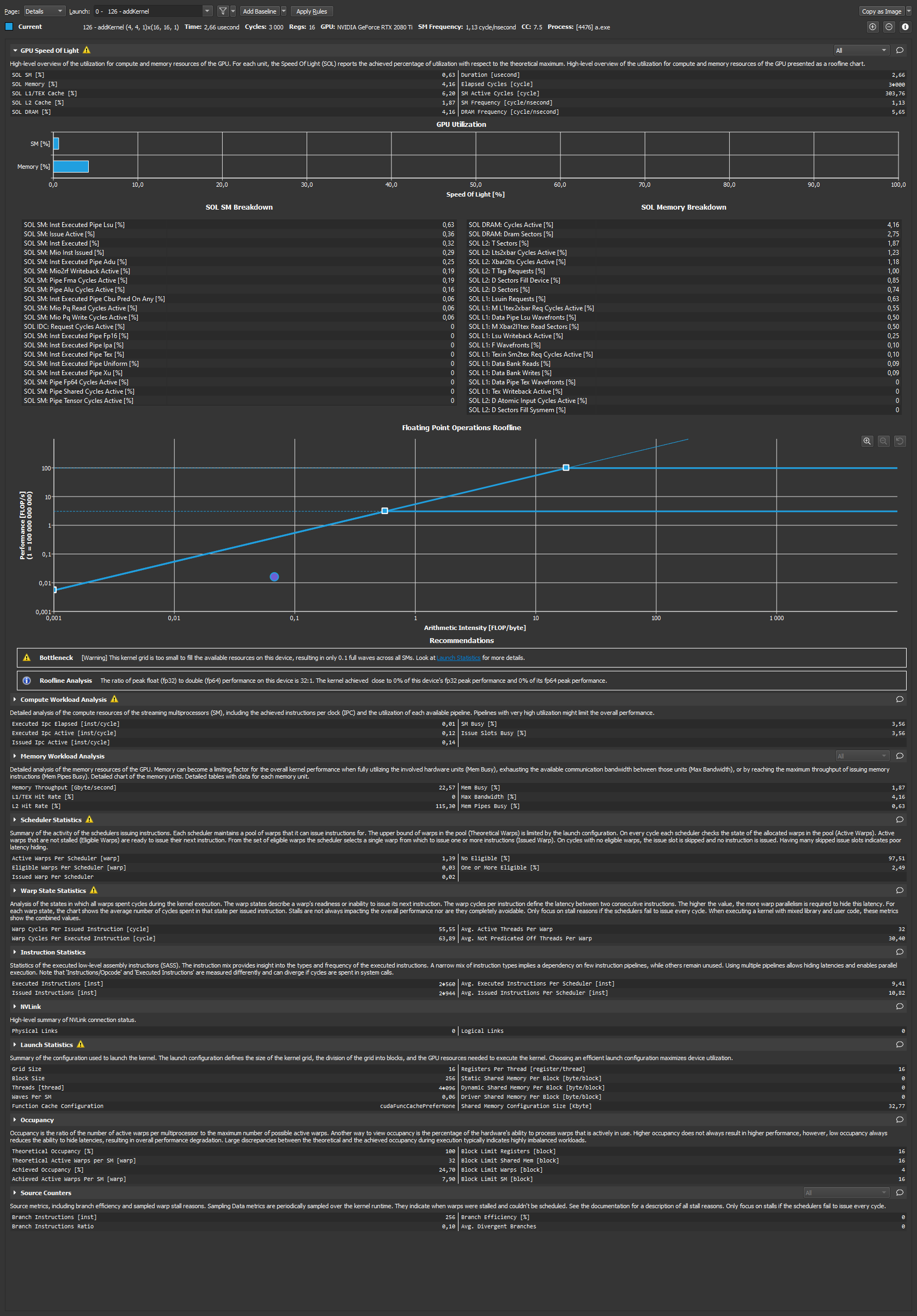
Profiling : cudamemcheck :

Aucune erreur de mémoire (out of bounds, mémoire non intialisé, etc)

Nvprof :



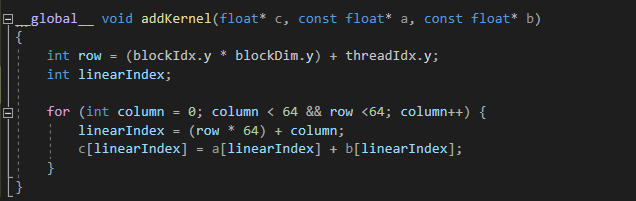
La fonction addKernel a pris en motene 3.1360us pour éxecuter.



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.

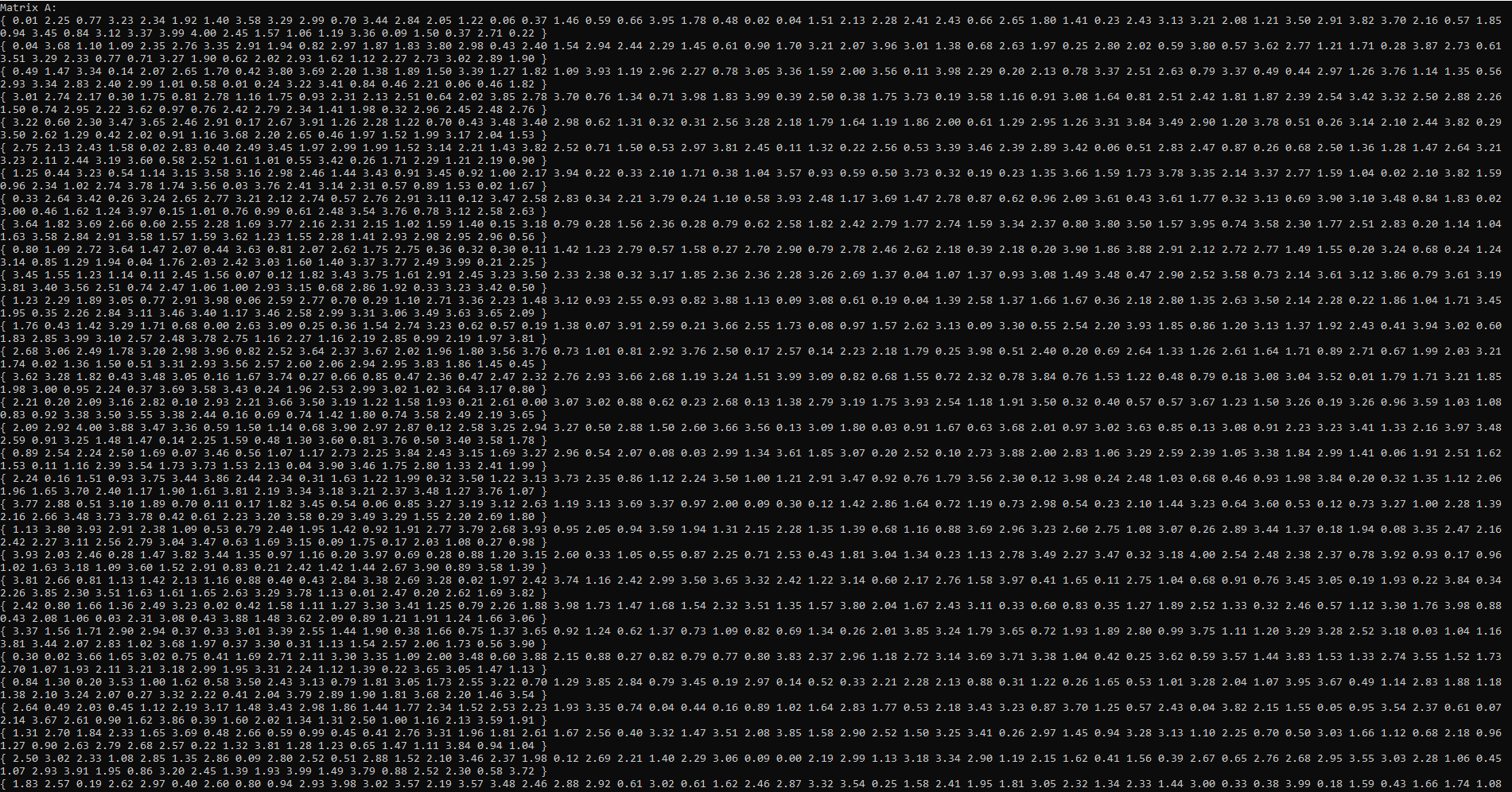
La host-stub function est inchagé de la question b.

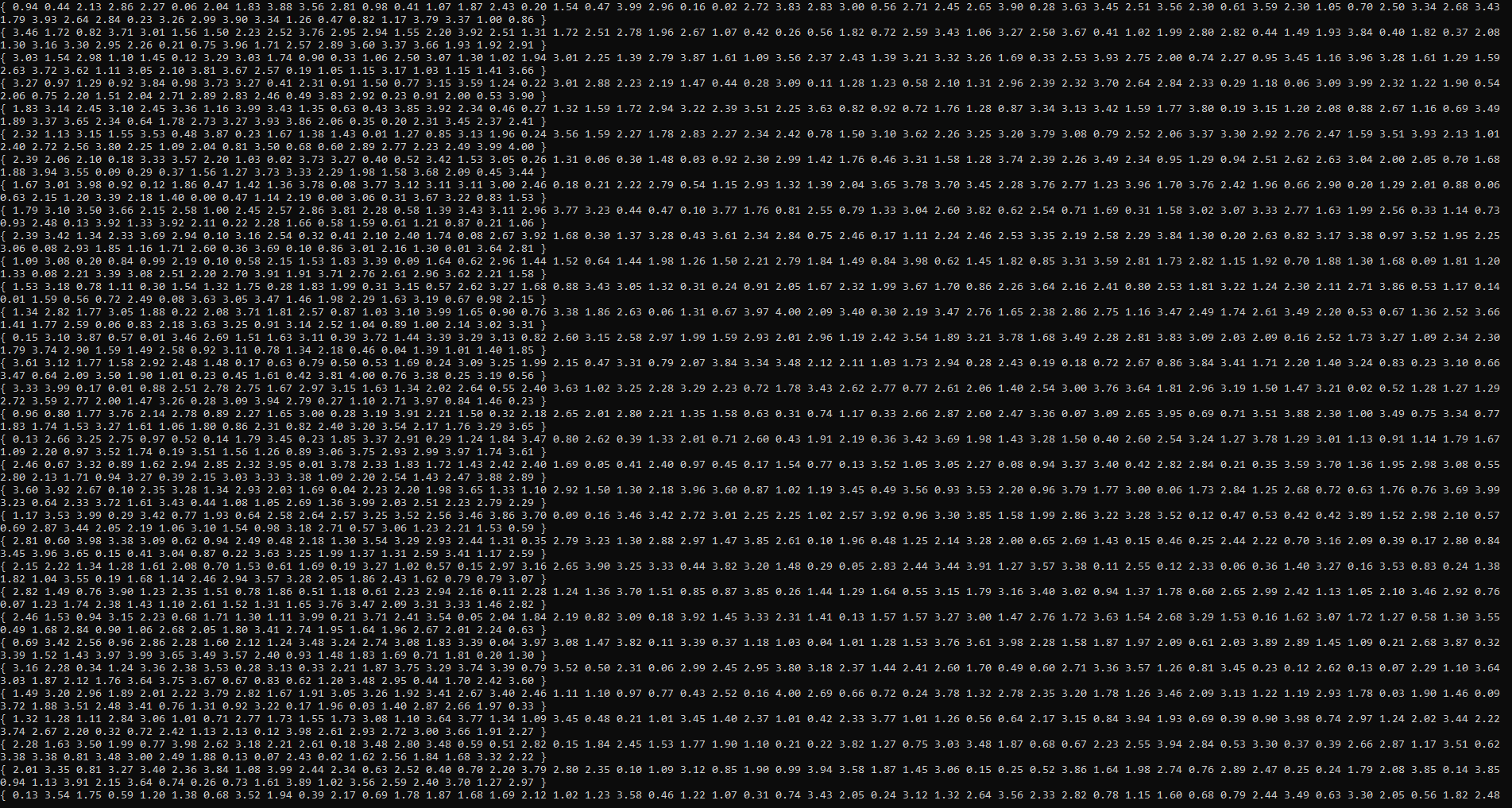
Voici la nouvelle fonction de Kernel:



Chaque thread s’exécutera sur une rangée et la rangée est calculé dans la première ligne. Par la suite, dans la boucle « for », chaque élément dans la rangée est calculé. Je m’assure de ne pas dépasser les dimensions de la matrice en mettant comme condition dans la boucle que la colonne et la rangée ne doivent être supérieur à 64. L’index de la matrice 1D final est calculé dans le même style que dans la question B.

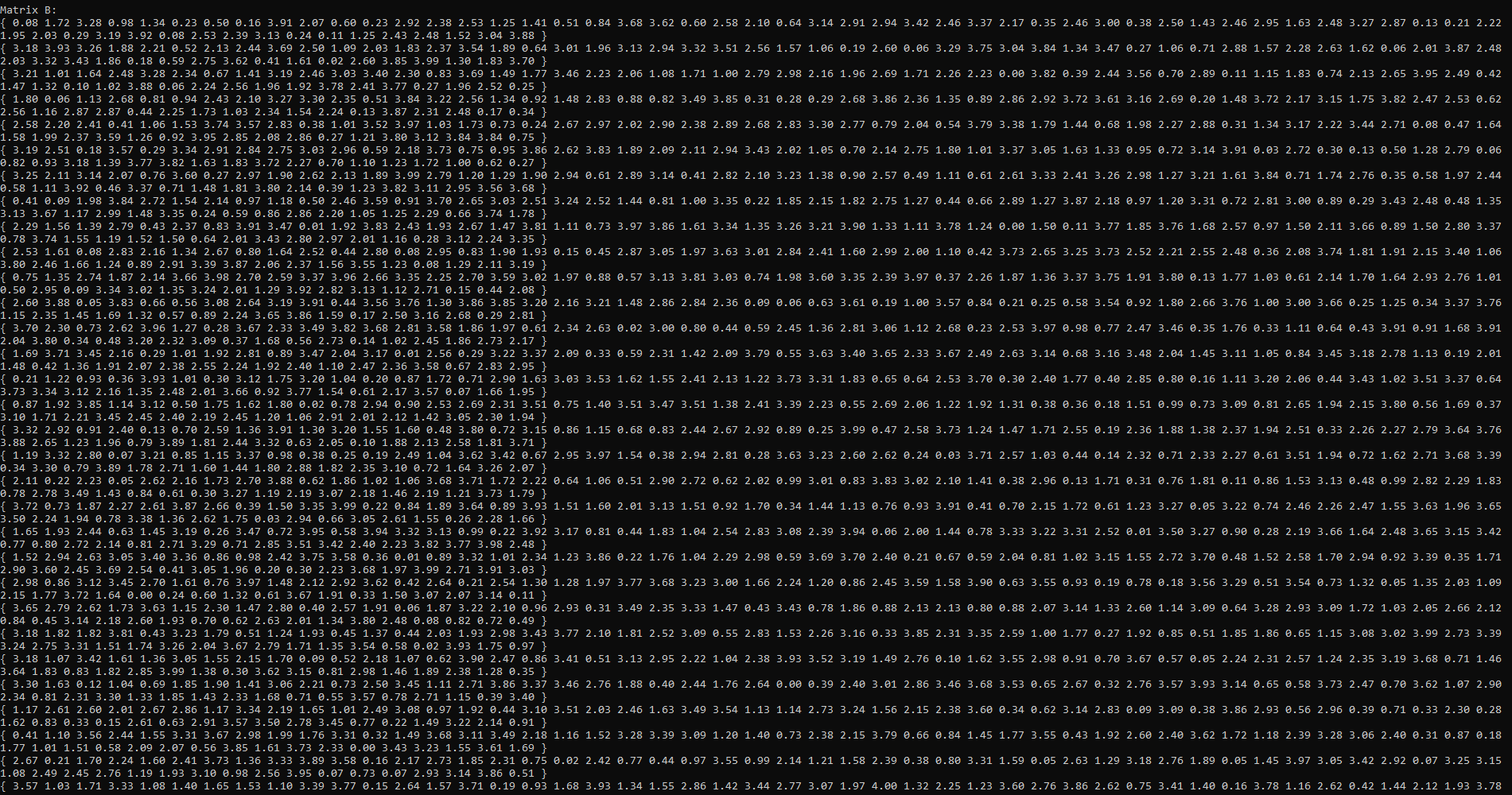
Voici la matrice A où chaque rangé est délimité par des accolades :

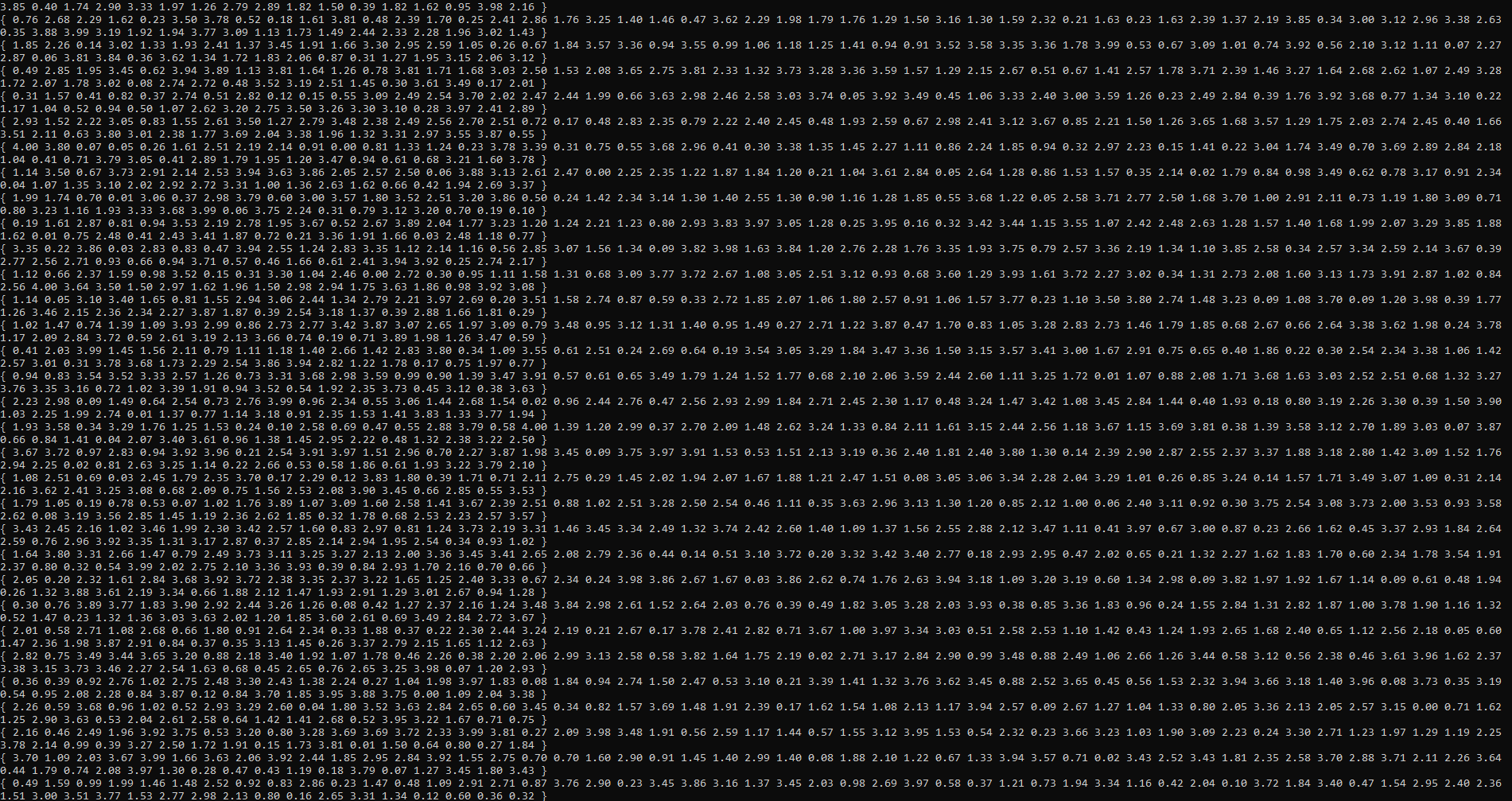


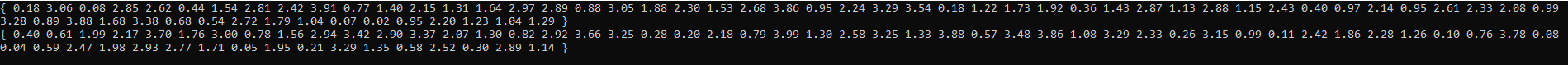




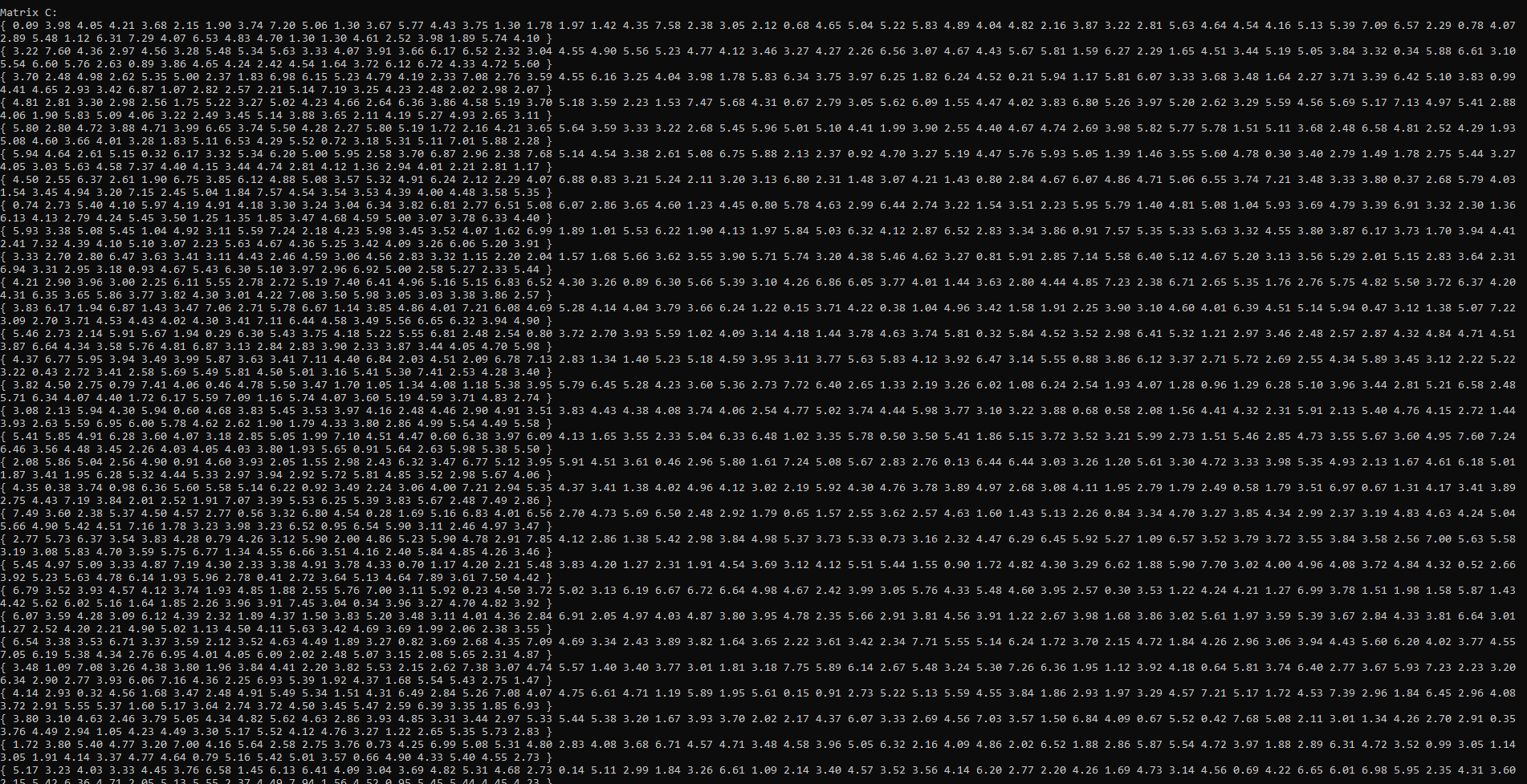
Voici la matrice B où chaque rangé est délimité par des accolades :

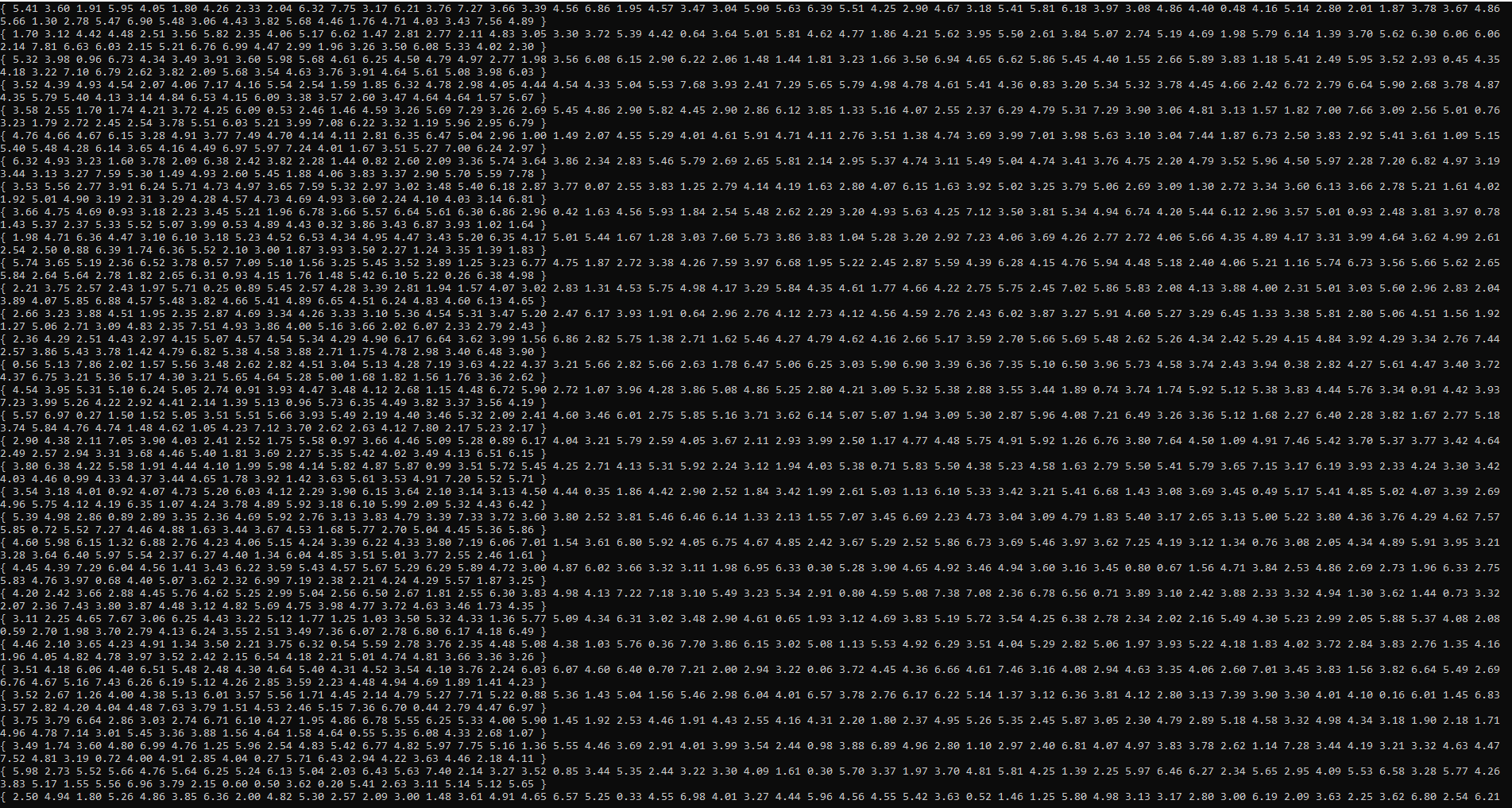


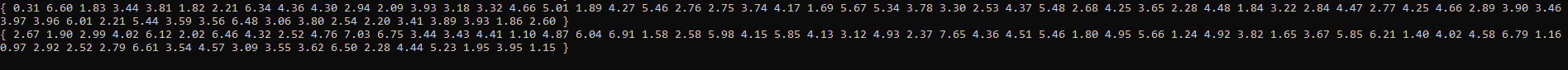




Voici la matrice C où chaque rangé est délimité par des accolades :



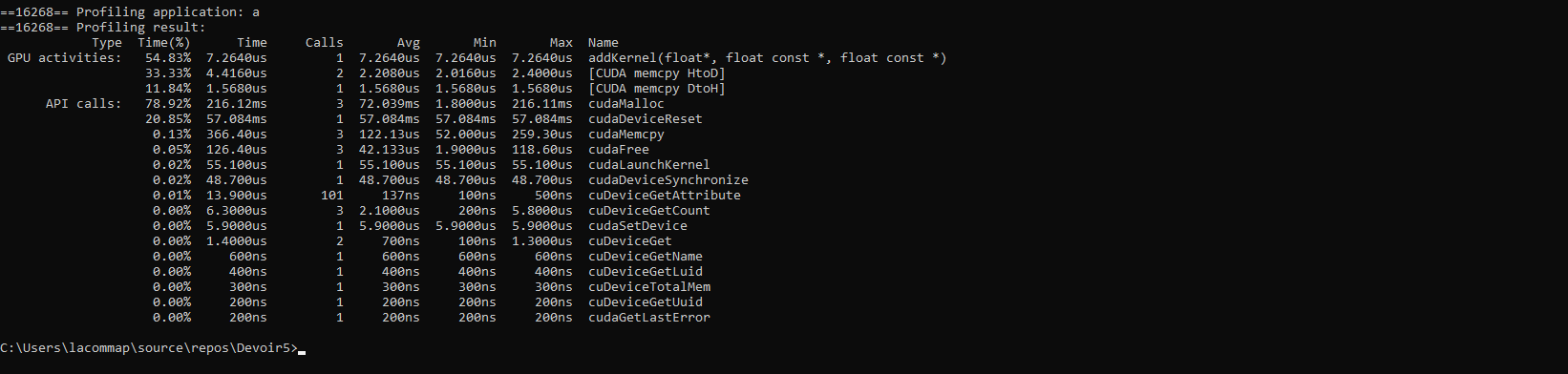




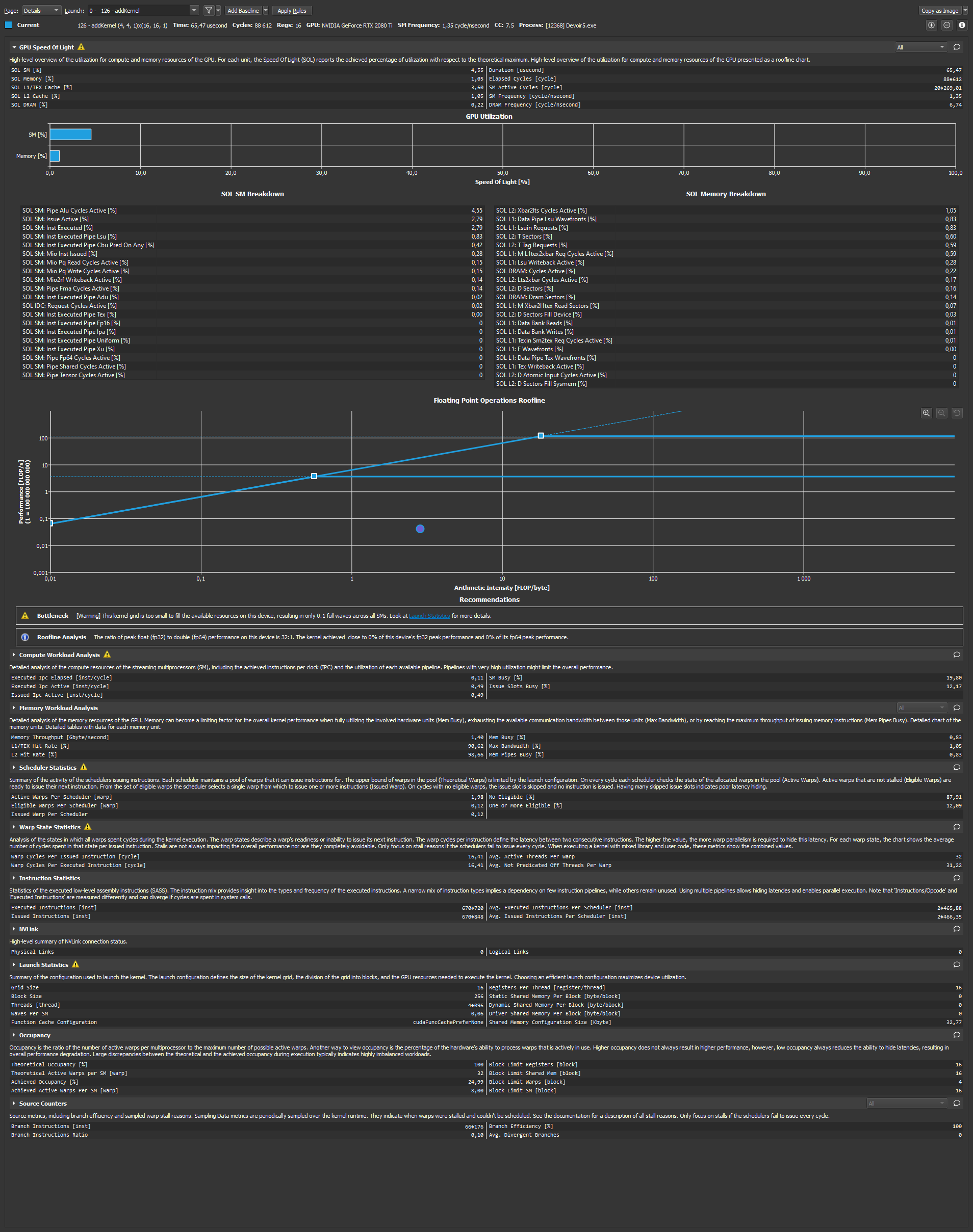
Profiling : cudamemcheck :

Aucune erreur de mémoire (out of bounds, mémoire non intialisé, etc)

Nvprof :



La fonction addKernel a pris en moyenne 7.2640us pour éxecuter.



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.
2. Analysez les avantages et inconvénients de chaque kernel ci-dessus. Pour A à E:
3. Validez que le kernel est fonctionnel et effectue le bon calcul en utilisant un simple programme en CUDA C.
4. Profilez votre kernel et discutez des résultats

**Méthodologie :** Suivez les mêmes étapes que dans le Pré-Lab 3 pour compiler et profiler vos kernels.

**Rapport :** Pour l’exercice 1, vous aurez donc 3 kernels à écrire, profiler et comparer. Dans le rapport, inclure des captures de la host-stub function et des 3 kernels **que vous expliquerez**. De plus, inclure des captures démontrant que chaque kernel effectue le bon calcul. Par la suite, effectuer le profiling des 3 kernels puis les comparer à l’aide de **nvprof** et **NSight**.

## Exercice 2

Une multiplication matrice-vecteur prend une matrice B en entrée et un vecteur C et produit un vecteur A en sortie. Chaque élément du vecteur de sortie A correspond au produit scalaire d’une ligne de la matrice B et de C, i.e., A[i] = Σj (B[i][j] x C[j]). Par souci de simplicité, nous allons uniquement utiliser des matrices carrées dont les éléments sont des single-precision floating- point.

## Pour des matrices de taille 64x64

1. Écrire un kernel effectuant la multiplication matrice-vecteur et une fonction “host-stub ” qui sera appeler avec 4 paramètres : Pointeur vers le vecteur de sortie, pointeur vers la matrice d’entrée, pointeur vers le vecteur d’entrée, dimensions. **Chaque thread calculera un élément du vecteur de sortie.**
2. Validez que le kernel est fonctionnel et effectue le bon calcul en utilisant un simple programme en CUDA C.
3. Profilez votre kernel et discutez des résultats

**Méthodologie :** Suivez les mêmes étapes que dans le Pré-Lab 3 pour compiler et profiler votre kernel.

**Rapport :** Pour l’exercice 2, inclure des captures de la host-stub function et du kernel que vous expliquerez. De plus, inclure des captures démontrant que le kernel effectue le bon calcul. Par la suite, effectuer le profiling du kernel et discutez des résultats.

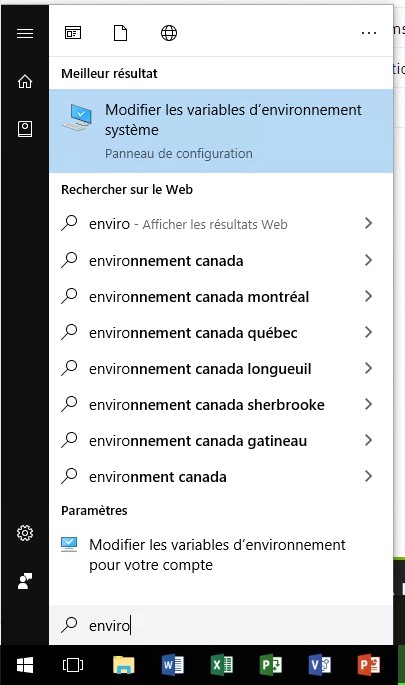
**REFERENCES**

[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).

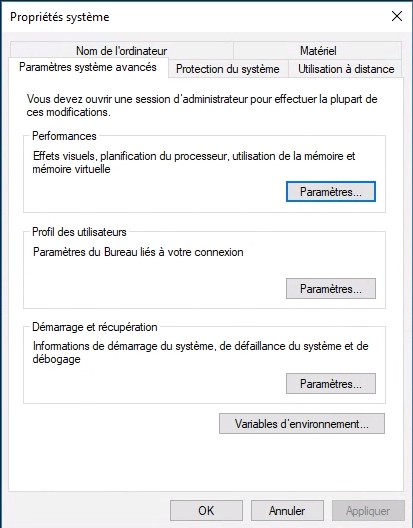
**APPENDIX A**

1. Add “cl.exe” to System Environment Variables

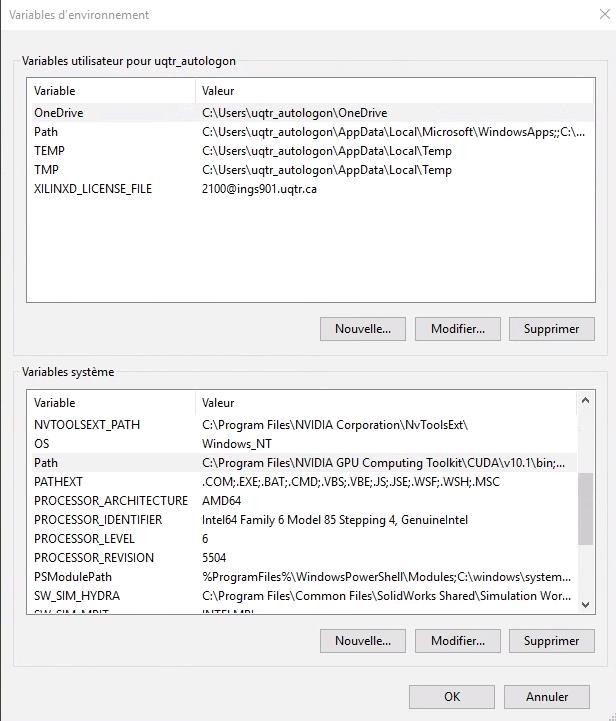
1.1.



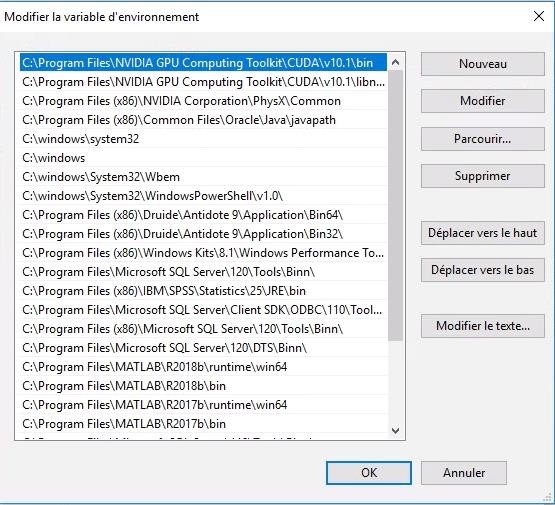
* 1. Click on *Variables d’environnement*



* 1. Select Path and click *Modifier*



* 1. Click on *Nouveau* and add this path C:\Program Files (x86)\Microsoft Visual Studio 14.0\VC\bin



* 1. Click ok to close the 3 windows