
Master Informatique, Synthèse d'Images et Conception Graphique

# Rapport de Projet GPGPU

*Florian AUBERVAL*
*Célestin MARCHAND*

Janvier 2023

# 1 Introduction :

Dans le cadre du cours de "Développement sur GPGPU", nous avions pour projet d'implémenter l'égalisation d'histogramme d'une image sur GPU avec le langage CUDA. Ce langage est une propriété de l'entreprise NVIDIA et ne fonctionne donc uniquement que sur les cartes que celle-ci produit.

Ce rapport présente les différentes évolutions du code et fait la comparaison entre les différentes versions.

Le dépôt GitHub du projet: `https://github.com/Me-k-01/Histogram_equalization`

# 2 Égalisation d'histogramme. Qu'èsaquo ?

L'égalisation d'histogramme dans le domaine de l'imagerie consiste à augmenter les contrastes d'une image et ainsi améliorer sa «compréhension». Pour réaliser ce traitement, plusieurs calculs sont nécessaires. En effet, l'égalisation d'histogramme ne se fait pas dans le domaine RGB (RVB en français) d'une image mais dans le domaine HSV (TSV en français). Ce domaine, qui permet tout autant de représenter un image, définit les pixels d'une image via d'autres informations que sont : la teinte, la saturation et la valeur (ou intensité). Ces sur cette dernière information que la modification a lieu.

Cependant, la plupart des images en informatiques sont aujourd'hui définies dans le domaine RVB. Il faut donc dans un premier temps passer de ce domaine à celui du HSV. Après ce travail, vient le calcule de l'histogramme lié à la Valeur. Cela permet ensuite de calculer la répartition des Valeurs des pixels de l'image. Cette répartition va nous permettre ensuite de calculer les nouvelles Valeurs de chaque pixels. Une fois cette étape faite, il ne reste plus que à retransformer les valeurs des pixels du domaine HSV vers le domaine RGB et admirer le résultat.

# 3 Versions CPU :

Dans un premier temps, il est important de commencer par des versions «naïves»sur CPU qui permettent d'avoir une base solide pour comparer les différentes versions codées sur GPU ainsi qu'une version fonctionnelle.

Pour l'égalisation d'histogramme d'une image, cinq fonctions sont nécessaires. En voici la liste :

- **RGB vers HSV** : Une fonction permettant de passer des valeurs RGB d'une image au valeurs HSV équivalentes

- **Calcule d'Histogramme** : Une fonction permettant de calculer l'histogramme d'un tableau de valeurs

- **Calcule de Répartition** : Une fonction permettant de calculer la répartition d'un histogramme

- **Calcule d'Égalisation** : Une fonction permettant d'égaliser les Valeurs des pixels d'une image

- **HSV vers RGB** : Une fonction permettant de passer des valeurs HSV d'une image au valeurs RGB équivalentes

Pour une meilleure compréhension du projet, chacun d'entre nous à codé toutes les fonctions. Plusieurs versions de chacune des fonctions sont donc disponibles et peuvent être testées séparément[1].

En moyenne, sur 15 itérations, les fonctions[2] s'exécutent en :

- **RGB vers HSV** : 29.416 millisecondes

- **Calcule d'Histogramme** : 5.385 millisecondes

- **Calcule de Répartition** : moins de 0.001 millisecondes

- **Calcule d'Égalisation** : 6.131 millisecondes

- **HSV vers RGB** : 15.980 millisecondes

Ces calculs de moyennes ont été effectués avec une image de taille 961 par 640 pixels[3] sur l'ordinateur 15 de la salle i212.

Nous pouvons voir que le calcul de la répartition d'un histogramme est déjà optimisé pour CPU. Cela s'explique par l'itérativité du calcul. A l'inverse, nous pouvons voir que le reste des calculs paraît plutôt lent en comparaison et est dû à la taille des données à traiter.

[1] La définition des fonctions de chaque personne sont trouvables dans le fichier : ./cpu/"Nom de la personne qui les a réalisées"/histoCPU.cpp

[2] Les fonctions utilisées pour effectuer le calcul du temps d'exécution moyen sont celles du fichier : ./cpu/celestin/histCPU.cpp et sont appelées par le fichier ./cpu/celestin/main.cpp

[3] Cette image est trouvable à cet endroit : ./img/chateau.png

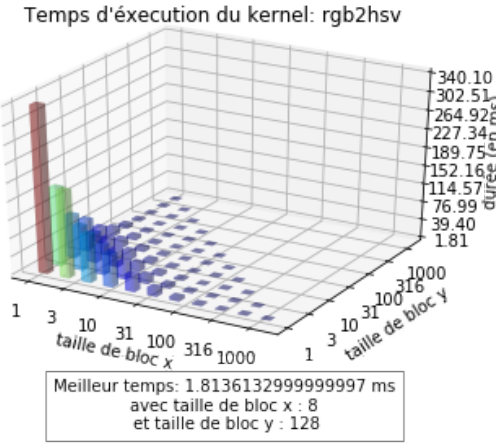
# 4 Versions GPU :

Une fois les versions CPU codées, la traduction en codes CUDA pour GPU est plutôt rapide à faire. Cependant, ce ne sont que des versions «naïves»adaptées depuis des versions séquentielles. Nous pouvons donc améliorer les performances en mettant en place quelques optimisations individuelles sur chacun des kernels. Pour le choix des kernels et des tailles de blocs, nous allons faire des benchmark avec une taille de grille égal à 1, afin d'évaluer si les kernels bénéficient de bloc en 2D.

Les dimensions des blocs que nous testerons seront des puissances de 2, allant de 1 à 1024, avec une échelle d'affichage exponentielle sur l'axe x et y afin d'être précis sur les petits nombres, et d'avancer rapidement vers les grands nombres. Le nombre total de threads ne dépassera pas 1024 dans nos tests.

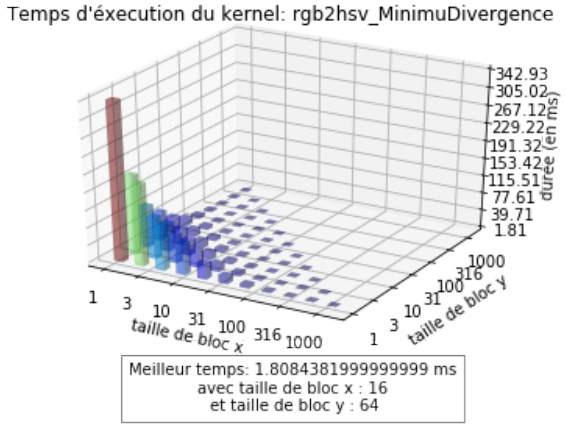
Les évaluations qui suivent sont les moyennes des durées de chaques kernels sur 10 essaies chacun.

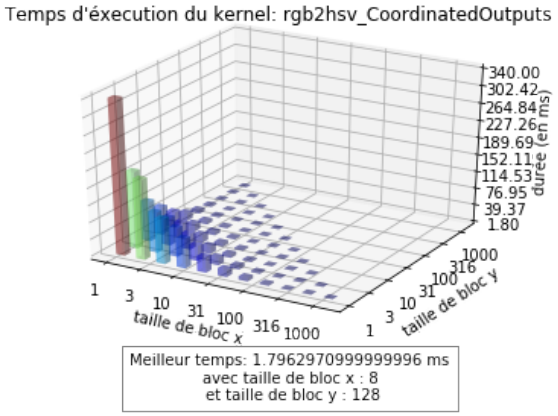
## 4.1 Convertion RGB vers HSV :




Pour améliorer le kernel de convertion, nous pouvons tenter d'éviter la divergence de code, en transformant les instructions conditionnels en opérateur logique dans des formules que chaque kernel exécutera. C'est le principe de fonctionnement de l'amélioration rgb2hsvWithMinimumDivergence.

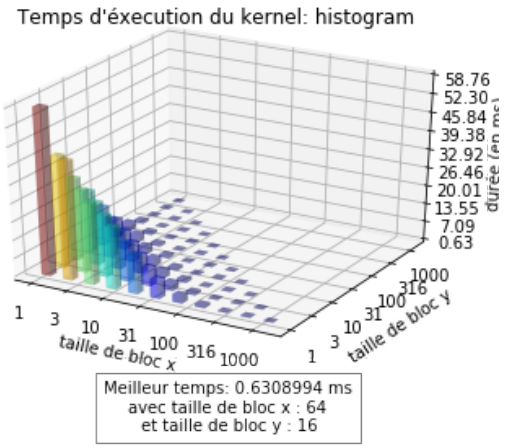
Ensuite, nous pouvons tenter d'optimiser en faisant en sorte que les accès à la mémoire se fassent en même temps pour les threads afin de minimiser l'impacte de la divergence qu'engendre les conditionnels que nous ne pouvons pas retirer. C'est le principe du kernel rgb2hsvCoordinatedOutputs



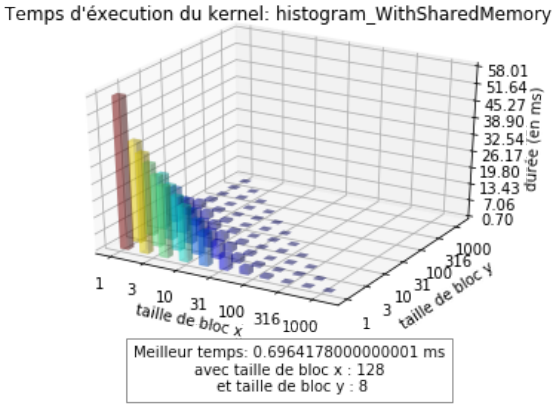




Le meilleur kernel est "rgb2hsvCoordinatedOutputs", avec une différence de 0.02 micro secondes.

## 4.2 Calcul d'Histogramme :

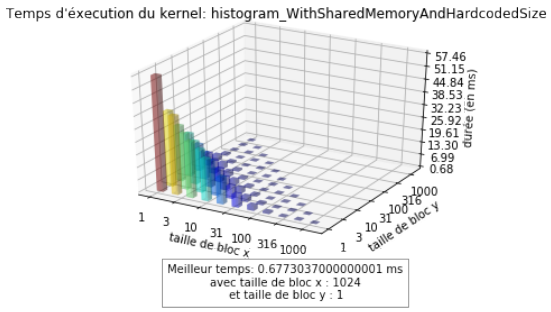


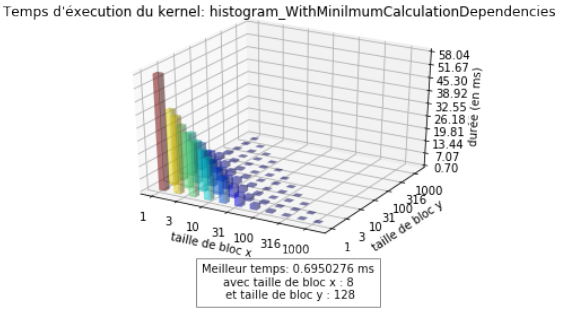

Pour améliorer le kernel de génération de l'histogramme des valeurs de l'image, on peut tirer avantage de la mémoire partagé, avec des histogrammes partiels, tel qu'inspiré par: `https://developer.nvidia.com/blog/gpu-pro-tip-fast-histograms-using-shared-atomics-maxwell/` C'est amélioration qui est faite par le kernel histogramWithSharedMemory




De plus, nous pouvons tenter de tirer parti de la mémoire constante, en enregistrant la taille de l'histogramme dans la mémoire global à tous les threads, avec une constante. C'est le kernel histogramWithSharedMemoryAndHardcodedSize qui implémente cette amélioration.

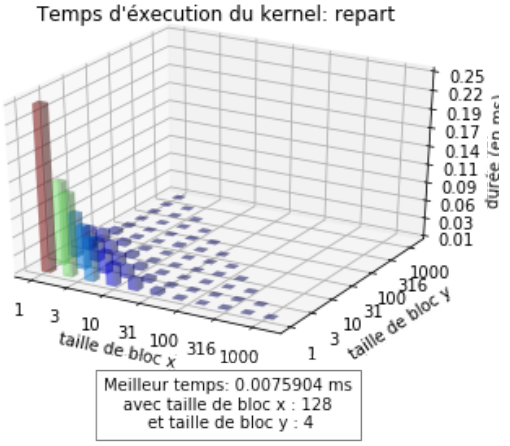
Par-dessus cela, nous pouvons retirer les dépendances de calculs, c'est le kernel histogramWithMinimumDependencies



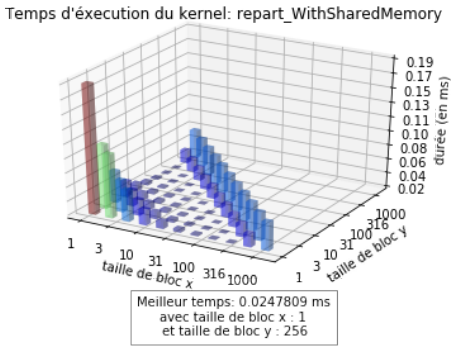



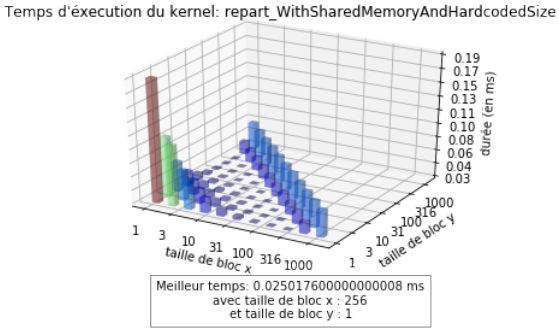

Le meilleur kernel est "histogram", les différentes optimisations n'ont pas améliorées le temps de calcul.

## 4.3 Calcul de la Répartition :



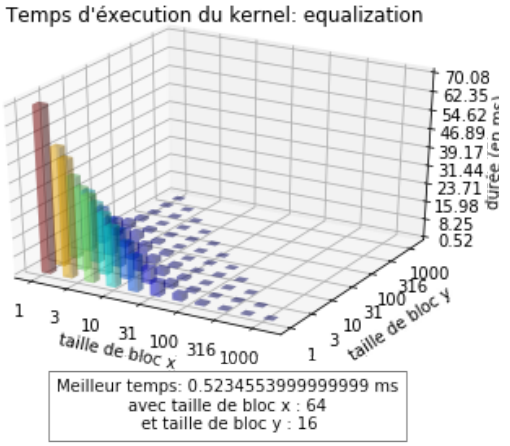

Pour améliorer le kernel qui applique la fonction de répartition de l'histogramme r(l), nous pouvons utiliser la mémoire partagée pour stocker le tableau d'histogramme (repartWithSharedMemory). Ainsi qu'avec l'utilisation de constante globale pour stocker la taille de l'histogramme (repartWithSharedMemoryAndHardcodedSize)



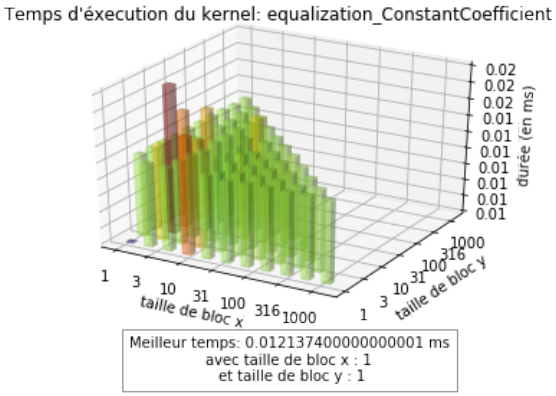




Le meilleur kernel est "repart", les autres versions n'apportent pas de meilleurs temps d'exécution. On remarque qu'il ne faut pas dépasser les 256 blocs pour les deux autres versions.

## 4.4 Calcul de l'Égalisation :

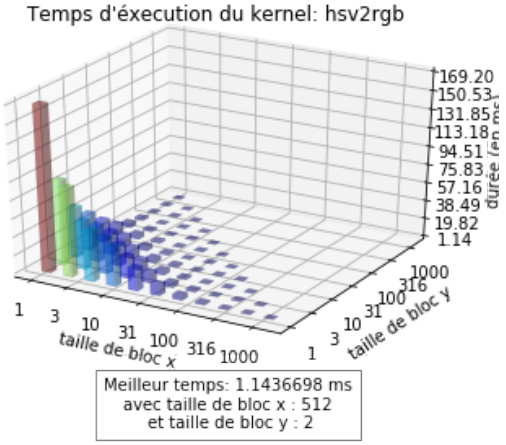


Pour améliorer le kernel qui étale l'histogramme à partir de la répartition, on peut éviter de recalculer pour chaque thread le coefficient. En effet, puisqu'elle ne change pas pour le calcul de la nouvelle valeur du pixel de l'image, nous pouvons la stocker en tant que constante pour tous les threads. C'est le rôle du kernel equalizationConstantCoefficient.

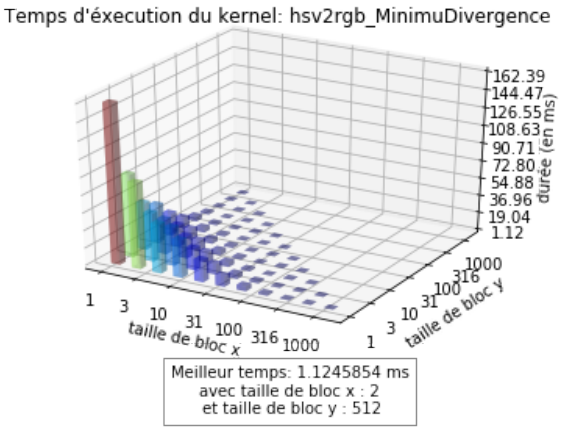



Le meilleur kernel est "equalizationConstantCoefficient", avec une différence de 0.4 milliseconde. On remarque qu'un bloc de taille 1 fonctionne le mieux pour le kernel optimisé, contrairement à celui sans l'optimisation.

## 4.5 Convertion HSV vers RGB :




Pour améliorer le kernel de convertion de l'image de format HSV en format RGB, nous pouvons éviter les divergences du code entre les thread en intégrant les conditionnels dans les opérations, de la même manière que l'on a procédé pour le kernel rgb2hsv. C'est le kernel hsv2rgbWithMinimumDivergence qui effectue cette version. On notera que pour ce kernel, cette optimisation est possible pour tous les branchements conditionnels.

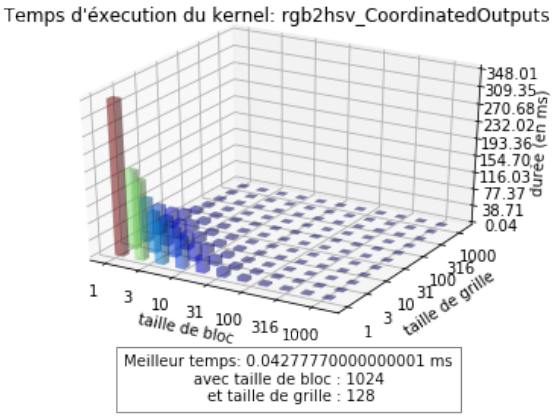



Le meilleur kernel est "hsv2rgbMinimumDivergence", avec une différence de 0.02 milliseconde.

# 5 Comparaison des résultats :

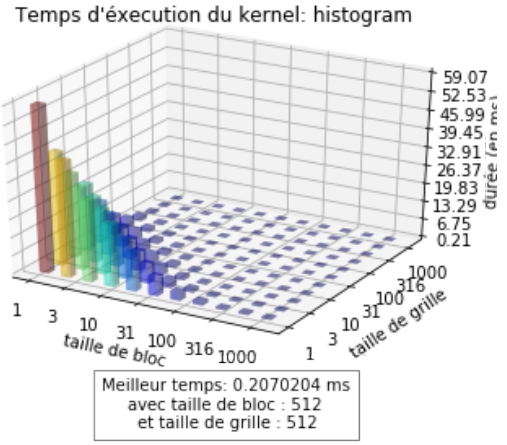
Comparons les meilleurs kernels avec leurs équivalent sur CPU. Nous essaierons dans cette partie de faire varier les grilles et les blocs sur une seul dimension étant donné qu'aucun des calculs ne bénéficient pas de la 2D. Cela nous permettra de garder des graphiques représentable en 3D. La taille de grille et de blocs sont des puissances de 2 allant de 1 à 1024, avec un affichage exponentiel. Les évaluations qui suivent sont les moyennes des durées de chaques kernels sur 10 essaies.

## 5.1 Convertion RGB vers HSV :



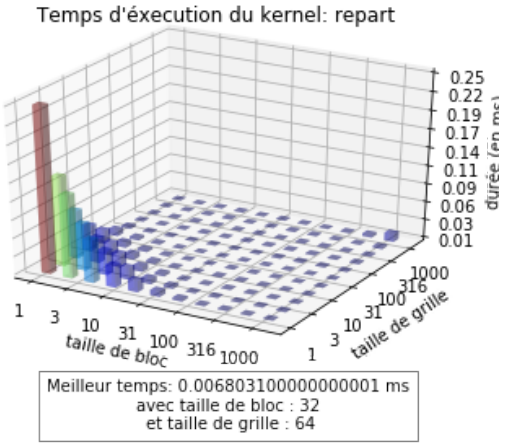

Soit 684 fois plus rapide que la version CPU.

## 5.2 Calcul d'Histogramme :



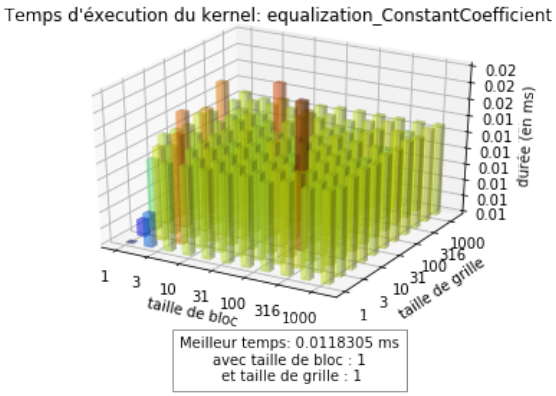

Soit 26 fois plus rapide que la version CPU.

## 5.3 Calcul de la Répartition :



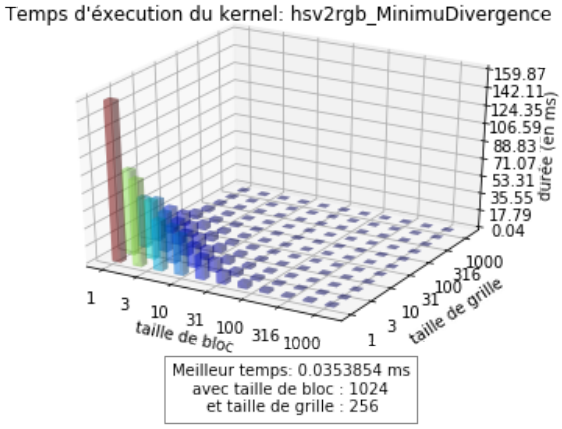

Soit plus de 6.8 fois moins rapide que la version CPU.

## 5.4 Calcul de l'Égalisation :




Soit 510 fois plus rapide que la version CPU.

## 5.5 Convertion HSV vers RGB :




Soit 457 fois plus rapide que la version CPU.

# 6 Conclusion :

En conclusion, les fonctions d'égalisation, et de convertion RGB/HSV et HSV/RGB bénéficient grandement de la parallélisation, et sont 500 fois plus rapide que leurs homologue CPU.

Le fait que la durée moyenne du kernel d'égalisation soit bien plus courte avec une taille de bloc très petite est difficile à expliquer. En toute logique, cela dire que l'on ne bénéficie pas de la parallélisation du calcul, mais on a une performance 500 fois supérieure à la version CPU lorsque le code est exécuté sur la carte graphique. C'est peut-être donc dû à la rapidité de la cadence de l'unité arithmétique de la carte graphique par rapport à la cadence du processeur, pour les machines de la salle i212?

La fonction d'histogramme bénéficie d'une bonne amélioration des performances, avec une durée moyenne du temps du kernel 26 fois plus courte que son équivalent CPU.

Le kernel de répartition possède des performances dégradées par rapport à sa version séquentielle. Avec une durée moyenne 7 fois moins rapide que la version CPU, c'est le seul cas qui ne bénéficie pas de la parallélisation du calcul, et se retrouve plus lents que sa version CPU.