Margot Saget Thomas Reffay Vialatte Sylvain

Compte Rendu

1. L'algorithme implémenté

Les convolutions que nous avons implémentés sont un flou normalisé avec une matrice 3*3, 5*5 et 11*11, un flou de gauss avec une matrice 3*3 et une 5*5, une matrice de convolution de netteté et une matrice de convolution de détection des contours.

Notre algorithme les exécute toutes à la suite sur l'image ciblée.

Pour le traitement des bords, qui peuvent poser des problèmes nous avons choisi principalement pour une question de temps une sorte de méthode crop. Les pixels problématiques (ceux qui n'ont pas suffisamment de voisins pour qu'on puisse faire le même calcul que pour tous les autres pixels) sont ignorés et sont passés en noir (0,0,0).

Notre algorithme peut gérer les images dotées d'une couche alpha tout comme celles qui n'en ont pas.

Pour l'utiliser 4 méthodes :

- sans paramètre de programme : le programme va tenter de lire un fichier nommé "in.jpeg" et appliquer les traitements dessus, les fichiers de sortie sont au format "out_[nom de la transformation].jpeg".

- avec un seul paramètre de programme : le programme va tenter de lire le fichier avec pour nom exactement le paramètre et appliquer les traitements dessus, les fichiers de sortie sont au format "out_[nom de la transformation][nom du fichier]".

- avec deux paramètres de programme : le programme va tenter de lire le fichier avec pour nom exactement le premier paramètre et appliquer les traitements dessus, les fichiers de sortie sont au format "out_[nom de la transformation][deuxième paramètre]", il n'ajoute pas l'extension, à l'utilisateur de la définir.

- avec 3 ou plus paramètres : il agit comme s'il n'en avait reçu aucun, ce n'est pas une méthode prévue pour être utilisée.

Margot Saget Thomas Reffay Vialatte Sylvain

Le déroulé de l'algorithme :

1) il ouvre l'image et en récupère les informations importantes

2) il lance tous les processus de traitement différent de façon séquentiel
    a) il crée le vector pour stocker les valeurs de sortie
    b) il crée la matrice de convolution associée au traitement
    c) il lance le traitement de l'image
    d) première double boucle de parcours en CPU, récupération de l'id du thread en GPU
    e) pour chaque pixel courant s'il est dans la limite des pixels à traiter, on applique la convolution
    f) on ne fait pas une normalisation linéaire pour netteté (sharpness) et pour détection des contours(detect edge) car cela change la luminosité de l'image
    g) le traitement terminé le fichier de sortie est créée et le programme passe au suivant

2. Les difficultés rencontrées

Réaliser les différentes convolutions même seulement en CPU s'est avéré plus difficile que prévu.
Si le concept de base est assez simple, son application concrète est beaucoup plus difficile et nous avons trouvé certaines informations très difficiles à obtenir sur internet pour des néophytes du traitement d'image.
En particulier en ce qui concerne la normalisation (ou son absence) lors du calcul. Ou encore que faire des pixels quand la somme pondérée est négative ?
Cela nous a amené à faire énormément de tests et de recherches pour enfin trouver des choses qui fonctionnent à peu près. Actuellement notre méthode qui effectue le traitement sur l'image n'est pas capable de s'adapter à n'importe quelle matrice de convolution (nous ignorons si une telle méthode existe) et nous avons donc abandonné l'idée de la matrice qui doit effectuer un masque de flou puisque nous n'avons jamais réussi à obtenir le résultat visé.

Pour le passage en GPU, le transfert de certains types de données à la carte graphique ne s'est pas fait simplement.
Nous utilisons une structure pour représenter une matrice de convolution et toutes les infos qui l'accompagnent dont on se sert. Nous avons eu des difficultés à transférer la structure à la carte graphique.
Nous avons aussi eu des difficultés avec le type std::vector de c++ que Cuda n'a pas l'air d'accepter. Nous avons donc dû les convertir en unsigned char*.

De plus de nombreux bugs sont parvenus et n'étaient pas simples à trouver, au départ, nous avons tenté de faire une simple convolution identité, car la fonction pour calculer les convolutions ne fonctionnait plus. Il s'est alors passé une chose intrigante. La première est que l'image de sortie censée être la même que celle d'entrée était totalement différentes, le premier tiers en haut de l'image était séparé en 2 parties, la moitié à gauche était identique à l'image d'entrée et la seconde était noire. Le reste de l'image était aussi noire, mais avec des pixels de couleurs sans cohérence dans leur placement.
Après des essais afin de savoir si la gestion des bords ne posait pas soucis ou bien si les threads agissent sur les bons pixels. Le problème était une inversion d'axes qui faisait que la largeur et la hauteur étaient inversées et qui créait ce phénomène.
Le second souci a pris plus de temps à être corrigé, nous avons tout d'abord émis l'hypothèse d'une mauvaise gestion des channels de couleurs à cause d'un camarade ayant eu le même problème, ou bien un problème de pointeur qui ne prenait pas la bonne valeur. À la suite de nombreux tests minutieux ou l'on retraçait le programme et les valeurs que les variables prenaient.
On s'est aperçu que le pointeur contenant les données de l'image de sortie était trop court, nous n'avions pas fait attention lors de l'allocation de la mémoire et lors de la

copie de données et avons oublié de multiplier par 3 le nombre de pixels de l'image expliquant que l'image ne s'affiche que sur le premier tier.
Ayant résolu le problème d'affichage de l'image, nous nous sommes attaqués à la fonction de convolution. Nous avons dû recommencer nos tests et afficher toutes les données afin de les analyser et trouver des solutions au problème. Il s'est avéré que la liste de la structure MatriceConvolution ne pouvait pas être lu dans les kernels. Nous avons dû créer un nouveau pointeur contenant la matrice de convolution, mais même une fois cela réglé, il restait le problème de la fonction calculant la convolution.

Un autre problème est apparu, le pointeur copié sur la carte graphique ne contenait qu'une partie des valeurs. Ce qui assombrissait l'image de sortie.
On utilise un tableau d'int qu'on transmet à la carte graphique avec un CudaMemCpy() mais lorsqu'on essayait d'afficher ce tableau d'entiers sur la carte graphique, il manquait les ⅔ des valeurs comme si elle n'avait pas été copiée. Elles sont donc initialisées à 0.

Ainsi par exemple, avec une liste comme ceci sur le CPU : {1,1,1,1,1,1,1,1,1}
Nous nous retrouvions sur le kernel avec une liste comme ceci : {1,0,0,1,0,0,1,0,0}.

La solution que nous avons trouvée était de prendre en compte la taille d'un entier dans la réservation. Nous ne l'avions pas fait car par exemple pour transmettre le vecteur des données de l'image nous n'en avons pas besoin, et nous ne l'avions jamais fait dans le cours non plus.

3. Les choix d'optimisation

Nous avons finalement réussi à avoir un kernel qui reproduit l'algorithme que nous avons écrit en c++.Mais le jour-même du rendu, ce qui nous a laissé peu de temps pour effectuer des tests d'optimisation.

Dans une configuration (pensée pour le débogage pas pour la performance) où nous avons autant de blocs qu'il y a de ligne dans l'image et autant de threads qu'il y a de colonne:

Pour une image en 168*300 pixels:
blur3 : 0.186784 ms
blur5 : 0.177152 ms
blur11 : 0.443072 ms
gaussianBlur3 : 0.070976 ms
netteté 3 : 0.071648 ms
detectEdges3 : 0.070816 ms

Nous ne nous expliquons pas vraiment pourquoi les traitements sont plus rapides sur une image plus grande mais ce doit être lié à l'architecture de la carte graphique.
Pour une image en 14400*7800 pixels:
blur3 : 0.003616 ms
blur5 : 0.003232 ms
blur11 : 0.003456 ms
gaussianBlur3 : 0.003488 ms
netteté 3 : 0.003424 ms
detectEdges3 : 0.003104 ms

Dans une configuration (que nous pensions bien meilleure) avec une taille de blocs de (32,4) threads et une taille de grille de ((colonnes de l'image-1) / tailleDeBloc.x+1 ,
(lignes de l'image-1) / tailleDeBloc.y+1) :

Pour une image en 168*300 pixels:
blur3 : 11.9788 ms
blur5 : 0.782336 ms
blur11 : 12.758 ms
gaussianBlur3 : 0.294656 ms
netteté 3 : 0.291008 ms
detectEdges3 : 0.070816 ms

Margot Saget Thomas Reffay Vialatte Sylvain

Pour une image en 14400*7800 pixels:
blur3 : 1263.99 ms
blur5 : 5523.96 ms
blur11 : 27888.8 ms
gaussianBlur3 : 3062.29 ms
netteté 3 : 3056.07 ms
detectEdges3 : 3046.59 ms

Nous ne nous attendions pas à ce résultat. Ici le nombre de thread/bloc est bien un multiple de 32 et il est suffisamment élevé pour que des recouvrements s'opèrent et pourtant il est bien bien plus long. Nous supposons que ce résultat est dû entre autres au très grand nombre de thread qui ne font pas partie de l'image. Plus les blocs sont gros, plus il y a de chances pour que des threads n'aient pas les id correspondant à des éléments de l'image.

Nous avons testé ensuite avec des blocs en dimension 2 (128,4)

Pour une image en 168*300 pixels:
blur3 : 0.436128 ms
blur5 : 1.01341 ms
blur11 : 4.82182 ms
gaussianBlur3 : 0.364832 ms
netteté 3 : 0.402496 ms
detectEdges3 : 0.374528 ms

Pour une image en 14400*7800 pixels:
blur3 : 1263.99 ms
blur5 : 5523.96 ms
blur11 : 27888.8 ms
gaussianBlur3 : 3062.29 ms
netteté 3 : 3056.07 ms
detectEdges3 : 3046.59 ms

Nous continuons les tests (en particulier à cause des résultats étranges) et là SURPRISE ! Notre algo fonctionnait bien pour l'image de chat ou encore de chien ou de grenouille mais sur une image plus grande (Grande.jpg) il ne fait qu'une partie de l'image. En regardant de plus près, nous avions effectivement fait une erreur dans les calculs d'indices. En la changeant pour la bonne valeur, ça ne fonctionne toujours pas. En ajoutant des affichages d'erreur nous avons des erreurs que nous n'avons pas réussi à réparer avant le jour du rendu.

Margot Saget Thomas Reffay Vialatte Sylvain

Nous allons tout de même ajouter des temps CPU obtenus :

Pour l'image de Galaxie(14400*7800) :
Temps blur3 : 6788 ms
Temps blur5 : 12890 ms
Temps blur11 : 41124 ms
Temps gaussianblur3 : 12932 ms
Temps gaussianblur5 : 6846 ms
Temps detectEdges3 : 6467 ms
Temps sharpness3 : 6380 ms

Pour l'image de chat(300*168) :
Temps blur3 : 3 ms
Temps blur5 : 6 ms
Temps blur11 : 18 ms
Temps gaussianblur3 : 6 ms
Temps gaussianblur5 : 3 ms
Temps detectEdges3 : 3 ms
Temps sharpness3 : 3 ms

On peut voir que notre code qui fonctionnait à moitié nous faisait tout de même gagné énormément de temps sur les images où il fonctionnait (l'image 300*168). On peut voir en effet que c'était jusqu'à 10* plus rapide.