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 une 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 aux 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 aux 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 est trouvable dans le fichier : ./cpu/"Prénom 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

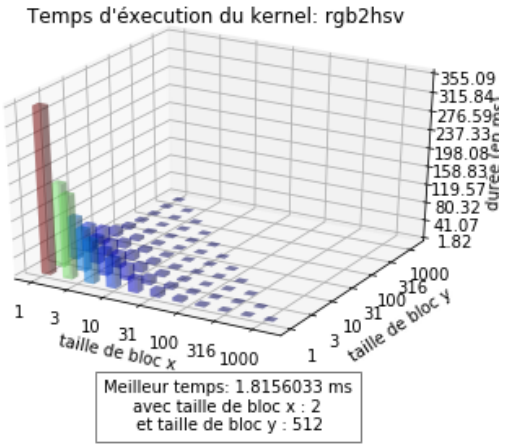
4 Versions GPU :

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 benchmarks 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, pour des tailles 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.

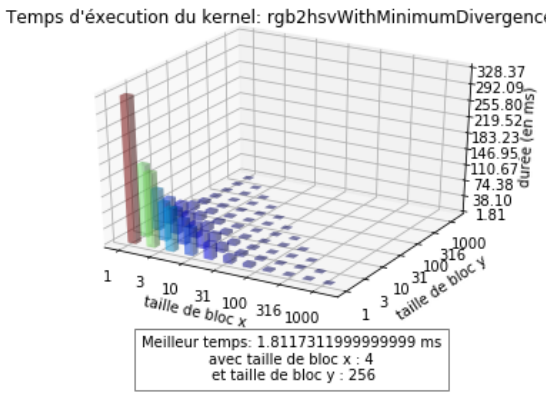
Les évaluations qui suivent sont les moyennes des durées de chaque kernel sur 10 essaies chacun.

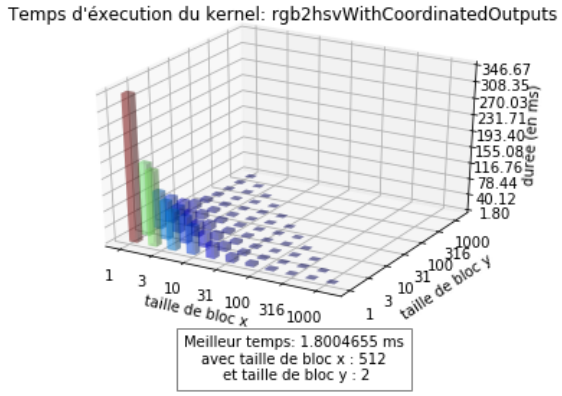
4.1 Convertion RGB vers HSV :



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.

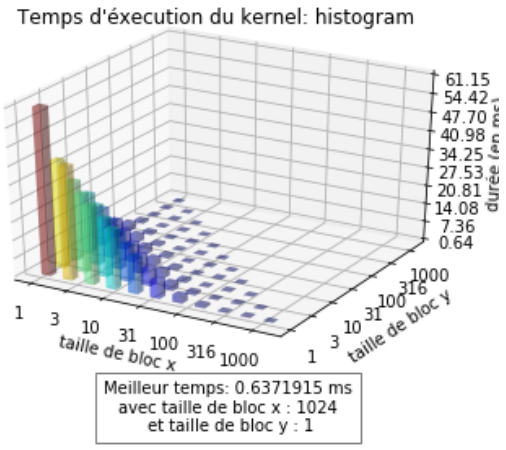
Ensuite, nous pouvons tenter d'optimiser en faisant en sorte que les accès à la mémoire se fassent en même temps pour les threads afin de minimiser l'impacte de la divergence qu'engendre les conditionnels que nous ne pouvons pas retirer. C'est le principe du kernel rgb2hsvWithCoordinatedOutputs.



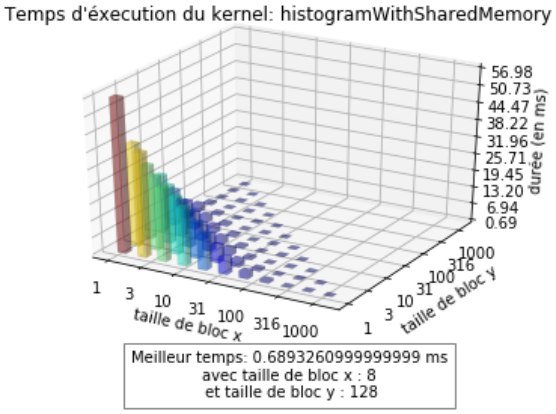


On remarque que la différence est minime entre les kernel (de lordre de la micro-seconde). Le meilleurs kernel est "rgb2hsvWithMinimumDivergence"

4.2 Calcul d'Histogramme :

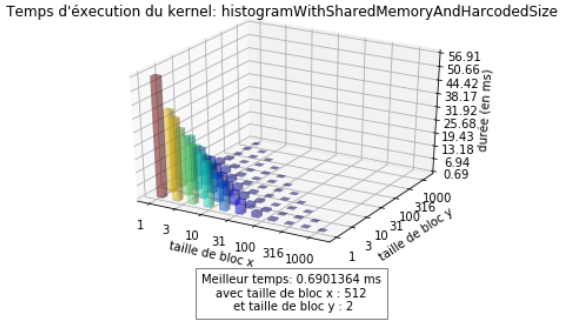


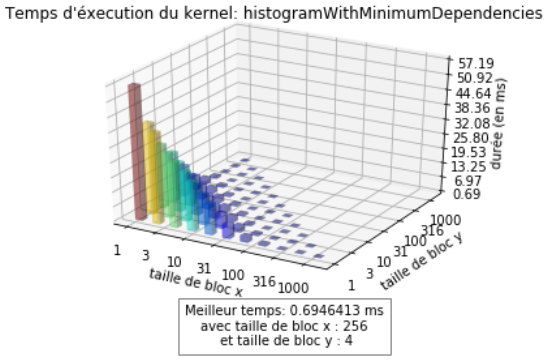
Pour améliorer le kernel de génération de l'histogramme des valeurs de l'image, on peut tirer avantage de la mémoire partagée, avec des histogrammes partiels, tel qu'inspiré par: `https://developer.nvidia.com/blog/gpu-pro-tip-fast-histograms-using-shared-atomics-maxwell/` C'est cette amélioration qui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



L'amélioration précédente utilisant une taille dynamique de mémoire partagée, il est intéressant de comparer le temps d'exécution avec une mémoire partagée dont la taille est static. C'est le kernel histogramWithSharedMemoryAndHarcodedSize qui implémente ce test.

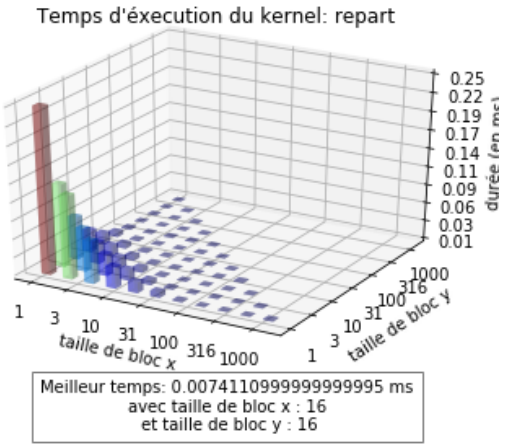
En dernier test, nous pouvons retirer au maximum les dépendances de calculs, c'est le kernel histogramWithMinimumDependencies.



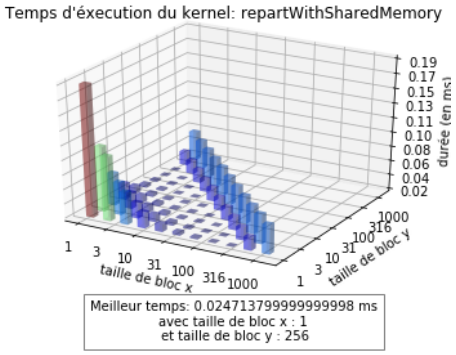


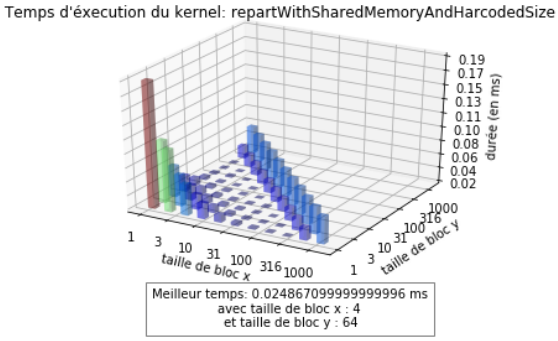
Le meilleur kernel est "histogram" sans l'optimisation, avec une différence d'environ 0.05 milliseconde environ

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).

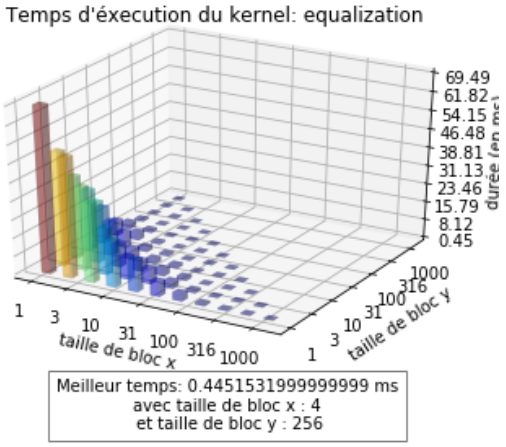




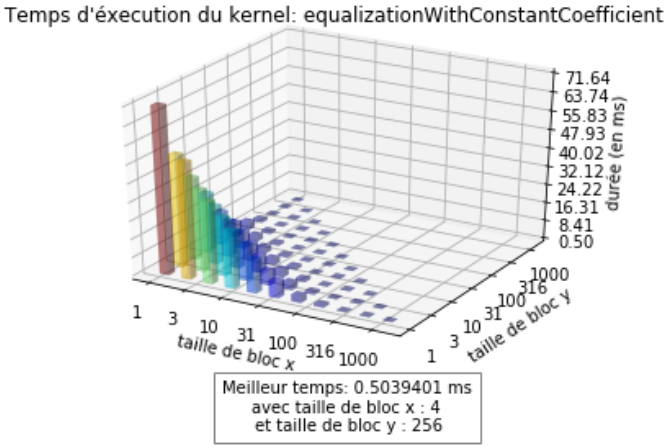
Le meilleur kernel est "repart" sans optimisation. On remarquera qu'il est meilleur de ne pas dépasser les 256 threads. Ce qui est logique puisqu'on n'a pas plus de 256 valeurs dans l'histogramme, donc au delà, il y a répétition de calcul, ce qui provoque des accès concurrents, et donc augmente le temps de calcul.

Cependant, nous aurions pu croire que le kernel utilisant la mémoire partagé aurait pu être plus rapide. En effet, de nombreux accès à la mémoire sont fait. La mémoire partagée étant plus rapide, cela aurait dû être rentable.

4.4 Calcul de l'Égalisation :

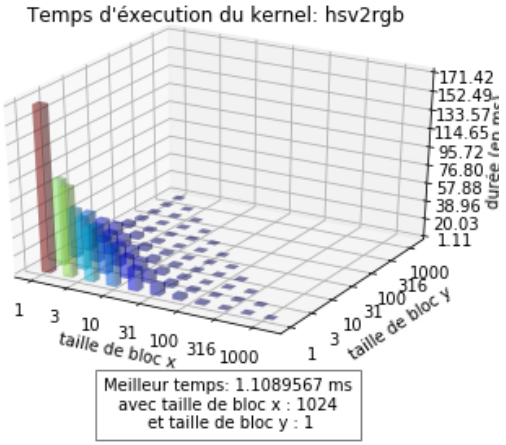


Pour améliorer le kernel qui étale l'histogramme à partir de la répartition, on peut éviter de recalculer pour chaque thread le coefficient. En effet, puisqu'il ne change pas pour le calcul de la nouvelle valeur du pixel de l'image, nous pouvons le stocker en tant que constante pour tous les threads.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

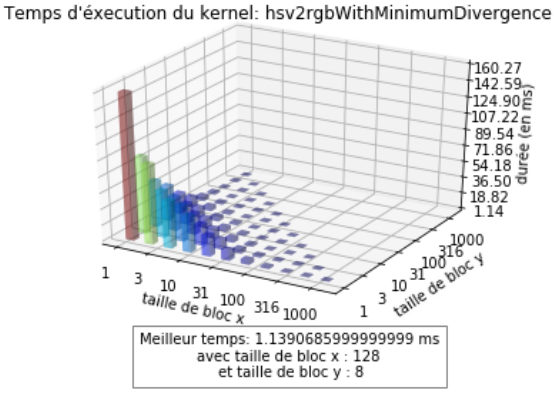


Le meilleur kernel est "equalization" sans optimisation, avec une différence de 0.05 milliseconde.

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 threads 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.

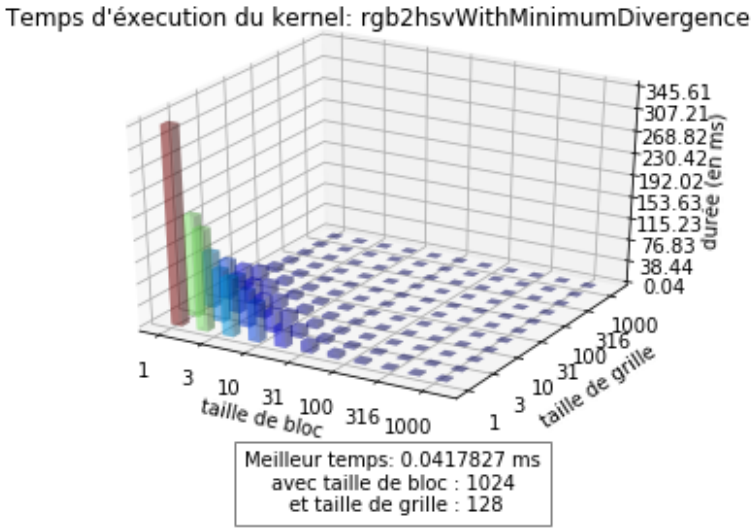


Le meilleur kernel est "hsv2rgb", avec une différence de 0.031 millisecondes.

5 Comparaison des résultats :

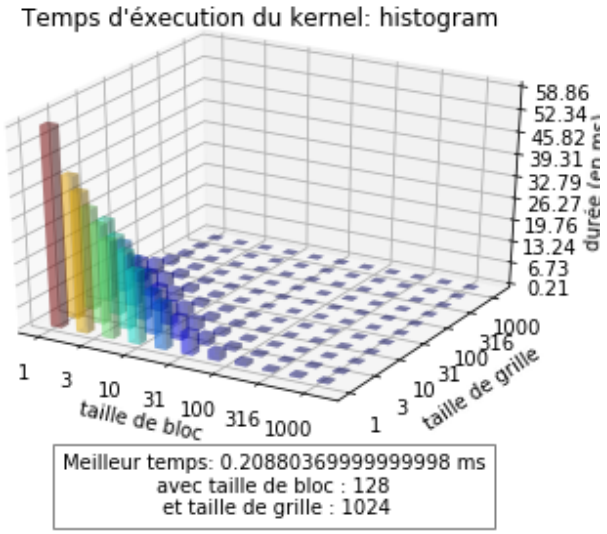
Comparons les meilleurs kernels avec leurs équivalent sur CPU. Nous essaierons dans cette partie de faire varier les grilles et les blocs sur une seule dimension étant donné qu'aucun des calculs ne bénéficient de la 2D. Cela nous permettra de garder des graphiques représentable en 3D. La taille de grille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

5.1 Convertion RGB vers HSV :



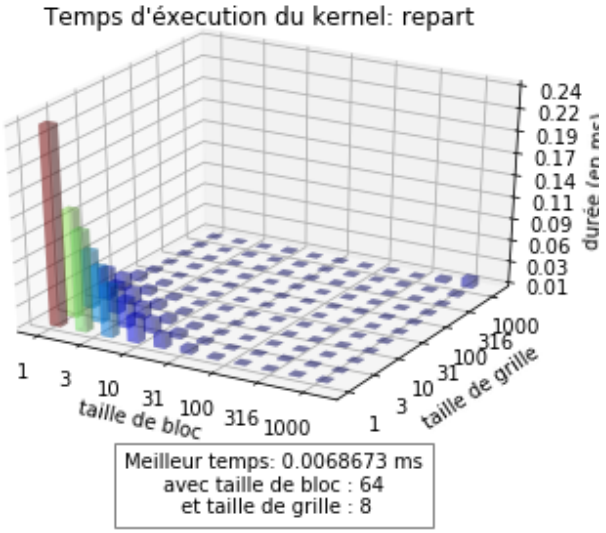
Soit 16 fois plus rapide que la version CPU.

5.2 Calcul d'Histogramme :



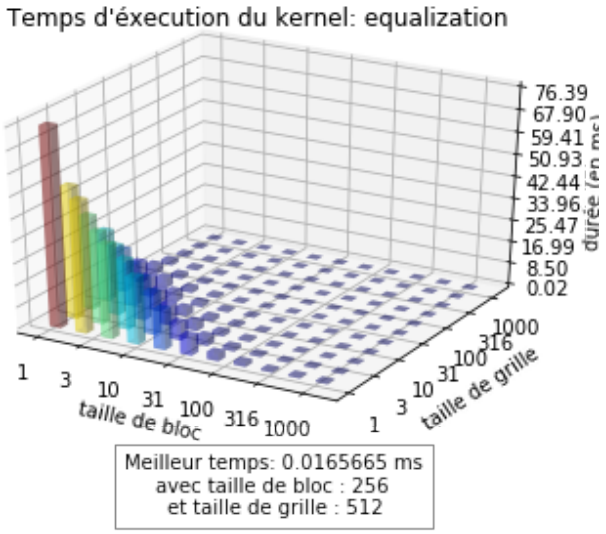
Soit environ 140 fois plus rapide que la version CPU.

5.3 Calcul de la Répartition :



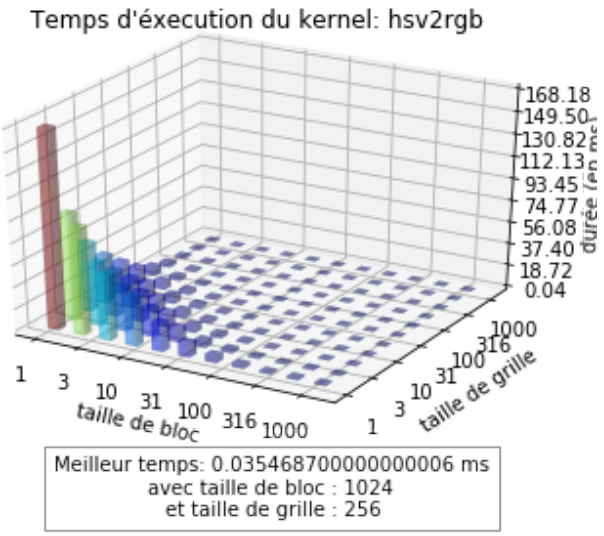
Soit un peu près de 7 fois moins rapide que sa version CPU.

5.4 Calcul de l'Égalisation :



Soit 370 fois plus rapide que la version CPU.

5.5 Convertion HSV vers RGB :



Soit 450 fois plus rapide que la version CPU.

5.6 Remarque importante

6 Conclusion :

En conclusion, presque toutes les fonctions en version CPU sont plus de centaines de fois plus rapide en étant parallélisées.

La plus grosse amélioration que l'on a pu obtenir est pour le code de conversion de HSV et RGB. Celui-ci est près de 500 fois plus rapides en version parallélisée.

La seule fonction qui n'a pas bénéficié de la transcription en code CUDA, est la fonction de répartition de l'histogramme. Elle est 7 fois plus lent que son homologue CPU. Cela est explicable par le fait qu'il faut soit effectuer des calculs redondants, soit faire de la synchronisation de threads pour gérer les accès mémoire.

Et enfin, la fonction de conversion du format RGB vers HSV est celle qui a eu l'amélioration la moins notable.

Les petites optimisations que l'on a pu écrire lors de l'élaboration du code n'ont que très peu aider à améliorer le temps de calcul.