CALCUL HAUTE PERFORMANCE

PROGRAMMATION PARALLELE SUR GPU

Image processing with OpenCL

Par El Amrani-Zerrifi Hamza & Schoonaert Bastien

Préambule

Cette archive contient :

- Un dossier *chp_gpu,* contenant tous les programmes et fichiers nécessaires à l'exécution des algorithmes.
- Un fichier *README.txt,* contenant les instructions pour exécuter les différents programmes.
- Ce compte-rendu

1ère partie : Etude du filtre moyenneur

Nous avons commencé par la création d'un **filtre moyenneur.** Le principe d'un tel filtre est d'effectuer la moyenne des valeurs des pixels d'une image autour d'un pixel donné, et d'attribuer cette moyenne au pixel de la nouvelle image correspondante.

En discrétisant en 2D, les pixels de la nouvelle image g sont obtenus à l'aide de la formule suivante :

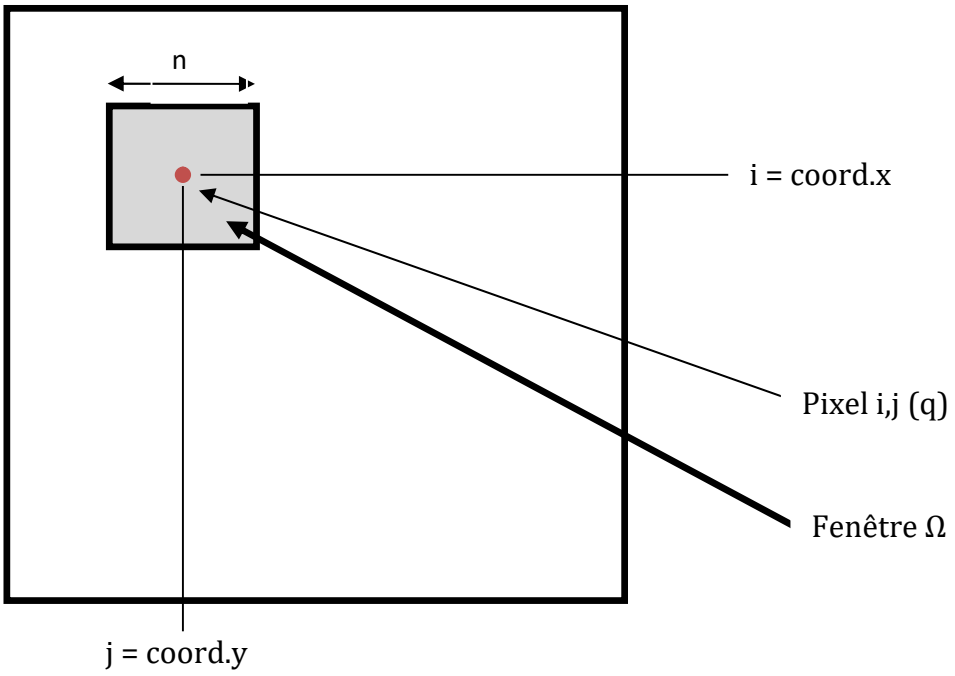
$$p \in \mathcal{D}, g(p) = \frac{1}{\#\Omega} \sum_{q \in \Omega} f(q)$$

où **Ω correspond à la fenêtre carrée** dans laquelle la moyenne est effectué par chaque thread en parallèle. Dans notre programme, nous désignerons par ***N* la largeur de cette fenêtre carrée (paramètre d'entrée),** tel que le cardinal #Ω vaille N^2.
f(q) correspond à la valeur d'un pixel q de la fenêtre Ω dans **l'image f.**

Se pose le problème des pixels aux bords, autour desquels une fenêtre de taille N ne peut rentrer entièrement. Ceux-ci recevrons arbitrairement la valeur **0** pour chacune des composantes. Nous rappelons que chaque pixel est constitué de 4 composantes : RGBA (red, green, blue, alpha (transparence)). L'algorithme effectue ainsi la moyenne sur **chacune des composantes** afin d'obtenir le pixel filtré.

Figure n°1 : Schéma explicatif du filtre moyenneur (avec n la demi-largeur (n = N/2))

$$g_{i,j} = 1/(2*n + 1)^2 * (\sum_{l=-n}^{n} \sum_{m=-n}^{n} f_{l,m})$$



Voici les résultats obtenus pour différentes valeurs de N passées en entrée :

Figure n°2 : N = 0

Figure n°3 : N = 4

Figure n°4 : N = 10

Figure n°5 : N = 100

On remarque que pour N=0, on retrouve bien l'image originale, car **aucun filtre n'est appliqué.** Plus on augmente la taille de la fenêtre N, plus on observe logiquement l'effet de flou induit par le filtre moyenneur. Pour N = 100, les moyennes sont effectuées sur un nombre de pixels assez grand pour que le floutage ne permette plus la distinction de l'image d'origine. On notera que l'image filtrée pour N = 100 apparaît comme **plus petite** que les autres, à cause des conditions de bords expliquées précédemment.

2ème partie : Étude du filtre gaussien

On étudie dès à présent le **filtre gaussien.** Similaire au filtre moyenneur, la formule donnant les coordonnées i,j de l'image de sortie g par rapport à l'image d'entrée f est la suivante :

$$g(i,j) = \frac{1}{\underbrace{\sum_{l=-N}^{N}\sum_{m=-N}^{N}\frac{1}{\sqrt{2\pi\sigma^2}}e^{-\frac{l^2+m^2}{2\sigma^2}}}_{normalization\,factor}} \sum_{l=-N}^{N}\sum_{m=-N}^{N}\underbrace{\frac{1}{\sqrt{2\pi\sigma^2}}e^{-\frac{l^2+m^2}{2\sigma^2}}}_{\text{Gaussian function}} \cdot f(i+l, j+m)$$

On a désormais **deux paramètres**, la **taille de la fenêtre N** (entier) et le **réel Sigma**. Pour ce qui est de N, le paramètre est le même que pour le filtre moyenneur et son influence est donc identique sur le filtre gaussien. Étudions désormais l'influence du paramètre Sigma sur notre nouveau filtre.

Prenons comme repères l'image initiale ainsi qu'une image donnée par le filtre moyenneur :

Figure n°6 : Image de départ

Figure n° 7 : Sortie du filtre moyenneur (N=12)

Voici désormais deux images de sorties du filtre gaussien, en faisant varier le paramètre Sigma :

Figure n° 8: Image de sortie du filtre gaussien avec N=12 et Sigma=1e3.

Figure n° 9: Image de sortie du filtre gaussien avec N=12 Sigma=1e-3.

On remarque que lorsque **Sigma tend vers l'infini**, notre filtre gaussien se rapproche du **filtre moyenneur**. Tandis que lorsque **Sigma tend vers 0**, notre filtre gaussien tend vers **une copie d'image**.

En effet, on vérifie facilement cette hypothèse en nous intéressant à la formule mathématique de la transformation du filtre gaussien :

$$g(i,j) = \frac{1}{\underbrace{\sum_{l=-N}^{N}\sum_{m=-N}^{N}\frac{1}{\sqrt{2\pi\sigma^2}}e^{-\frac{l^2+m^2}{2\sigma^2}}}_{normalization\,factor}} \sum_{l=-N}^{N}\sum_{m=-N}^{N} \underbrace{\frac{1}{\sqrt{2\pi\sigma^2}}e^{-\frac{l^2+m^2}{2\sigma^2}}}_{\text{Gaussian function}} \cdot f(i+l,j+m)$$

Lorsque Sigma tend vers l'infini, **l'exponentielle tend vers 1** et on retrouve **la formule du filtre moyenneur**.

Lorsque Sigma tend vers 0, **l'exponentielle tend vers 0**, sauf lorsque l=m=0 où l'exponentielle vaut 1. Dès lors, seul le terme l=m=0 domine dans la somme et l'on retrouve que **g(i,j)=f(i,j),** c'est-à-dire une copie d'image.

3ème partie : Optimisation du temps de calcul

Nous traitons désormais les optimisations faites pour diminuer le temps de calcul de l'image de sortie du filtre gaussien. Étudions la formule mathématique de la transformation :

$$g(i,j) = \frac{1}{\underbrace{\sum_{l=-N}^{N}\sum_{m=-N}^{N}\frac{1}{\sqrt{2\pi\sigma^2}}e^{-\frac{l^2+m^2}{2\sigma^2}}}_{normalization\,factor}} \sum_{l=-N}^{N}\sum_{m=-N}^{N} \underbrace{\frac{1}{\sqrt{2\pi\sigma^2}}e^{-\frac{l^2+m^2}{2\sigma^2}}}_{\text{Gaussian function}} \cdot f(i+l,j+m)$$

Le facteur $\frac{1}{\sqrt{2\pi\sigma^2}}$ **se simplifie.** De plus, on remarque que dans la boucle, on peut **factoriser le terme exponentiel en sigma** pour le calculer une unique fois. Également, on remarque que le calcul du facteur de normalisation peut se faire en **même temps** que le terme de la fonction gaussienne, c'est-à-dire dans les même boucles.

Calculons le **temps d'exécution de l'algorithme naïf** avec uniquement ces optimisations mineures :

N	Sigma	Execution time
12	10	0.04573 seconds

Maintenant que l'on possède un temps de référence, poussons les optimisations plus loin. On remarque qu'à l'intérieur de la boucle, il suffit à chaque passage d'une multiplication entre deux termes de la forme e^{-l^2} et e^{-m^2} pour l et m parcourant l'intervalle entier [0,N], afin de calculer le terme exponentiel.

L'idée est donc la suivante : on calculera **à l'extérieur de la boucle** un tableau *exponentials* de taille N+1, contenant les termes *exponentials*[k]=e^{-k^2}

En utilisant ce tableau, nous n'aurons plus besoin de calculer les exponentielles à chaque tour de boucle, mais nous aurons seulement à **effectuer les multiplications** expliquées précédemment à partir des éléments du tableau déjà calculés.

De plus, il serait intelligent de ne pas avoir à calculer ce tableau pour chacun des threads qui effectue le calcul en parallèle, mais de le placer **en mémoire partagée**. Pour cela, nous calculons ce tableau *exponentials* au sein du fichier **imageCopyFilter.cpp** et l'introduisons comme **paramètre de notre kernel** gauss.

Nous sommes en revanche obligés de le placer en **mémoire globale**, pour qu'il soit accessible par tous les work-groups, ce qui malheureusement engendre une certaine latence. Néanmoins, nous déclarons notre tableau comme un __constant float*, c'est-à-dire qu'il est placé **en mémoire constante**. L'accès à cette mémoire est à priori un peu plus rapide que pour le reste de la mémoire globale, néanmoins le tableau sera en **read-only**. Comme nous n'avons pas besoin de modifier le tableau par chacun des threads, c'est bien cette mémoire constante qui est la plus appropriée pour stocker ce tableau.

Effectuons le test avec ces optimisations :

N	Sigma	Execution time
12	10	0.03193 seconds

L'algorithme optimisé avec le tableau exponentiel calculé en mémoire constante offre un speed-up de **1.5.**