Programmation parallèle avec OpenMP

Thibaud Lambert, Brett Ridel

Ce TP a pour but d'introduire et de pratiquer les notions de base de la programmation parallèle. L'objectif est de paralléliser des fonctions à l'aide d'OpenMP, permettant de diminuer le temps d'exécution. Nous allons nous intéresser à deux cas d'application qui sont l'amélioration de la dynamique d'une image en faisant un étirement d'histogramme, et l'application d'un effet de type pointillisme à une image.

Rappel

Le code fourni est basé sur celui réalisé en projet C++, mais simplifié. Voici un rappel des méthodes qui pourraient vous être utiles :

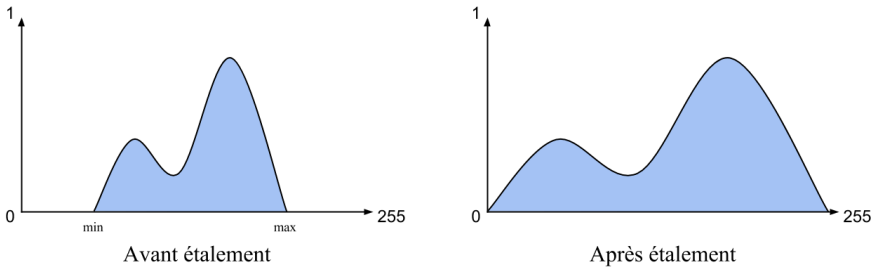
- **Image::getHeight()** renvoie la hauteur de l'image,
- **Image::getWidth()** renvoie la largeur de l'image,
- **Image::getPixel(int x, int y, int c)** renvoie la valeur du pixel aux coordonnées (x, y), et à la profondeur c,
- **Image::setPixel(int x, int y, int c, float v)** modifie la valeur du pixel aux coordonnées (x, y), et à la profondeur c, en utilisant la valeur v.

Voici quelques directives OpenMP qui vous seront utiles :

- **omp_get_max_threads()** renvoie le nombre maximum de threads,
- **omp_get_thread_num()** renvoie le numéro du thread actuel.

1 Amélioration de la dynamique

L'étirement d'histogramme a pour but d'augmenter le contraste d'une image. L'idée est de faire en sorte que le minimum et le maximum de l'image soient bien 0 et 1.



1.1 Appliquer la transformation

En considérant que l'on connaisse le maximum et le minimum de l'image, il est possible d'améliorer le contraste de l'image en appliquant la formule suivante:

$$v' = (v - v_{min})/(v_{max} - v_{min}) \tag{1}$$

Commencez par implémenter la fonction **equalizeSequential** du fichier **histogramequalization.cpp**. Pour cela, parcourez l'ensemble des pixels de l'image et appliquez la transformation nécessaire (équation 1).

Comme l'opération à appliquer se fait de manière indépendante sur chaque pixel, il est alors simple de paralléliser ce code à l'aide OpenMP. En vous inspirant de la fonction **equalizeSequential**, implémentez la fonction **equalizeParallelized**. Pour cela, utilisez **#pragma omp parallel for**. Exécutez le programme et comparez les temps d'exécution entre les deux versions.

1.2 Calcul du minimum et maximum

Jusqu'à présent, les valeurs du minimum et du maximum sont fixées directement dans le code. Afin de pouvoir gérer n'importe quelle image, il faut calculer ces valeurs. Pour activer le calcul dynamique de ces valeurs, **commencez par passez la variable useManualBoundary à false** dans le fichier **main.cpp**.

En s'inspirant de la version séquentielle proposée dans la fonction **minMaxSearchSequential**, **implémentez la fonction minMaxSearchReduction du fichier histogramequalization.cpp** permettant de calculer les valeurs minimum et maximum d'une image. Vous devez pour cela utiliser la directive **#pragma omp parallel**. Les paramètres **min** et **max** sont communs à tous les threads, vous devez donc faire attention aux accès concurrents.

2 Pointillisme

Le pointillisme est une opération qui se base sur une technique d'échantillonnage préférentiel, permettant de simuler des coups de crayon (figure 1). L'idée est de générer aléatoirement des coordonnées de pixels (ou échantillons) avec lesquelles on va dessiner un point sur l'image. Si l'on veut que l'image en sortie ne soit pas un simple bruit, mais ressemble tout de même à l'image référence, les échantillons sont générés de manière *préférentielle*, permettant par exemple de préférer les zones claires aux zones sombres.

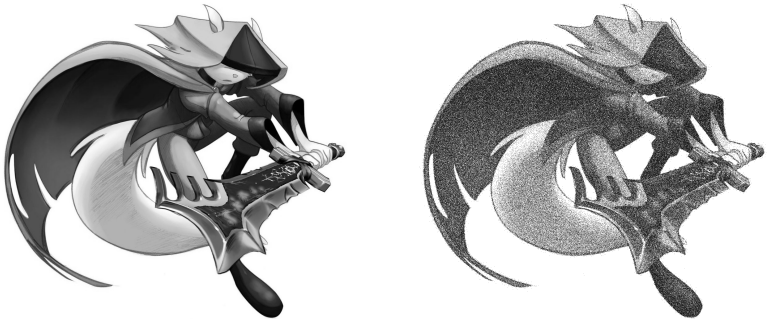


Figure 1 – Illustration de l'effet pointillisme. Image originale (gauche), transformée (droite).

En considérant la valeur d'un pixel comme une probabilité non normalisée, la première étape de cette méthode consiste à calculer la cdf (fonction de répartition) de notre image, permettant par la suite de guider notre échantillonnage. L'implémentation proposée calcule une cdf pour chaque ligne (somme de la valeur de tous les pixels précédents) puis une cdf pour l'ensemble des colonnes (chaque ligne est associée à une valeur, et l'on somme les valeurs de toutes les lignes précédentes). L'étape suivante consiste à générer les échantillons. Pour cela, on génère des nombres aléatoires compris entre 0 et 1. En utilisant l'inverse de la cdf et ces nombres aléatoires, on obtient alors le numéro de la ligne et de la colonne de l'échantillon (figure 2). Finalement, on utilise ces échantillons (des coordonnées 2D) pour ajouter un point dans l'image.

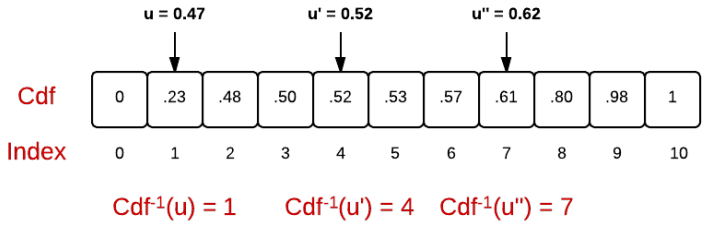


Figure 2 – Illustration d'une cdf.

L'échantillonnage est la partie la plus coûteuse de la méthode et donc la plus intéressante à paralléliser. Nous allons tester plusieurs approches pour la parallélisation de la génération des échantillons.

2.1 Avec section critique

L'approche la plus intuitive consiste à paralléliser la boucle générant les échantillons. Cependant deux échantillons ne doivent pas écrire en même temps dans le même pixel. En s'inspirant de la méthode **sequentialSampling** de la classe **Pointillisme**, implémentez la méthode **parallelizedSampling**. Utilisez la directive **#pragma omp critical** pour éviter les accès concurrents. Adaptez le code pour que chaque thread possède sa propre instance de **Random**. Dans le cas contraire, les threads pourraient utiliser les même nombres aléatoires, créant un biais sur le résultat.

2.2 Sans section critique

La présence de sections critiques ralentit fortement le programme, peu importe le nombre de threads. L'idée est donc de modifier notre approche (l'algorithme) pour éviter d'utiliser des sections critiques. Pour cela, on aimerait attribuer le traitement d'une ligne à un seul et même thread. Un thread pourra donc s'occuper de plusieurs lignes, mais une ligne ne pourra être traitée que par un seul thread.

2.2.1 Version séquentielle

L'idée est de séparer le calcul de la position en x et y de chaque échantillon :

1. pour chaque échantillon, on tire aléatoirement un nombre entre 0 et 1 puis on se sert de la cdf pour déterminer le numéro de ligne (la coordonnée y). Ce numéro est sauvegardé. Ainsi, à la fin de cette étape, on sait combien d'échantillons doivent être lancés pour chacune des lignes.

2. pour chacune des lignes, on génère le nombre d'échantillon calculé précédemment, et on modifie l'image en fonction des coordonnées (x, y) calculées.

Implémentez cela dans la méthode **twoStepSampling** de la classe **Pointillisme**.

2.2.2 Version parallélisée

La première étape peut se paralléliser sous la forme d'une réduction : chaque thread stocke dans son propre tableau le nombre d'échantillons obtenus. Il faut ensuite fusionner ces tableaux.

On peut paralléliser la seconde étape assez facilement. Dans la deuxième étape, il faut que chaque ligne soit traitée par un thread différent. Comme chaque thread s'occupe d'une ligne différente, la zone critique n'est plus nécessaire.
Implémentez cela dans la méthode **twoStepParallelizedSampling** de la classe **Pointillisme**.

Il est possible d'utiliser la directive suivante pour répartir dynamiquement la charge entre les threads : **#pragma omp parallel for schedule(dynamic)**. Comparer l'ordonnancement dynamique et static sur différentes images.