CAQUELARD Vincent

GBOHO Thierry

HEBRAS Jérôme

Université d'Orléans 2020-2021

Rapport de Projet
Programmation graphique

Sommaire

1. Utilisation du programme

Cette section décrit comment compiler puis exécuter le programme. A noter que ces informations sont déjà succinctement présentes dans le README.md. Toutes les commandes suivantes peuvent être copiées-collées dans l'ordre.

A. Compilation

Cloner le projet, puis se déplacer dans le dossier build obtenu :

```
$ git clone https://github.com/JeyJer/CUDAProject.git
$ cd ./CUDAProject/build
```

Réaliser un cmake sur le répertoire parent puis compiler avec make :

```
$ cmake ..
$ make
```

Vous obtenez ainsi 3 executables :

1. ***img_processor*** *: obsolète, vestige d'une version antérieure,*
2. **gpu_img_processor** : traitement d'image depuis le GPU,
3. **cpu_img_processor** : traitement d'image depuis le CPU.

B. Exécution

Commençons par **gpu_img_processor**. Tout d'abord, il peut être exécuté tel quel – sans argument – en s'auto-configurant d'après les valeurs suivantes :

Paramètre	Valeur
in_path	"./in.jpg"
out_path	"./out.jpg"
dimX	32
dimY	32
shared	0
stream	0
filter	"boxblur"
pass	1

Voici une description de ces paramètres :

- **in_path** : chemin vers l'image d'entrée
- **in_path** : chemin vers l'image de sortie.
- **dimX** : configuration de la dimension X pour le device,
- **dimY** : configuration de la dimension Y pour le device,
- **shared** : définit l'utilisation de la mémoire shared. (1 = true, 0 = false)
- **stream** : définit le nombre de streams à utiliser. 0 signifie pas de stream. Une autre valeur que 0 signifiera le nombre de streams qui seront utilisés.
- **filter** : le filtre utilisé pour la modification de l'image. Les filtres suivants sont supportés :
  - ***boxblur***,
  - ***gaussianblur*** : flou de Gauss avec une matrice 3*3,
  - ***gaussianblur5*** : flou de Gauss avec une matrice 5*5,
  - ***emboss*** : repoussage avec une matrice 3*3,
  - ***emboss5*** : repoussage avec une matrice 5*5,
  - ***sharpen*** : amélioration du contraste.
- **pass** : le nombre de fois qu'un filtre peut être appliqué.

Tous les paramètres ci-dessus peuvent être modifiés lors de la commande d'appel :

$gpu_img_processor [in_path] [out_path] [dimX] [dimY] [useSharedMemory] [useNumberStream] [filter pass]

A noter que le paramètre [filter pass] peut être répété autant de fois que désiré. Par exemple, pour appliquer un flou de Gauss (matrice 5*5) 10 fois, puis une amélioration du contraste 5 fois et enfin un repoussage (matrice 3*3) 2 fois, il est possible d'écrire :

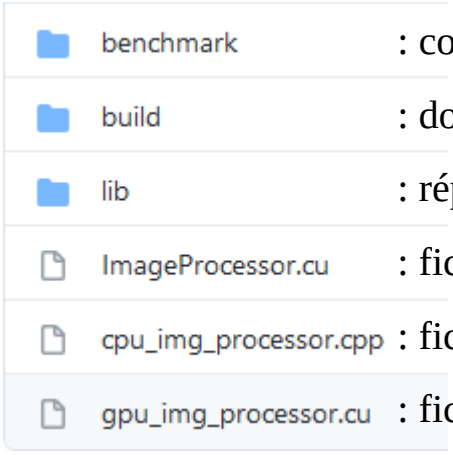
$gpu_img_processor … gaussianblur5 10 sharpen 5 emboss 2

Enfin, pour ce qui est de **cpu_img_processor**, il prend moins de paramètres en entrée que son équivalant gpu. Toutefois, les valeurs par défaut de ces paramètres sont les mêmes.

$gpu_img_processor [in_path] [out_path] [filter pass]

2. Explication du code

Cette section détaillera les fichiers principaux du projet tels que nous les voyons depuis le dépôt Git. Depuis la racine du projet, nous visualisons 3 répertoires et 3 fichiers sources.



: contient divers fichiers de mesures pour les benchmarks.

: dossier de build du projet.

: répertoire contenant l'essentiel des fichiers sources.

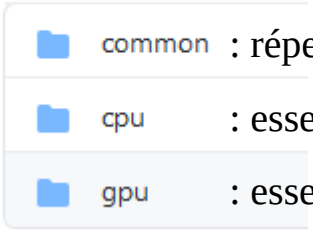
: fichier main de ImageProcessor. Obsolète.

: fichier main du programme pour le côté CPU.

: fichier main du programme pour le côté GPU.

Lorsque nous entrons dans lib, nous observons à nouveau 3 dossier :

CUDAProject / lib /



common : répertoire contenant des fichiers communs de CPU et GPU.

cpu : essentiel des fichiers sources pour le traitement d'image CPU.

gpu : essentiel du code pour le traitement d'image GPU.

Enfin, voyons les fichiers sources contenus par ces 3 dossiers :

CUDAProject / lib / common /

Contient 2 fichiers :

- **utilies.hpp** : contient les valeurs par défaut des paramètres communs CPU/GPU, une initialisation des matrices de convolutions ainsi que des structs :
  - **ConvolutionMatrixProperties** : définit les propriétés d'une matrice, sa taille, son diviseur et un index de départ nécessaire pour le traitement d'image.
  - **ConvMatrixPointers** : contient une matrice de convolution dans son ensemble, c'est-à-dire un pointeur vers le tableau en 2 dimensions représentant la matrice et un autre pointeur vers la struct ConvolutionMatrixProperties.
  - **RgbPointers** : contient 2 pointeurs vers des tableaux d'*unsigned char*, ces tableaux contiennent les valeurs des pixels de l'image d'entrée et de sortie.
  - **Pointers** : contient 2 structs RgbPointers et ConvMatrixPointers.
- **utilies.cpp** : permet d'initialiser et gérer les matrices de convolution, mais aussi d'effectuer des opérations sur les pointeurs *unsigned char* contenant la liste des pixels de nos images. Il s'agit de certaines opérations communes aux GPU et CPU. Notamment, lorsque nous effectuons plusieurs passages d'un filtre sur une image, nous inversons les tableaux de sortie et d'entrée afin de rappeler la méthode de traitement sans avoir à faire d'autre changement.

CUDAProject / lib / cpu /

Contient 4 fichiers :

- **cpu_img_transform.hpp** : Définit le chrono et la base de la classe CpuImgTransform.
- **cpu_img_transform.cpp** : Implémente la classe CpuImgTransform. Il s'agit de la classe principale du programme **cpu_img_processor**. C'est elle qui réalise la transformation d'image en appliquant les filtres successifs.
- **cpu_utilities.hpp** : Définit la classe CpuUtilMenuSelection qui permet de récupérer et d'afficher les paramètres d'entrée du programme. Contient également 2 struct :
  - **CpuUtilPointers** : Contient 2 pointeurs *unsigned char* vers les tableaux de pixels des images in/out et des informations sur la matrice à appliquer.
  - **CpuUtilExecutionInfo** : Donne des informations sur le traitement à effectuer, avec la matrice de convolution et le nombre de passe à effectuer.
- **cpu_utilities.cpp** : Implémentation de la classe CpuUtilMenuSelection.

CUDAProject / lib / gpu /

Contient 6 fichiers :

- **gpu_img_transform.cuh** : Déclaration de la classe **GpuImgTransform**.
- **gpu_img_transform.cu** : Implémente la classe **GpuImgTransform**. Initialise et libère les ressources en VRAM et transforme une image d'après la demande de l'utilisateur. Chronomètre l'opération de transformation. Peut utiliser la mémoire shared.
- **gpu_img_transform_stream.cuh** : Déclaration de la classe **GpuImgTransformStream**. Ajoute également la déclaration d'une struct permettant l'utilisation des streams.
- **gpu_img_transform_stream.cuh** : Implémente la classe **GpuImgTransformStream**. Initialise et libère les ressources en VRAM et transforme une image d'après la demande de l'utilisateur en utilisant les streams. Chronomètre l'opération de transformation. Peut utiliser la mémoire shared.

- **gpu_utilities.cuh** : Définit la classe **GpuUtilMenuSelection** qui permet de récupérer et d'afficher les paramètres d'entrée du programme. Contient également 1 struct :
  - **GpuUtilExecutionInfo** : Donne des informations sur le traitement à effectuer, avec la matrice de convolution et le nombre de passes à effectuer. Ajoute également une variable relative au nombre de streams utilisés et une seconde variable de type dim3 pour la répartition des threads.
- **gpu_utilities.cuh** : Implémentation de la classe **GpuUtilMenuSelection**.

3. Benchmarks

Premier relevé :

Les benchmarks sont effectués sur la machine distante. Pour chronométrer, nous avons utilisé le std::chrono côté CPU et les évènements CUDA côté GPU. Le test effectué a toujours été le même : nous effectuons le traitement d'image repoussage (emboss 3*3) 10 fois sur une image de taille 1920*1200, puis nous relevons le temps nécessaire à cette opération et enfin nous répétons cette opération 10 fois afin d'obtenir une moyenne.

Deuxième relevé :

Nous nous basons sur la 1ère approche mais répétons 100 fois l'opération. De plus, nous prenons en compte que la machine peut être utilisée par plusieurs utilisateurs :

- Etant donné le large paramétrage du programme GPU (cf. 3.B.), nous avons mesuré le temps entre chacune des 100 répétitions et avons refait les tests lorsque la répétition était plus longue que la moyenne, afin de ne pas biaiser les tests dû à un autre utilisateur sollicitant la même machine.
- Cela ne suffisant pas, nous avons effectué certaines transformations sur les données de nos benchmarks. Pour chaque ensemble de résultat d'une configuration, nous avons pris la moyenne des 30 % meilleurs temps, et avons supprimé sur les 70 % des valeurs restantes celles qui présentées une différence supérieure de 20 % à notre moyenne.

A. CPU

Côté CPU, nous avons des temps relativement lent. En moyenne, le temps nécessaire au traitement est de **2 secondes**.

B. GPU

Concernant les temps de traitement GPU, les mesures sont assez bonnes. Nous avons effectué les tests sur un paramétrage très large, soit pour des dimensions X et Y allant de 64 à 1024 (par pas de 64) chacune (soit 256 combinaisons), en combinant cela avec la mémoire shared seulement, 2 streams seulement, 4 streams seulement et enfin 4 streams et mémoire shared réunis. La totalité des mesures se trouvent dans un fichier Excel joint à l'archive du projet.

De façon générale, les mesures révèlent des problèmes d'implémentation de notre part. En effet, que ce soient les streams ou la mémoire shared, ils ralentissent légèrement l'exécution du traitement plutôt que de le raccourcir..

Nous n'avons pas compris les problèmes d'implémentation malgré nos recherches.

Aussi, malgré le nombre élevé de répétitions, les mesures entre les différentes configurations X et Y sont très éparses et ne semblent pas présenter de cohérence. Aucun schéma ne se dégage nous permettant de préférer avec certitude certaines valeurs X et Y à d'autres.

C. Comparaison et analyse

Nous sommes étonnés par les relevés que nous avons eu. Comme décrit en introduction du grand 3, nous avons avons réalisé 2 benchmarks, basés sur 2 approches différentes (d'où la raison des 2 fichiers excel).

La première approche n'était constituée que de 10 répétitions contre 100 pour la 2ème approche. Aussi, nous n'avons pas filtré les résultats éparses de la 1ère approche, là où nous avons écarté les valeurs incohérentes (càd présentant une différence >20 % par rapport à la moyenne des 30 % meilleurs temps. Cet "élagage" est indépendant à chaque configuration).

Ce qui nous a clairement le plus surpris, c’est la nette différence des relevés entre les 2 benchmarks alors que nous avons effectué le relevé avec le **même programme**. (Mais pas à la même heure, **seule différence**).

Pour mieux comprendre, voici un tableau présentant la moyenne générale des temps (càd toutes configurations confondues) de la 1ère et 2ème prises :

	1ère prise (en ms)	2ème prise (en ms)
Standard	0.0083532	0.00697667
Shared	0.008383075	0.00699112
2 Streams	0.042010725	0.00735224
4 Streams	0.11040245	0.00734064
4 Streams & Shared	0.110745575	0.00727213

Nous observons une très nette différence. Notamment pour l’option "4 Streams & Shared" où le temps d’exécution de la 1ère à la 2ème prise est **15 fois supérieur**.

A noter que les fichiers des mesures obtenues lors de la 1ère prise ont été supprimé mais versionnés, ils sont sûrement récupérables. En revanche, les fichiers des mesures de la 2ème prise sont complètement disponibles et toujours présent dans le dépôt.

4. Difficultés rencontrées

Les principales difficultés rencontrées furent sur l’implémentation des shared et streams. D’ailleurs, comme indique les benchmarks, il semblerait qu’ils ne fonctionnent pas complètement.

Pour les streams, une importante difficulté a été de répartir l’image à travers les différents streams puis de la récupérer. Nous avions un problème récurent de l’apparition d’une ou deux lignes de pixels noirs entre chaque section d’image de streams.

Aussi, debugger en CUDA est une opération particulièrement délicate. Nous ne disposons que du printf pour vérifier les valeurs. Sachant que tous les threads peuvent appeler le printf, il faut encadrer ce printf avec une condition et viser seulement certains threads, qui ne sont parfois pas les bons…

Enfin, ces benchmarks ont été difficiles à mettre en place et encore plus difficiles à comprendre (si jamais vous avez une idée, nous aimerions vraiment savoir pourquoi nous avons obtenu une telle incohérence dans nos résultats).