SCALIAN

INSA
RENNES

TP GPU CUDA INSA 2021

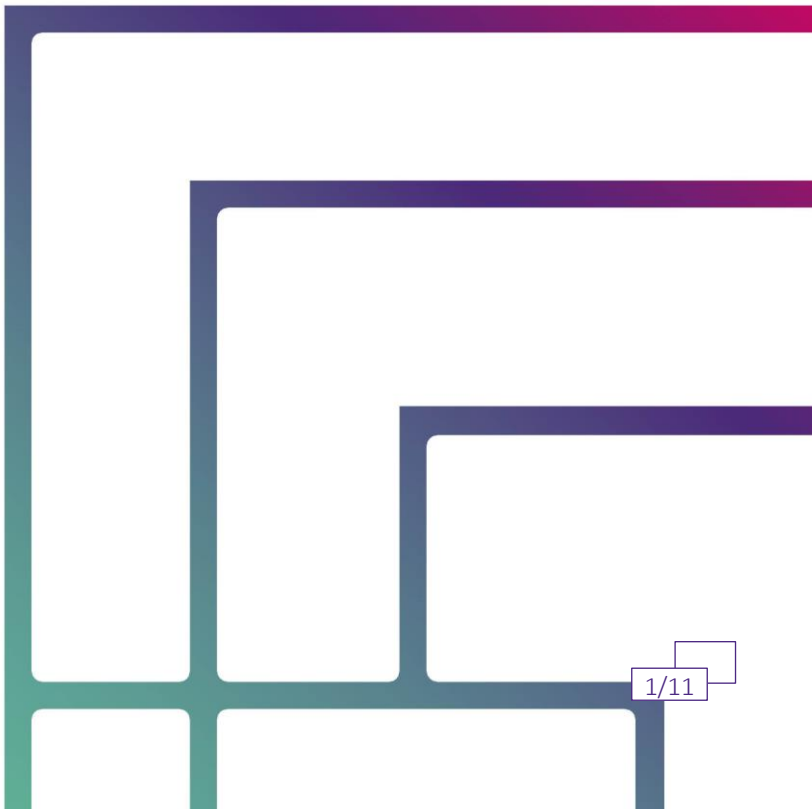


Table des matières

1 HELLO WORLD !

Le premier exemple proposé ici consiste à afficher la chaine de caractères « Hello World ! » depuis le GPU. Ce programme minimal introduit la syntaxe spécifique à CUDA :
• Le mot clé « __global__ » indique qu'une fonction est exécutée sur le GPU et appelée depuis le CPU. Les fonctions qui ont ce mot clé sont appelées « kernels » (ou « noyaux » en français).
• La syntaxe « <<<nombre de blocs, nombre de threads par bloc>>> » permet de spécifier la taille de la grille de calcul utilisée par le noyau.

EXERCICE

• Rendez-vous dans le répertoire « TP_INSA/HelloWorld » puis compiler le programme en utilisant le compilateur de Nvidia :
> nvcc --gpu-architecture=sm_50 helloword_example.cu -o helloword
• Exécutez le programme créé et vérifiez que la chaine de caractères est bien écrite en console :
> ./helloword
• Regardez le code et comprendre l'intérêt de la ligne cudaDeviceSynchronize();

2 ADDITION DE DEUX VECTEURS

Cet exercice propose d'additionner deux vecteurs sur GPU et de vérifier le résultat. Pour ce faire, une version CPU est écrite et la version GPU est en partie rédigée. Le principe est simple : deux tableaux de float sont initialisés avec des valeurs par défaut. L'addition est effectuée sur le CPU puis sur le GPU et une comparaison entre les résultats est effectuée.
Les exercices vont vous permettre de remplir le tableau suivant. On pourra constater les différences de performances pour des implémentations plus ou moins optimisées.

Version	Temps d'exécution (ms)	Facteur d'accélération
CPU		1
GPU – 1 bloc / 1 thread		
GPU – 1 bloc / plusieurs thread		

Les sources de ce problème sont dans le dossier « /VectorAdd »

PREMIERE VERSION : CPU

Le fichier « VectorAdd_example.cpp » contient la version C de l'addition de deux vecteurs qu'il faut porter sur GPU.

DEUXIEME VERSION : GPU MONO-THREAD

Une ébauche de code est disponible dans le fichier « VectorAdd_todo.cu ». Des commentaires sont présents dans le code pour vous guider vers les solutions.

• Copiez le fichier « VectorAdd_todo.cu » et nommez la copie « VectorAdd_MonoThread.cu ». Modifiez le fichier « compile » pour compiler le nouveau fichier.

• Utilisez les erreurs dans la console et les commentaires dans le fichier pour implémenter une version qui donne le bon résultat avant de passer à la suite.

Comme montré en Figure 1, l'implémentation que vous avez faite est une solution naïve qui utilise une grille de 1 bloc et 1 thread par bloc c'est-à-dire une version qui utilise le GPU comme un cœur du CPU et traite l'information de façon séquentielle !

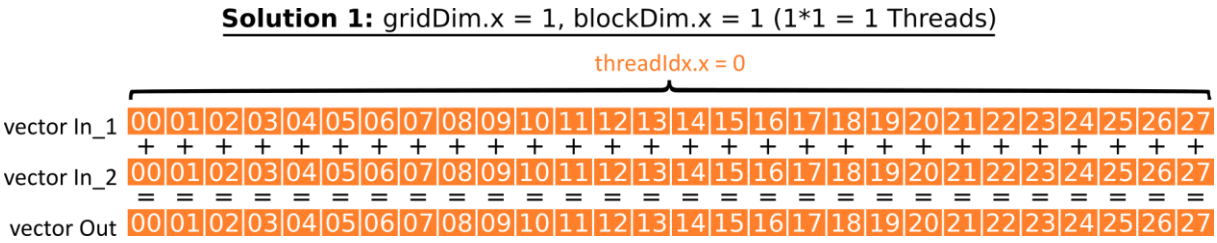


Figure 1 Le thread additionne les deux vecteurs séquentiellement case par case

On peut tout de même faire un profilage rapide avec les commandes suivantes et remplir le tableau ci-dessus :

> /usr/local/cuda/bin/nvprof ./VectorAdd_MonoThread

TROISIEME VERSION : GPU MONO-BLOC ET MULTI-THREAD

Cette deuxième version implémente l'algorithme sur plusieurs threads dans un même bloc. Matériellement, il n'est pas possible de lancer plus de 1024 threads au sein d'un même bloc et cette limitation nous empêche de lancer autant de threads que de valeurs dans le vecteur (la dernière version de l'algorithme expliquera comment contourner ce problème). Nous nous contenterons dans cette partie de répartir le travail sur 256 threads en conservant le même socle algorithmique, c'est-à-dire une boucle pour additionner les deux vecteurs. Pour cela plusieurs étapes sont nécessaires :

• Copiez le fichier « VectorAdd_MonoThread.cu » et nommez la copie « VectorAdd_MultiThread.cu ».

• Modifiez le fichier « compile » pour compiler le nouveau fichier.

• Changer la taille de la grille en entrée, on pourra par exemple partir sur une grille de 1 bloc avec 256 threads par bloc. CUDA donne alors un moyen pratique de se repérer dans la grille :

> threadIdx.x : contient l'indice du thread à l'intérieur du bloc courant
> blockDim.x : contient le nombre de threads dans le bloc courant.

• Changer le kernel « vector_add_thread_cuda » pour utiliser les nouvelles données de bloc et de threads. L'idée est alors la suivante : la boucle d'addition est la même mais l'incrément est différent, et l'indice de départ n'est plus le même selon le numéro du thread du bloc comme le montre la Figure 2.

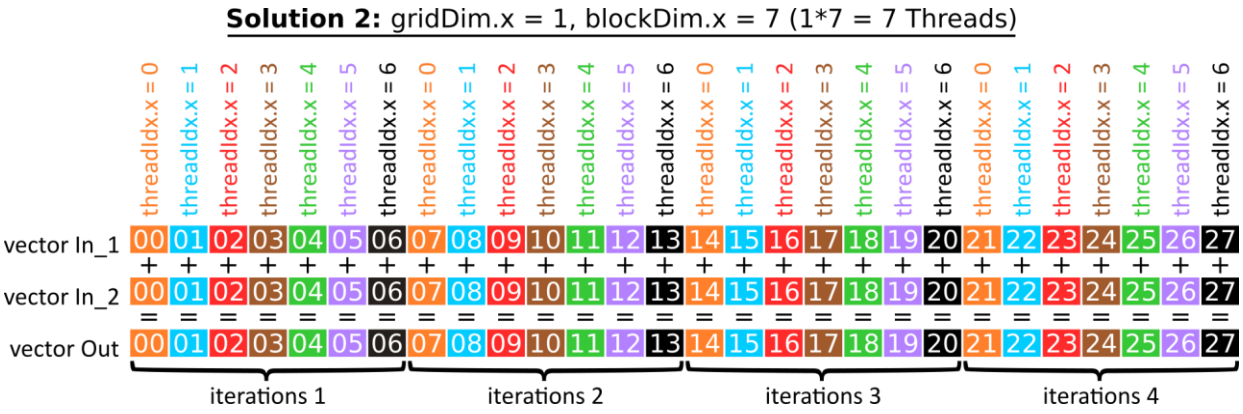


Figure 2. Chaque thread ne calcul qu'une partie de l'addition des deux vecteurs

La solution est disponible dans le fichier « VectorAdd_MultiThread.cu ». On peut là-aussi lancer un test de performance via /usr/local/cuda/bin/nvprof et noter les résultats obtenus en comparaison de ceux issus de « VectorAdd_MonoThread.cu ».
Qu'observe-t-on alors ? En faisant varier le nombre de threads ? (16, 32, 64, 128, 256, 512 ou encore 257...)

QUATRIEME VERSION : GPU MULTI-BLOC ET MULTI-THREAD

Pour finir, l'objectif est de reprendre la version précédente en utilisant cette fois-ci une grille possédant plusieurs blocs et chacun de ces derniers possédant plusieurs threads !
De la même façon que précédemment, l'objectif est de modifier le dernier algorithme d'addition pour utiliser un nombre de blocs différent de 1 et un nombre de thread lui aussi différent de 1. Pour cela plusieurs étapes sont nécessaires :

• Copiez le fichier « VectorAdd_MultiThread.cu » et nommez la copie « VectorAdd_Grid.cu ».
• Modifiez le fichier « compile » pour compiler le nouveau fichier.
• Changer la taille de la grille en entrée. En partant sur 256 threads par bloc, on doit donc avoir au moins N/256 blocs pour avoir N threads (N étant la taille du vecteur en entrée). CUDA donne alors un moyen pratique de se repérer dans la grille de la même façon que dans un bloc : –

> threadIdx.x : contient l'indice du thread à l'intérieur du bloc courant
> blockDim.x : contient le nombre de threads dans le bloc courant.

> blockIdx.x : contient l'indice du bloc courant à l'intérieur de la grille
> gridDim.x : contient le nombre de blocs qui constitue une grille (c'est-à-dire la taille de la grille).
> Formule pour calculer le nombre de bloc : (ceil(N/256))

• Changer le kernel « vector_add_grid_cuda » pour utiliser les nouvelles données de bloc et de threads. Comme montré en Figure 3, l'idée est alors la suivante : chaque thread effectue simplement une addition. Il n'y a donc plus de boucle for ! Cette dernière étant en fait « cachée » dans la découpe en <<<bloc, threads>>>.

De la même façon que précédemment, la solution est disponible dans le fichier « VectorAdd_Grid.cu ». On peut là aussi lancer un test de performance via /usr/local/cuda/bin/nvprof et noter les résultats obtenus en comparaison de ceux issus de « VectorAdd_MonoThread.cu » et «VectorAdd_MultiThread.cu ».

Qu'observe-t-on alors ? On peut remplir le tableau de résultat avec les temps d'exécution pour avoir un comparatif des performances.

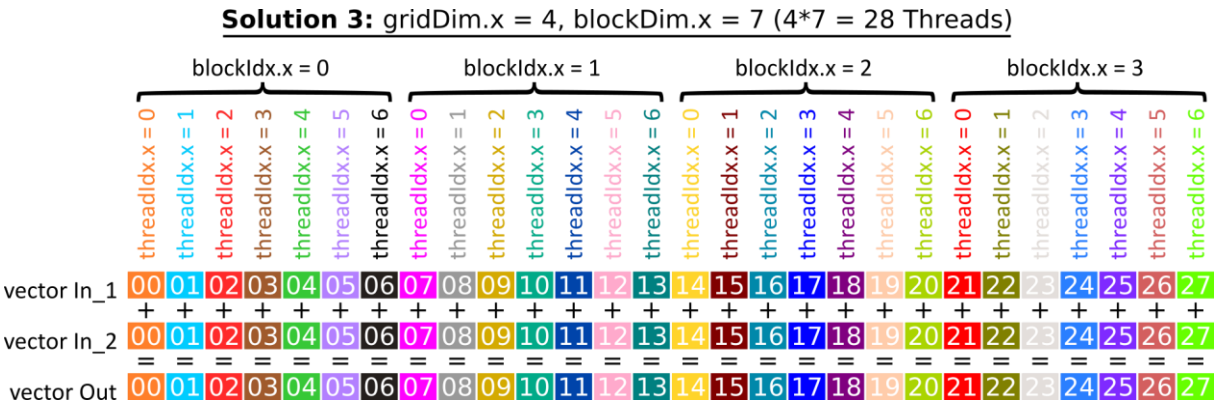


Figure 3. Chaque thread effectue simplement une addition.

3 PERFORMANCES THEORIQUES

Nous allons comparer ici les puissances théoriques des deux processeurs. L'unité de mesure utilisée est le FLOP. Il est défini de la façon suivante :

$$\text{FLOPS} = \text{cœurs} \times \text{fréquence} \times \frac{\text{FLOP}}{\text{cycle}}.$$

COEURS ET FREQUENCE

Récupérez le nombre de coeurs CUDA de votre GPU ainsi que la fréquence d'un coeur. Pour cela, rendez-vous dans le dossier « /Utilities/ », compilez et exécutez « deviceQuery » :

```
> ./compile
> ./deviceQuery
```

Nombre de cœurs		-
Fréquence d'un cœur		GHz

Rapport “FLOP / CYCLES”

Ce rapport est le nombre d’opérations élémentaires qu’un processeur peut réaliser en un seul cycle d’horloge. Sans rentrer dans les détails, cette page donne les valeurs pour quelques CPU et GPU : https://en.wikipedia.org/wiki/FLOPS.
Sachant que votre GPU (Quadro P620 ou K620) possède une architecture “Pascal” ou “Maxwell”, donnez en Giga Flops la puissance de calcul théorique en simple et en double précision de votre GPU :

	Simple précision	Double précision
Flops / cycles		
Flops théoriques (Gflops)		

Pour donner une idée, votre processeur possède 6 cœurs cadencés à 3,10 GHz et un nombre de flops par cycles de 16 ou 32.

	Simple précision	Double précision
Flops / cycles	32	16
Flops théoriques (Gflops)		

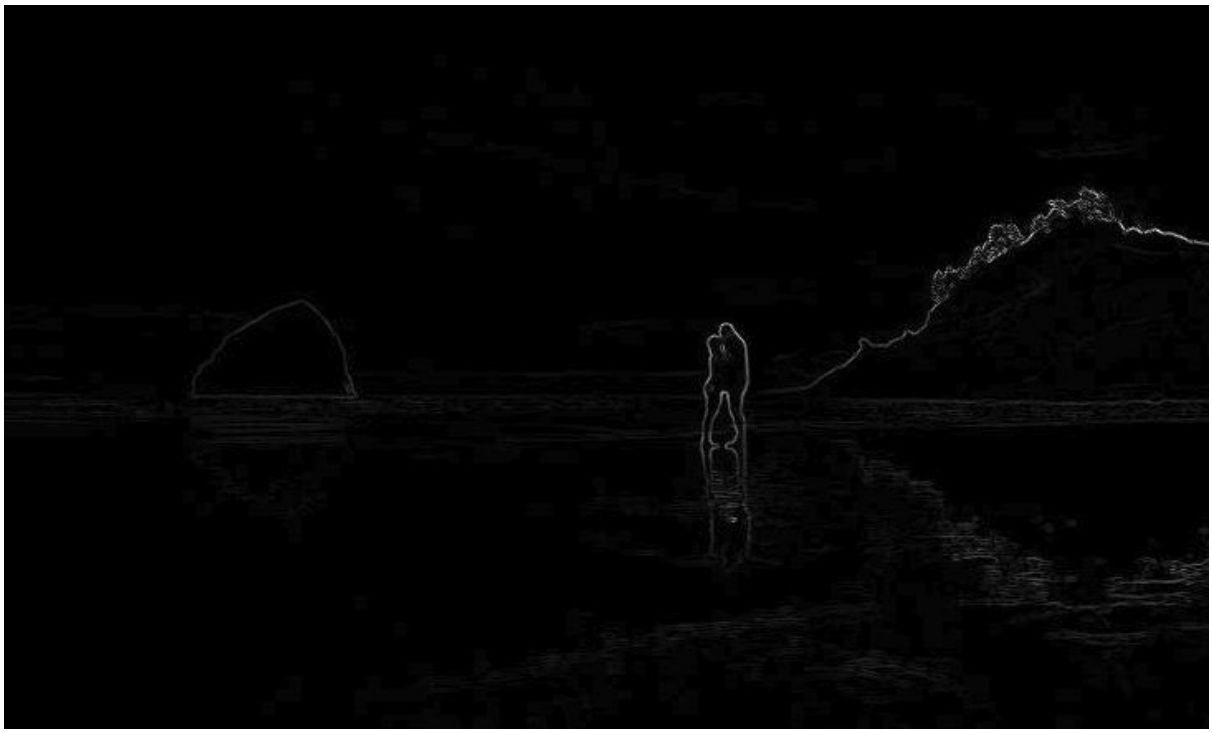
On voit que sur cette machine, un GPU est plus puissant pour traiter des données en simple précision (32 bits) mais n’est pas adapté pour le traitement de données en double précision (64 bits).
Certains GPU sont fabriqués pour avoir de bonnes performances en double précision. C’est le cas de la Testla P100 que vous avez vu en cours.
L’autre exemple dans ce dossier s’appelle « bandwidthTest » permet d’avoir un aperçu de la bande passante pratique des différents bus mémoire du GPU.

4 SEAM CARVING – RECADRAGE INTELLIGENT

THEORIE

Ce dernier exercice s’intéresse à un algorithme de recadrage intelligent d’images.
À partir d’une image rectangulaire, il est par exemple possible de former une image carrée en gardant les éléments importants de la scène. On peut voir le résultat d’un recadrage sur les captures suivantes :

Ici, on a supprimé 350 “chemins de pixels de basse énergie” de la photo de gauche pour produire la photo de droite.
Le principe général est le suivant :
• Dans un premier temps, on passe un algorithme de détection de contours pour repérer les zones d’intérêt de l’image
• Ensuite, un autre algorithme détecte les chemins de basse énergie et les retire de l’image originale
Des implémentations de cet algorithme sont disponibles sur github. Nous avons choisi celle-ci, qui a l’avantage de ne faire appel à aucune bibliothèque tierce :
https://github.com/avik-das/seam-carver
Cette implémentation utilise uniquement le CPU pour réaliser les calculs. L’objectif de cet exercice est d’implémenter une version GPU de la première étape (calcul de la “matrice d’énergie”) afin d'accélérer les calculs.
Sur l’exemple précédent, la matrice est la suivante :



Afin de visualiser ce qui se passe, nous avons créé une vidéo “seam-carving.mp4” qui montre sur l'exemple ci-dessus les suppressions successives des chemins de basse énergie. Pour information, la création de ce film a été faite grâce à la commande FFMPEG suivante :

```
> ffmpeg -framerate 25 -i img-seam-%04d.jpg -vf
"scale=w=1280:h=720:force_original_aspect_ratio=1,pad=1280:720:
(ow-iw)/2:(oh-ih)/2" seam_carving.mp4
```

Du code commenté permet de générer les images utilisées.

EXERCICE

• Regarder la vidéo pour comprendre ce que fait l’algorithme
• Compiler le code en utilisant “make”
• Essayez les exemples suivants et relevez les temps de calcul

```
> ./seam-carver ./castel.jpg . 600
> ./seam-carver ./landscape1.jpg . 300
> ./seam-carver ./landscape2.png . 200
```

Ou encore un exemple qui supporte moins bien le recadrage...

```
> ./seam-carver ./Lenna.png . 150
```

Temps de calculs pour quelques exemples	Temps de calcul CPU (s)	Temps de calcul GPU (s)
castel.jpg . 600		
landscape1.jpg . 300		
landscape2.png . 200		

Il s'agit ici d'écrire la version GPU des fonctions "compute_energy_cpu" et "energy_at_cpu" du fichier "seam-carver.cu".
Ces fonctions s'appellent "compute_energy" et "compute_energy_kernel". Le but est de compléter les "TODO" qui se trouvent dans le code.
Pour vérifier vos résultats, utilisez la commande "diff" pour comparer votre image générée avec une image de référence.

- Générer une image de référence avec la version CPU ;
- Donnez un nom explicite à ce fichier : "nom de l'image_ref_CPU.jpg" ;
- Générer la même image avec votre version GPU ;
- Comparer les deux fichiers en utilisant la commande "diff" : diff -s ./landscape1_300_CPU.jpg ./img.jpg
- Vous pouvez de façon optionnelle écrire un script qui génère l'image GPU puis teste le résultat

Vous avez réussi ? Vous pouvez maintenant tenter d'optimiser votre code en regardant ce qui prend du temps avec la commande "/usr/local/cuda/bin/nvprof".

REMARQUES

Jusqu'à présent, nous avions défini uniquement des blocs et grilles à une seule dimension. Il est en fait possible d'utiliser jusqu'à trois dimensions en définissant :

```
> dim3 block_size(b_width, b_height, b_depth);
> dim3 grid_size(g_width, g_height, g_depth);
> my_kernel<<<grid_size, block_size>>>(my_params) ;
```

Cela permet de simplifier le code en utilisant un nombre dimensions égal à celui des données à traiter. Ainsi, dans exercice on peut utiliser deux dimensions dont les tailles sont calibrées selon la hauteur et la largeur de l'image 2D. De cette manière, chaque Thread pourra déduire les coordonnées du pixel qui lui est assigné en utilisant :

> threadIdx.x, threadIdx.y, ou threadIdx.z : contient l'indice du thread à l'intérieur du bloc courant respectivement sur l'axe largeur, hauteur ou profondeur.

- blockDim.x, blockDim.y, blockDim.z : contient le nombre de threads dans le bloc courant respectivement sur l'axe largeur, hauteur ou profondeur.
- blockIdx.x, blockIdx.y, blockIdx.z : contient l'indice du bloc courant à l'intérieur de la grille respectivement sur l'axe largeur, hauteur ou profondeur
- gridDim.x, gridDim.y, gridDim.z : contient le nombre de blocs qui constitue une grille (c'est-à-dire la taille de la grille) respectivement sur l'axe largeur, hauteur ou profondeur.

5 FRACTALE DE MANDELBROT (FACULTATIF)

THEORIE

En mathématiques, l'ensemble de Mandelbrot est une fractale définie comme l'ensemble des points c du plan complexe pour lesquels la suite de nombres complexes définie par récurrence par la relation suivante est bornée.

$$\begin{cases} z_0 = 0 \\ z_{n+1} = z_n^2 + c \end{cases}$$

Le but de cet exercice et de donner une représentation graphique (dans le plan complexe) d'une telle suite avec plusieurs valeurs initiales données. Pour ce faire, on attribue à z_0 une valeur qui dépend de la position du pixel courant, et on procède aux différentes itérations. Il est possible de caractériser l'ensemble de Mandelbrot par la condition suivante qui est plus adaptée à un calcul numérique : « Si la suite des modules des (z_n) est strictement supérieure à 2 pour un certain indice alors, cette suite est croissante à partir de cet indice, et elle tend vers l'infini. ».

Ainsi, l'algorithme pour dessiner une image en niveau de gris sur 8 bits pourra être le suivant :

- Pour chaque pixel i,
- Fixer une valeur pour z_0 en rapport avec l'indice i
- Itérer au maximum 256 fois, tant que $|z_n| \leq 2$ faire $z_{n+1} = z_n^2 + z_0$
- Dès que $|z_n| > 2$ ou bien que le nombre maximale d'itérations est atteint, stocker le numéro de l'itération courante dans un vecteur à l'indice i.

EXERCICE

Le code « **mandelbrot_cpu.cpp** » fournit une référence écrite en C++. Le but de l'exercice est d'adapter ce code pour l'exécuter sur GPU.
Pour ce faire on peut partir du fichier « **mandelbrot_gpu_todo.cu** » et compléter le code (voir les TODO dans le code). L'algorithme est le même que dans la version C++, la différence majeure se situant dans la boucle for qu'il est possible de faire disparaître pour chaque thread sur le GPU effectuant le calcul pour un pixel.
Pour une image de bonne qualité, on peut par exemple partir sur une taille 800 x 800 pixels.
Vérifier le résultat du code en affichant la fractale.

```
> display ./output_CPU.pgm
```

Pour effectuer le calcul directement avec des nombres complexes comme sur CPU, on peut utiliser sur CUDA les méthodes et éléments suivants :

- `cuDoubleComplex z = make_cuDoubleComplex(real, imag);`
- `cuCAdd(z1, z2)` pour additionner les nombres complexes `z1` et `z2`
- `cuCmul(z1, z2)` pour multiplier les nombres complexes `z1` et `z2`
- `cuCabs(z)` pour calculer la norme de `z`

Il ne reste plus qu'à exécuter le code sous CPU et GPU et observer les temps de calcul pour une image de même taille.

6 ANNEXE 1 - INSTALLATION DE L'ENVIRONNEMENT

Cette annexe décrit comment installer CUDA sous environnement Linux (debian, ubuntu etc.)
L'installation de CUDA peut se faire de deux manières différentes : via apt-get ou manuellement. La première a l'avantage d'être simple, la seconde, recommandée, offre plus de contrôle sur les paramètres d'installation.

Version apt-get

sudo apt-get update
sudo apt-get install nvidia-cuda-toolkit
-> Redémarrer la machine

Version manuelle

L'installation manuelle est détaillée par exemple dans ce document
http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_Getting_Started_Linux.pdf
Cuda est normalement déjà installé sur votre poste. Vérifiez que votre version de compilateur est identique à la version suivante :

```
> nvcc -V
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2015 NVIDIA Corporation
Built on Tue_Aug_11_14:27:32_CDT_2015
Cuda compilation tools, release 7.5, V7.5.17
```

Si vous disposez d'un ordinateur personnel et que la commande n'est pas reconnue, installer CUDA avec la version apt-get sera suffisant pour ce TP (Attention ça prend beaucoup de temps !).

Version WSL

Windows Subsystem for Linux (WSL) est une couche de compatibilité permettant d'exécuter des exécutables binaires Linux (au format ELF) de manière native sur Windows 10. La version 2 de WSL supporte la compilation et l'exécution de programmes CUDA (la version 1 permet simplement de compiler).