Master Informatique, Synthèse d'Images et Conception Graphique

Rapport de Projet GPGPU

BEC Hugo
KADRI Marouen

04 Décembre 2021

Table des figures

1 Image égalisée 2
2 Image Initiale 2
3 (a) HSV Image (b)La composante S (c) La composante V (d) La composante H 3
4 Image traitée 4
5 Lancement du programme 6
6 Temps d'exécution 7

Table des matières

1 Introduction **2**
1.1 Contexte général 2

2 Conversion :RGB2HSV & HSV2RGB **3**
2.1 Résultat 4

3 Histogramme **5**

4 Répartition **5**

5 EQUALIZATION **6**

6 Conclusion **6**

7 Bibliographie **7**

1 Introduction

L'égalisation d'histogramme est une méthode d'ajustement du contraste d'une image numérique qui utilise l'histogramme. Elle consiste à appliquer une transformation sur chaque pixel de l'image, et donc d'obtenir une nouvelle image à partir d'une opération indépendante sur chacun des pixels. Cette transformation est construite à partir de l'histogramme cumulé de l'image de départ. [3]

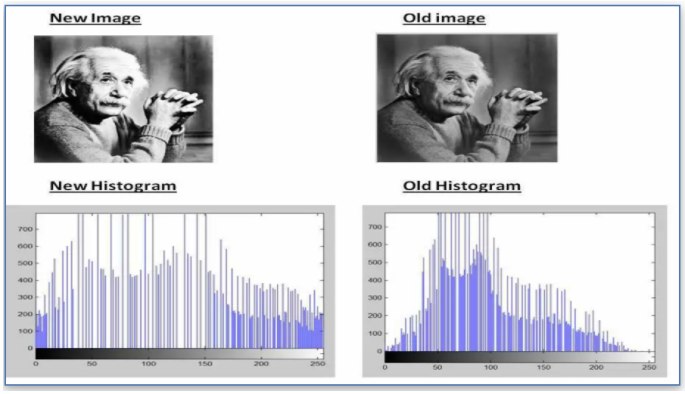


FIGURE 1 – Image égalisée

1.1 Contexte général

L'objectif de ce projet est de coder l' ´égalisation d'histogramme sur des images couleurs en CUDA.

FIGURE 2 – Image Initiale

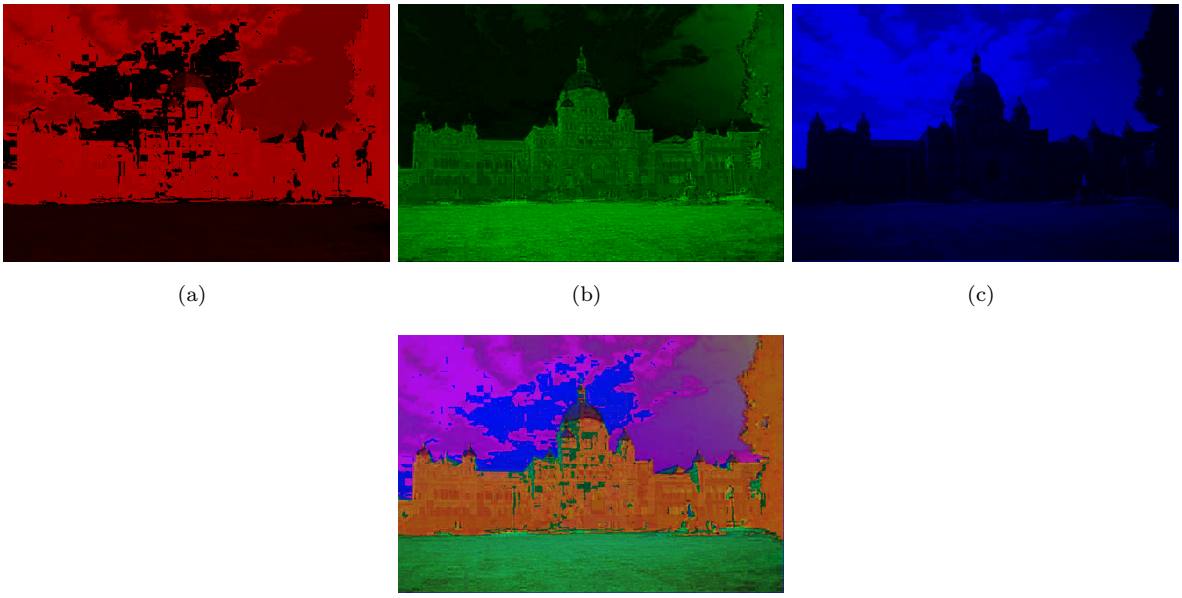
On va implemnter les fonctions suivantes en CUDA

1. Conversion RGB en espace HSV
2. Calculer l'histogramme
3. Répartition de l'histogramme calculé
4. Égalisation de l'histogramme
5. Conversion HSV en espace RGB

2 Conversion :RGB2HSV & HSV2RGB

Algorithm 1 Conversion RGB pixel to HSV

Require: $SizeOfImage \neq null$
Ensure: $n = SizeOfImage$
  while $i \neq SizeOfImage$ **do**
    $R \leftarrow i$
    $G \leftarrow i + 1$
    $B \leftarrow i + 2$
    $V \leftarrow max(R, G, B)$
    if $max(R, G, B) = min(R, G, B)$ **then**
      $H \leftarrow 0$
      $V \leftarrow 0$
    else
      $S \leftarrow (max(R, G, B) - min(R, G, B)/max(R, G, B)$
    end if
    if $max = R$ **then**
      $H \leftarrow 60 \times (0 + (G - B)/(max - min)$
    end if
    if $max = G$ **then**
      $H \leftarrow 60 \times (2 + (G - B)/(max - min)$
    end if
    if $max = B$ **then**
      $H \leftarrow 60 \times (4 + (G - B)/(max - min)$
    end if
    if $H < 0$ **then**
      $H \leftarrow H + 360$
    end if
    $i \leftarrow Channel + i$
  end while



(a) (b) (c) (d)

FIGURE 3 – (a) HSV Image (b)La composante S (c) La composante V (d) La composante H

Algorithm 2 Conversion HSV pixel to RGB

$c \leftarrow v * s$
$m \leftarrow v - c$
$x \leftarrow c * \left(1 - |\frac{h}{60} \mod 2 - 1|\right)$
$V \leftarrow max(R, G, B)$
if $(h \geq 0 \ \& \ 60 \geq h)$ **then**
  $RGB \leftarrow (c + m, x + m, m)$
end if
if $(h \geq 60.0 \ \& \ 120 \geq h)$ **then**
  $RGB \leftarrow (x + m, c + m, m)$
end if
if $(h \geq 120.0 \ \& \ 180.0 \geq h)$ **then**
  $RGB \leftarrow (m, c + m, x + m)$
end if
if $(h \geq 180.0 \ \& \ 240.0 \geq h)$ **then**
  $RGB \leftarrow (m, x + m, c + m)$
end if
if $(h \geq 240.0 \ \& \ 300.0 \geq h)$ **then**
  $RGB \leftarrow (x + m, m, c + m)$
end if
if $(h \geq 300.0 \ \& \ 360.0 \geq h)$ **then**
  $RGB \leftarrow (c + m, m, x + m)$
end if

2.1 Résultat

FIGURE 4 – Image traitée

- **rgb2hsv_v1** : Pour ce qui concerne l'adaptation GPU du calcul de conversion rgb vers hsv et inversement, nous avons commencer par calculer la valeur de Hue (teinte) dans une intervale entre 0 et 360 et la saturation S et la valeur V entre 0 et 100. nous avons simplement adapté le code de façon à se que chaque thread effectue la conversion d'un pixel. Nous avons ajouté une boucle while dans le cas où le nombre de thread serait insufisant par rapport au nombre de pixel à traiter. Ainsi par exemple le thread 4 va effectuer la conversion du pixel 4 puis du pixel blockDim.x*gridDim.x + 4 si ce pixel existe et ainsi de suite.

- **rgb2hsv_v2** : Or pour revenir en RGB notre convertisseur divisait de nouveau les valeurs pour obtenir un flottant entre 0 et 1. Nous avons donc supprimé ces multiplication par 360 et 100 pour chacune des 3 valeurs dans un flottant compris entre 0 et 1 ce qui nous permet d'économiser quelques calculc par conversion.

- **rgb2hsv_v3** : Nous avons réfléchi à utiliser de la mémoire partagé mais nous n'avons pas trouvé de manière intéressante de l'utiliser. En effet, nous avons pensé à charger respectivement les valeurs RGB et HSV dans la mémoire partagé pour effectuer les conversion mais en sachant que les valeurs sont utilisé qu'une seule fois, cela revient à charger ces éléments dans la mémoire partagé pour rien. Egalement nous ne pouvions séparer les taches car la conversion dépend de du chargement de ces valeurs donc impossible de gagner du temps par cette méthode.

3 Histogramme

- **Compute_hist_v1** : Nous avons réalisé une première version GPU naïve en utilisant uniquement la mémoire globale. Nous avons simplement adapté le code de façon à se que chaque thread ajoute la composante de son pixel dans un tableau d'histogramme global. Nous avons de nouveau ajouté une boucle while dans le cas où le nombre de thread serait insuffisant par rapport au nombre de pixel à traiter.

- **Compute_hist_v2** : Nous avons ensuite réalisé une seconde version en utilisant la mémoire partagé. Chaque thread incrémente sa valeur dans le tableau partagé de son block puis une fusion de chacun de ces tableau est opère dans l'histogramme de la mémoire globale.

- **Compute _hist_v3** : Ici chaque block charge les valeurs v dont elles ont besoin dans leur mémoire partagé pour travailler. Également le kernel est adapté pour un nombre de thread fixe de 1024, grâce à ça nous avons pu enlever des calculs inutile à cause des while. Nous avions cherché à initialiser les tableaux cache_v et cache_hist de manière parallèle mais par définition cache_v a un nombre de cases égal au nombre de thread d'un block donc il y a forcément des threads qui se retrouveront à effectuer l'initialisation des deux tableaux, donc, notamment à cause du syncthreads() nécessaire pour la suite, cette parallélisation demeure inintéressante. De la même manière, si on dédit une moitié à copier une case du tableau cache_hist et une autre moitié à copier chacun 2 cases du cache_v, le problème que un thread effectue 2 opération demeure.

4 Répartition

- **Compute_répartition_v1** : C'est une fonction GPU naïve appelé par compute_equalization_v1, une copie littérale de la version sequentielle.

- **compute_repartition_v2** : Au lieu de recalculer à chaque fois la repartition pour chacun des pixels. Nous calculons une et une seule fois chacune des valeurs possible que nous stockons dans un tableau de la même taille que l'histogramme initialisé auparavent dans la mémoire GPU. Le calcul demeure malgré ça identique à la version 1 mise à part que chacune des valeurs est effectué par un thread.

- **compute_repartition_v3** : Pour cette 3ème version, vu que chacune des valeurs du tableau de répartition dépend de la valeur précedente, nous avons essayé de calculer de cette manière avec rep[n] = rep[n-1] + hist[n]. Vu que chaque valeurs dépend de la précedente, nous ne pouvons parralélisé le calcul, nous avons donc essayé de créer un kernel pour un seul thread.

- **Compute_repartition_v4** : Même si la 3ème version demande moins de calcul, elle demeure plus lente que la 2nd version. Ainsi nous sommes reparti dans sa direction en essayant d'amélioré le temps de calcul. Ainsi, à la place de calculer pour une valeur l avec une simple boucle de O(n), nous avons utilisé la parralélisation. Chaque thread aditionne 2 éléments, puis la moitié des threads fait additionne ces sommes et ainsi de suite. Grâce à cette technique, nous arrivons à un calcul en O(log2(n)) pour chaque calcul de valeur du tableau de répartition. Egalement évité les conflits de bank dans la mémoire partagé, chaque thread copie la valeur d'index tid avec l'index tid + stride, stride étant une variable se divisant en 2 jusqu'à 1.

- **Compute_repartition_v5** : Enfin la 5ème version ajoute une optimisation de warp. Dès que cela est possible, c'est à dire dès que le nombre de thread actif passe en dessous de 32, on consacre le calcul sur un seul warp grace à une fonction GPU warpReduce.

5 EQUALIZATION

- **Compute_equalization_v1** : Chaque thread calcul la répartition V d'un pixel en utilisant compute_repartition_v1

- **Compute_equalization_v2** : Effectue la même chose mise à part qu'elle utilise le précalculer de la repartition par compute_repartition_v2+.

6 Conclusion

Nous avons pu atteindre l'objectif demandé de notre projet avec application des méthodes d'optimisation que nous avons vu pendant les séances de TP

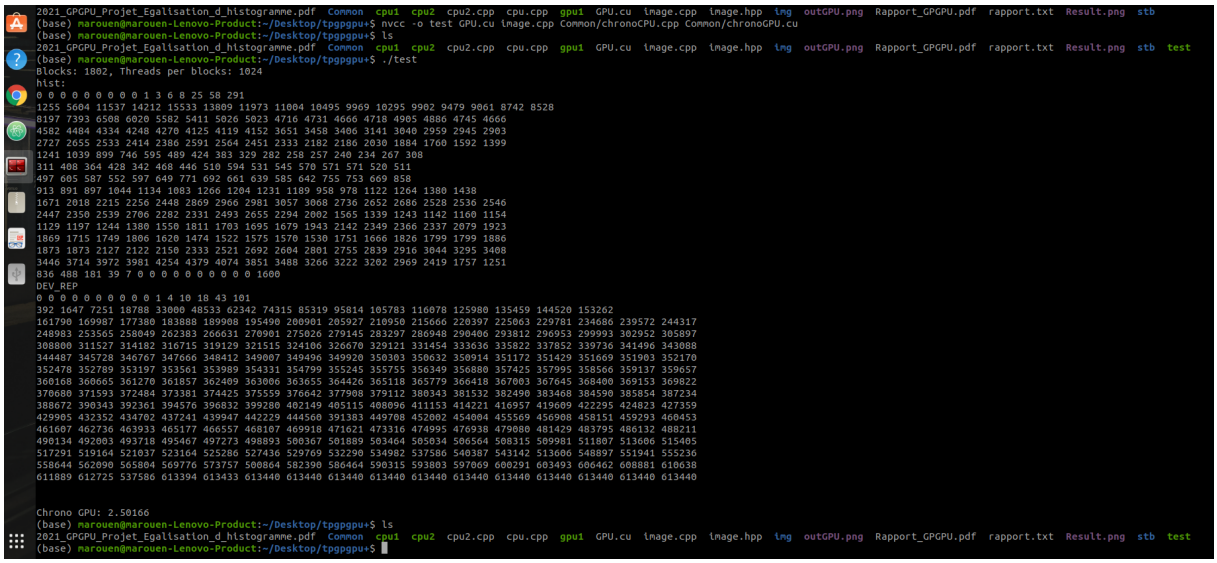


FIGURE 5 – Lancement du programme

```
rgb2hsv_v3: -0.05ms
compute_hist_v2: -0.1ms
compute_hist_v3: -0.02ms
compute_repartition_v2: -1ms
compute_repartition_v3: +0.2ms
compute_repartition_v4: -0.2ms
compute_equalization_v2: -1ms
```

FIGURE 6 – Temps d'exécution

En terme de perspectives, le projet dans sa version actuelle n'est pas encore optimal : Nous avons constaté le problème suivant :**Non-coalesced global loads and stores** au vu de l'utilisation d'**Unsigned char**

Solution proposée :

— On peut combiner/ decombiner les chars en groupe de 32bits

7 Bibliographie

Ci-dessous une collection de références vers les différentes ressources m'ayant servi lors de la réalisation de ce projet

Références

[1] *How to Fix a Low-Contrast Color Image using Histogram Equalization in GPU (CUDA) and C++. Disponible sur :*
`https://www.youtube.com/watch?v=yFhKT3_qr7k`

[2] STACKOVERFLOW.. *In CUDA, what is memory coalescing, and how is it achieved ?*
`https://stackoverflow.com/questions/27371941/nvprof-application-not-found?rq=1`

[3] WIKIPEDIA. *Égalisation d'histogramme*
`https://fr.wikipedia.org/wiki/%C3%89galisation_d'histogramme`

[4] *color-conversion-algorithms.js*
`https://gist.github.com/mjackson/5311256`

[5] *HSL and HSV*
`https://en.wikipedia.org/wiki/HSL_and_HSV`