



SCALIAN



TP GPU CUDA CSM 2022

Table des matières

1	HELLO WORLD !	3
	Exercice.....	4
2	ADDITION DE DEUX VECTEURS.....	4
	Première version : CPU.....	5
	Deuxième version : GPU mono-thread	5
	Troisième version : GPU mono-bloc et multi-thread	6
	Quatrième version : GPU multi-bloc et multi-thread.....	7
3	PERFORMANCES THEORIQUES.....	3
4	SEAM CARVING – RECADRAGE INTELLIGENT	8
	Théorie	8
	Exercice.....	9
	Remarques	10
5	FRACTALE DE MANDELBROT (Facultatif).....	11
	Théorie	11
	Exercice.....	11
6	CONVOLUTION + PROJET PERSO	12
	Théorie.....	12
	Pratique	12
	Aller plus loin	13
7	ANNEXE 1 - INSTALLATION DE L'ENVIRONNEMENT	13

1 PERFORMANCES THEORIQUES

Dès votre arrivée, connectez-vous au cluster Nuvolos et demandez des ressources GPU. Cette demande peut prendre quelques minutes. Pour patienter, nous vous proposons un petit exercice théorique visant à comparer les puissances crête des CPU et GPU utilisés pour ce TP. L'unité utilisée est le FLOPS (*floating-point operations per second*). Il est défini de la façon suivante :

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

Architecture GPU

Sachant que le GPU sur le cluster est une « Tesla T4 », trouvez le nom de son architecture grâce à la page suivante : <https://en.wikipedia.org/wiki/CUDA>. Ce GPU possède 2560 cœurs cadencés à 1,59 GHz.

Rapport "FLOP / CYCLES"

Ce rapport est le nombre d'opérations flottantes élémentaires qu'un processeur (CPU ou GPU) peut réaliser en un seul cycle d'horloge. En utilisant l'architecture trouvée et la page suivante : <https://en.wikipedia.org/wiki/FLOPS>, donnez en Giga Flops la puissance de calcul théorique en simple et en double précision de votre GPU :

	Double précision (FP64)	Simple précision (FP32)
Flop / cycles		
Flops théoriques (Gflops)		

Pour comparer, faites le même exercice sur le CPU présent sur le cluster. Le modèle est un AMD EPYC ROME 7v12 avec 64 cœurs cadencés à 2,45 GHz.

	Double précision (FP64)	Simple précision (FP32)
Flop / cycles		
Flops théoriques (Gflops)		

Comparer les deux tableaux et donner vos conclusions.

Certains GPU sont fabriqués pour conserver de bonnes performances pour du calcul en double précision. C'est le cas par exemple de la Tesla A100.

Informations hardware

Arrivé ici, votre demande de ressources sur le cluster a dû aboutir. Vous allez pouvoir vérifier le hardware présent sur le cluster. Allez dans le dossier « TP_CUDA/Utilities/ », compilez (./compile) et exécutez « deviceQuery ». C'est un fichier source fourni par NVIDIA qui donne beaucoup d'information sur le ou les GPU présents. Relevez le modèle, le nombre de cœurs CUDA, la fréquence d'un cœur, la quantité de mémoire disponible et les champs « Max dimension size of a thread block » et « Maximum number of threads per block » qui vous serviront par la suite.

Les informations sur le CPU peuvent être affichées via la commande :

➤ Cat /proc/cpuinfo

Bandes passantes

Dans le même dossier exécutez « bandwidthTest » qui permet de lancer un bench sur les bandes passantes disponibles. Notez les résultats en Go/s. Pour comparaison, la bande passante entre un processeur et la RAM est d'environ 100 Go/s.

2 HELLO WORLD !

Le premier exemple proposé ici consiste à afficher la chaîne 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 le nombre de threads à utiliser et le dimensionnement de la grille de calcul utilisée par le noyau. Nous y reviendrons.

EXERCICE

- Rendez-vous dans le répertoire « TP_CUDA/HelloWorld » puis compilez le programme en utilisant le compilateur de Nvidia :

```
> nvcc helloworld_example.cu -o helloworld
```
- Exécutez le programme créé et vérifiez que la chaîne de caractères est bien écrite en console :

```
> ./helloworld
```
- Regardez le code et comprenez l'intérêt de la ligne `cudaDeviceSynchronize();`

3 ADDITION DE DEUX VECTEURS

Cet exercice propose d'ajouter 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 / n thread		

GPU – n blocs / n 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. Compilez et exécutez cette version. Un timer permet de mesurer le temps de calcul de l'addition uniquement. La commande « time » de linux vous permet de relever le temps d'exécution total.

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 !

Solution 1: gridDim.x = 1, blockDim.x = 1 (1*1 = 1 Threads)

threadIdx.x = 0

vector In_1	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
vector In_2	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=
vector Out	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27

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

Faites un « profiling » de votre application GPU avec les commandes suivantes et remplir le tableau ci-dessus :

- 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 (cf. le « Maximum number of threads per block » de tout à l'heure) 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. Attention ! La taille du tableau n'est pas forcément un multiple du nombre de threads...

Solution 2: gridDim.x = 1, blockDim.x = 7 (1*7 = 7 Threads)

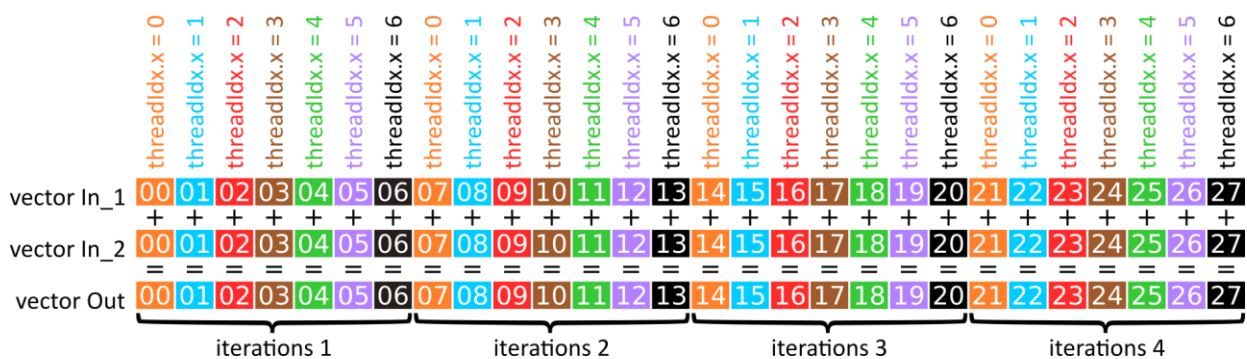


Figure 2. Chaque thread ne calcule qu'une partie de l'addition des deux vecteurs

Une fois votre programme au point, lancez un test de performance via 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)

QUATRIEME VERSION : GPU MULTI-BLOC ET MULTI-THREAD

Pour finir, l'objectif est de reprendre la version précédente en utilisant cette fois-ci plusieurs blocks (ce qu'on appelle une « grille ») possédant plusieurs threads. C'est de cette manière que l'on peut dépasser la limite de 1024 threads vue précédemment.

L'objectif maintenant est de modifier le fichier pour utiliser un nombre de threads égal à la taille du vecteur traité de façon à faire disparaître la boucle for : 1 thread = 1 addition. Pour cela, suivez les étapes suivantes :

- 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 : $(N/256 + 1)$**

• 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>>>.

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 ».

Tirez vos conclusions sur les différents temps de calcul obtenus. Est-il « rentable » d'avoir porté cet algorithme sur GPU ? Pourquoi ?

Solution 3: gridDim.x = 4, blockDim.x = 7 ($4 \times 7 = 28$ Threads)

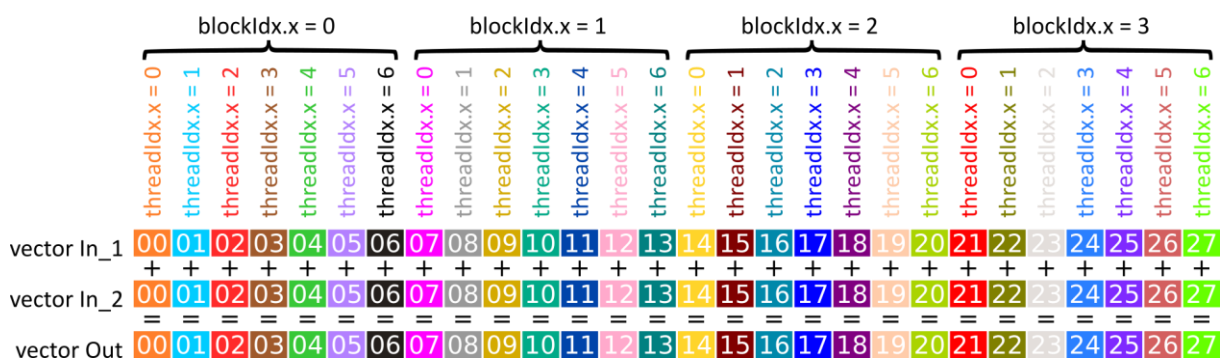


Figure 3. Chaque thread effectue simplement une addition.

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

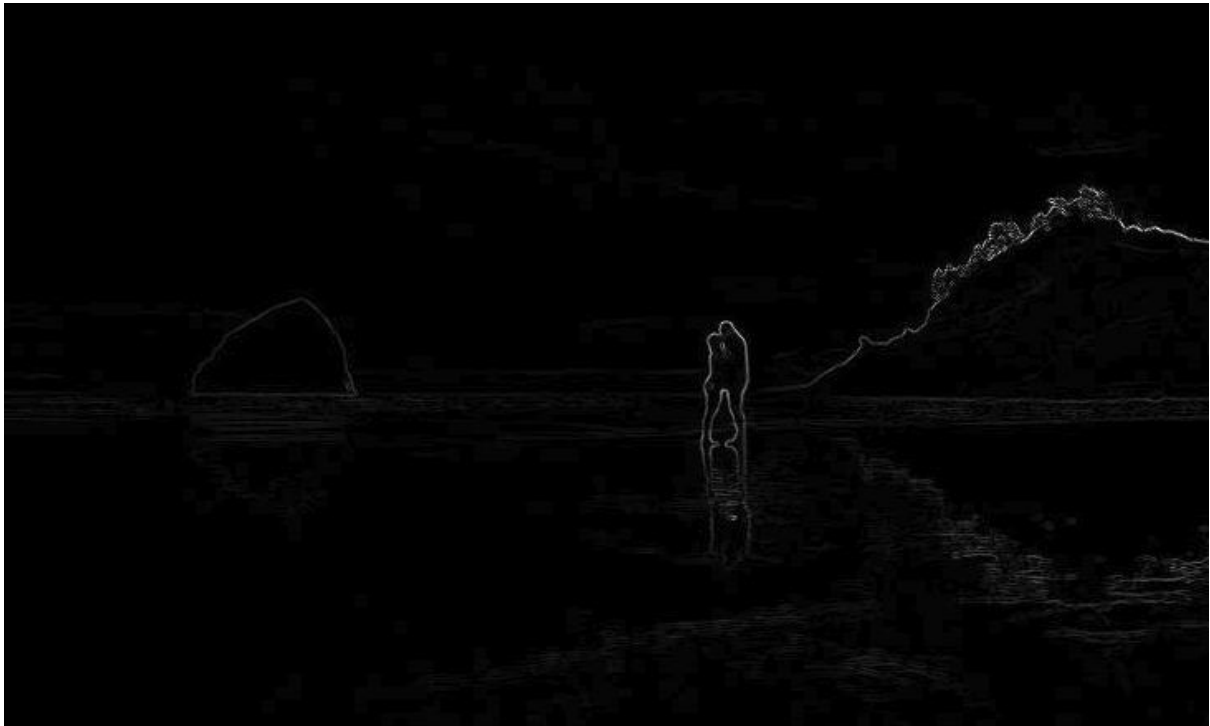
- Ensuite, un autre algorithme détecte les chemins de basse énergie et les retire de l'image originale

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 ;
 - Donner 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 ? Bravo! Essayer de trouver une piste d'optimisation de votre code en utilisant l'outil de profiling "/usr/local/cuda/bin/nvprof".

REMARQUES

Jusqu'à présent, nous avons 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 l'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 est 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 CONVOLUTION + PROJET PERSO (FACULTATIF)

Bravo d'être arrivé jusqu'ici, comme dernier exercice, nous vous proposons d'explorer des algorithmes de traitement d'image. Le premier d'entre eux sera la convolution (floutage) et les autres sont aux choix de l'étudiant.

Nous utiliserons comme entrée les images fractales générées dans l'exercice précédent.

Théorie

Le fait de flouter une image est aussi appelé "convolution d'image". La convolution la plus élémentaire remplace chaque pixel de l'image par la "moyenne" des pixels environnants.

Plusieurs configurations de pondérations des pixels voisins existent. Pour notre problème, on prendra dans un premier temps la plus basique :

	1	
1	1	1
	1	

Chaque voisin compte pour 1/5 éme de la couleur du pixel central.

Les autres hypothèses sont les suivantes :

- On ne s'occupe pas du cas particulier des bords ni celui des coins
- On travaille en niveau de gris et pas en RGB

Pratique

Se rendre dans le répertoire /Convolution/ et compiler les sources

- `> ./compile`

Exécutez la convolution sur l'image du répertoire avec 20 itérations

Relevez les performances

Le but de l'exercice est de coder une version GPU de la convolution en s'inspirant de la version CPU donnée et en partant du fichier « **convolution_gpu_todo.cu** »

- Copiez le fichier « **convolution_gpu_todo.cu** » et nommez la copie « **convolution.cu** ».
- Modifiez le fichier « **compile** » pour compiler le nouveau fichier.
- Ecrire le kernel « **convol kernel** »
- Ecrire le code manquant dans « **apply_convolution** »
- Lancer une convolution sur l'image fractale : `> ./convolution_gpu ./image.pgm 20`
- Afficher le résultat : `> display ./image_convolution GPU.pgm`

- Comparer les performances entre la version CPU et la version GPU

Tips :

- Récupérer un pointeur depuis un `std::vector` se fait avec la méthode `data()`.

Aller plus loin

Vous pouvez maintenant implémenter l'algorithme de traitement d'image de votre choix. Quelques exemples pour s'inspirer :

- Retourner l'image suivant un des axes
- Transposer l'image
- Augmenter/diminuer la luminosité
- Filtrer une plage de valeur
- Augmenter le contraste

7 ANNEXE 1 - INSTALLATION DE L'ENVIRONNEMENT

Natif Linux

Cette annexe décrit comment installer CUDA sous environnement Linux (debian, ubuntu etc.).

Pour compiler un code CUDA, vous avez seulement besoin du compilateur NVCC de NVIDIA. Pour le récupérer, c'est très simple :

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

Mais attention ! Cette installation permet uniquement de compiler un code mais pas de l'exécuter. Exécuter un code CUDA requiert également l'installation du driver graphique adapté. Pour ce faire, il faut suivre le tuto fourni par NVIDIA à cette adresse :

http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_Getting_Started_Linux.pdf

Attention, il faut avoir une bonne ½ journée devant soi avant de se lancer (si tout se passe bien)...

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).