
Rapport projet GPGPU 2023 - 2024
Par Emilien MENDES et Téo ZWIEBEL
Enseignants encadrants :
Cyprien PLATEAU–HOLLEVILLE
Vincent LARROQUE

Table des matières

1	Introduction	4
2	Visibilité	4
2.1	Version séquentielle	4
2.2	Version GPU non optimisé	6
2.3	Les différences CPU/GPU	9
2.3.1	Résultats	9
2.3.2	Modification de l'algorithme	9
2.4	Version GPU optimisé	9
2.4.1	Mémoire cache	9
2.4.2	Grid Stride Loop	10
2.5	Optimisation par analyse	11
2.6	Optimisation complète	11
3	Réduction d'image	14
3.1	Version séquentielle	14
3.2	Version GPU non optimisé	14
3.3	Version GPU optimisé	14
3.3.1	Utilisation de la mémoire partagée	14
3.3.2	Utilisation de mémoire constante	14
3.3.3	Optimisation par analyse	15
4	Conclusion	16
5	Annexes	17
5.1	Analyse du double kernel avec Nsight	17

Table des figures

1	Visualisation de la DDA sur un point p	5
2	Image obtenu à partir du point (245,497) sur le fichier 1.input.ppm sur CPU	6
3	Spécificité technique de la carte graphique en salle de TP	7
4	Image obtenue à partir du point (245,497) sur le fichier 1.input.ppm sur GPU	8
5	Erreur lors de la création d'un cache de trop grande taille (684×687)	9
6	Image obtenue à partir du point (245,497) sur le fichier 1.input.ppm pour le kernel optimisé	10
7	Utilisation des Pipeline sur Nsight avant optimisation	11
8	Utilisation des Pipeline sur Nsight après optimisation	12
9	Utilisation de la capacité du GPU pour le kernel optimisé par analyse	12
10	Utilisation de la pipeline LSU pour le kernel optimisé par analyse	12
11	Charte des accès mémoire pour le kernel optimisé par analyse	13
12	Utilisation de la capacité du GPU pour le kernel réduction d'image naïf	15
13	Utilisation des Pipeline pour le kernel réduction d'image naïf	15
14	Utilisation de la capacité du GPU pour le kernel réduction d'image optimisé	16
15	Utilisation des Pipeline pour le kernel réduction d'image optimisé	16
16	Utilisation de la capacité du GPU pour kernel_angle.calc	17
17	Utilisation de la capacité du GPU pour kernel_view_test.tab	17
18	Utilisation de la pipeline LSU pour kernel_angle.calc	17
19	Charte des accès mémoire pour kernel_angle.calc	18
20	Utilisation de la pipeline LSU pour kernel_view_test.tab	18
21	Charte des accès mémoire pour kernel_view_test.tab	19

1 Introduction

Lors de ce semestre nous avons étudié le fonctionnement de l'architecture GPU et de l'avantage de la parallélisation du traitement des données comparée au processeur qui traite les données de manière séquentielle. Ce projet traitera de la parallélisation sur des cartes de hauteur, qui seront représentés par des nuances de gris.

2 Visibilité

Dans cette partie l'objectif est de créer une carte en 2 dimensions représentant l'ensemble des points des points visibles depuis un point prédéfinie. Les cartes en entrées et en sortie sont en 2 dimensions afin de diminuer le temps et le nombre de calculs comparé à celles en 3 dimensions, mais l'utilisation d'une carte en hauteur permet tout de même de perdre assez peu d'informations.

2.1 Version séquentielle

Avant de coder l'algorithme sur GPU, il est intéressant de coder la version sur CPU afin d'avoir une comparaison du temps de calcul et des différences de l'approche CPU contre GPU.

Pour l'ensemble de cette partie, on utilisera les fichiers *ppm.hpp* et *ppm.cpp* qui permettent de convertir une image au format ppm en une matrice de pixels. Chacun des pixels sera représenté sous forme de **uint_8t** car les valeurs ne peuvent pas dépasser 255. L'implémentation de l'algorithme est décrit ci-dessous sous forme de pseudo code.

Algorithme 1 Visibilité

```
1: procédure VISUALISATIONPOINTS(Image entree, Point centre) :  
2:    $r \leftarrow a \bmod b$   
3:   Pour  $i$  allant de 0 à hauteur, faire :  
4:     Pour  $j$  allant de 0 à largeur, faire :  
5:        $z \leftarrow \text{hauteur}(i, j)$   
6:       Point  $p \leftarrow (i, j, z)$   
7:       visible  $\leftarrow$  vrai  
8:        $angle \leftarrow \angle(c, p)$  ▷ Calcul avec la formule de l'arc tangente  
9:       Pour tout  $p_{i,j} \in \text{DDA}$  Faire ▷ Angle pour les points intermédiaires  
10:         $angle_{p_{i,j}} \leftarrow \angle(c, p_{i,j})$   
11:        Si  $angle_{p_{i,j}} > angle$  alors  
12:          visible  $\leftarrow$  faux  
13:        Fin si  
14:        Si visible alors  
15:          sortie[pixel]  $\leftarrow$  blanc  
16:        Sinon  
17:          sortie[pixel]  $\leftarrow$  noir  
18:        Fin si  
19:      Fin pour  
20:    Fin pour  
21:  Fin pour  
22:  renvoyer sortie ▷ Renvoie l'image avec la visibilité depuis le centre  
23: Fin procédure
```

Les différents points $p_{i,j}$ sont calculés grâce à l'algorithme de type analyseur différentiel numérique de la droite entre le point courant et le centre.



FIGURE 1 – Visualisation de la DDA sur un point p

On calcule donc l'équation de la droite D sur laquelle on place un certain nombre de points $p_{i,j}$. Lors du déroulé de l'algorithme, les points obtenus ne seront pas toujours des valeurs entières (donc des pixels). On choisira de calculer les angles directement avec les valeurs flottantes plutôt qu'un arrondi car le résultat obtenu est meilleur.



FIGURE 2 – Image obtenu à partir du point (245,497) sur le fichier 1.input.ppm sur CPU

2.2 Version GPU non optimisé

Maintenant que la version séquentielle a été implémentée, on cherche à effectuer le même travail sur GPU afin de voir les bénéfices de la parallélisation du travail. Il faut donc commencer par définir l'architecture du programme pour traiter tous les pixels de l'image. Dans un premier temps, on ne prendra en considération uniquement l'image **1.input.ppm** donnée qui a une taille de 684×687 pixels. On discutera de l'efficacité et des éventuelles problèmes pour des images de taille plus grande lors de l'optimisation de l'algorithme.

Dans cette partie, la taille de bloc de grille, le nombre de bloc par grille et le nombre de threads par blocs ne sont pas un problème techniques conformément aux spécificités techniques de la carte graphique qui sera utilisée comme référence.

```
- registers per blocks: 65536
- warpSize: 32
- max threads dim: 1024, 1024, 64
- max threads per block: 1024
- max threads per multiprocessor: 1024
- max grid size: 2147483647, 65535, 65535
- multiProcessor count: 30
```

FIGURE 3 – Spécificité technique de la carte graphique en salle de TP

On commencera par donner un certain nombre de threads par blocs que l'on fixera en dur dans notre programme. Pour calculer le nombre de bloc et donc la taille de grille on utilisera les formules suivantes :

$$\text{Taille grille } x = \lceil \frac{\text{hauteur}}{\text{nombre de thread dans le bloc en } x} \rceil$$

$$\text{Taille grille } y = \lceil \frac{\text{largeur}}{\text{nombre de thread dans le bloc en } y} \rceil$$

$$\lceil x \rceil = \text{partie entiere superieure de } x$$

Maintenant que l'architecture est créée, on alloue de la mémoire sur le GPU avec **cudaMalloc** et **cudaMemcpy** de notre ensemble de pixels qui sont sous la forme d'un tableau 1D, ainsi que notre tableau de sortie sous la même forme.

Chacun des threads aura donc les paramètres suivant lors de l'appel du kernel :

1. Le tableau de pixels de l'image d'entrée
2. Le tableau de pixels de l'image de sortie
3. Le point central
4. Les dimensions de l'image

Chacun des pixels effectuera le travail de visibilité pour un seul point, en reprenant le même algorithme que pour la version séquentielle. Pour éviter un accès concurrent en écriture sur l'image de sortie entre les différents threads on indexera les threads selon la documentation. On ajoutera également une condition pour vérifier que si l'index sort de la grille de pixel, celui-ci ne puisse pas faire de traitement.

Afin d'améliorer la lisibilité du code, on crée une fonction **angle_calc** qui va effectuer le calcul de l'angle avec la fonction arc tangente. Cependant, on doit lui rajouter `__device__` pour pouvoir l'utiliser dans le kernel. Dans ce kernel, toutes les valeurs seront converties en flottants pour éviter les arrondis, qui fausserait grandement le résultat final.



FIGURE 4 – Image obtenue à partir du point (245,497) sur le fichier 1.input.ppm sur GPU

Cette version fait beaucoup de calcul inutiles. En effet, si un des points bloque la visibilité depuis le point courant, il est inutile de faire les calculs sur les derniers points.

Avec cette version, le calcul se termine uniquement lorsqu'on atteint le centre. On peut donc s'arrêter dès qu'un point bloque la visibilité et sortir de la boucle. On préférera une boucle while afin de mieux contrôler les conditions de sortie.

On considérera que cette modification de programme ne fait pas partie de l'optimisation du kernel car cela dépend uniquement de la partie mathématique et non la spécificité du GPU.

2.3 Les différences CPU/GPU

2.3.1 Résultats

Lors de la comparaison entre la version CPU et GPU, on obtient deux images légèrement différentes. Nous supposons que cela est dû aux arrondis qui sont calculés différemment sur CPU et GPU, ainsi que les fonctions qui n'arrondissent pas à la même précision.

2.3.2 Modification de l'algorithme

Afin d'avoir une meilleure comparaison entre les performances du CPU et du GPU, on utilisera l'optimisation décrite dans la version du GPU, consistant à changer la boucle `for` par une boucle `while` et donc de s'arrêter avant d'avoir effectué des calculs inutiles.

2.4 Version GPU optimisé

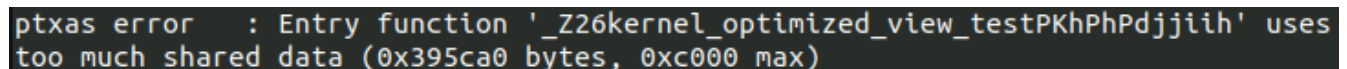
Maintenant que nous avons effectué le traitement sur une version naïve, on va chercher à optimiser l'algorithme à l'aide des différentes techniques vues en cours et en TP.

2.4.1 Mémoire cache

Pour diminuer le nombre de calculs redondants, il est possible d'utiliser la mémoire cache, donc de partager la mémoire entre tous les threads d'un même bloc. La première idée a été de créer un cache qui fait la taille de l'image afin de stocker tous les calculs d'angle de nos points et ainsi de simplement faire un accès de lecture en mémoire, qui est beaucoup plus rapide que de faire le calcul de l'angle à chaque fois. Cependant, d'après les spécifications de la carte la taille du cache est de **48 Ko**.

$$Cache = 48 \times 2^{10} \text{ octet} < 684 \times 687 \times \text{sizeof(float)} = 684 \times 687 \times 4 \sim 1836 \times 2^{10}$$

On obtient ainsi l'erreur suivante :



```
ptxas error : Entry function '_Z26kernel_optimized_view_testPKhPhPdjjiih' uses
too much shared data (0x395ca0 bytes, 0xc000 max)
```

FIGURE 5 – Erreur lors de la création d'un cache de trop grande taille (684×687)

Il serait possible de changer la valeur de cache en modifiant la variable d'environnement qui lui est associé. Cependant, cette méthode n'est pas envisageable pour une taille d'image grande, et présenter des risques si la valeur n'est pas choisie judicieusement.

Néanmoins, on peut se restreindre à un cache plus petit, qui fera la taille du bloc de notre image.

Ce cache sera sous la forme d'un tableau de taille **Nombre bloc** × **Nombre bloc**. On commencera par synchroniser les threads du bloc entre eux avec *sync_threads* lors de l'initialisation du tableau à 0 et une fois que les valeurs seront mises dans le tableau.

Il est nécessaire de synchroniser les threads une fois les valeurs mises dans le tableau, pour éviter qu'un thread accède à une case du tableau qui n'a pas encore été calculé par un autre thread. On utilise la fonction *atomicAdd* pour mettre la valeur dans le tableau et les threads pourront ainsi accéder aux angles des autres points du même bloc. Cela ne représente pas un grand nombre de points car les calculs ne se font que sur le même bloc, mais cela réduit tout de même le nombre de calcul nécessaire.



FIGURE 6 – Image obtenue à partir du point (245,497) sur le fichier 1.input.ppm pour le kernel optimisé

2.4.2 Grid Stride Loop

Cette amélioration n'est valable que pour les images de grande taille. Si le nombre de bloc dépasse la capacité maximum de la carte graphique, il est nécessaire de passer par une grid stride loop . Chaque thread ne va donc pas gérer un seul pixel mais plusieurs. Le thread va effectuer un travail sur un premier pixel puis se déplacer en x et en y de la taille de la grille, travailler sur ce pixel ... jusqu'à sortir de la grille et donc se terminer. Il n'y a pas de différences par rapport à la version non optimisée sur les images, mais on ne remarque pas de changement significatifs. On peut relier ce manque de changement à une image qui n'est pas suffisamment grande pour dépasser la taille limite de bloc même avec l'image *limousin-full.ppm*. Il serait possible de réduire volontairement le nombre de bloc dans la grille pour créer un dépassement. Cependant, cela n'améliorera pas les performances car les threads auront plus de travail à exécuter. Cela peut-être utile pour des images avec une très grande résolution ou bien une carte graphique avec peu de performances.

2.5 Optimisation par analyse

Nous avons tenté d’optimiser ”à la main” notre kernel sans gain de performances, voir même de moins bonnes. Afin d’avoir une analyse pertinente de nos kernel nous nous sommes tournés vers l’outil *NVIDIA Nsight Compute*. Cet outil est pratique dans le sens où on aura un retour explicite des problèmes ralentissant nos kernel. Les analyses seront faites sur 1.input.ppm. L’outil nous fait remarquer un point important, la pipeline FP64 (La suite d’instruction de calcul sur des doubles) est sur-utilisée et est utilisée 86% du temps ce qui va sûrement amener à une saturation de cette dernière et donc à une perte de performance.

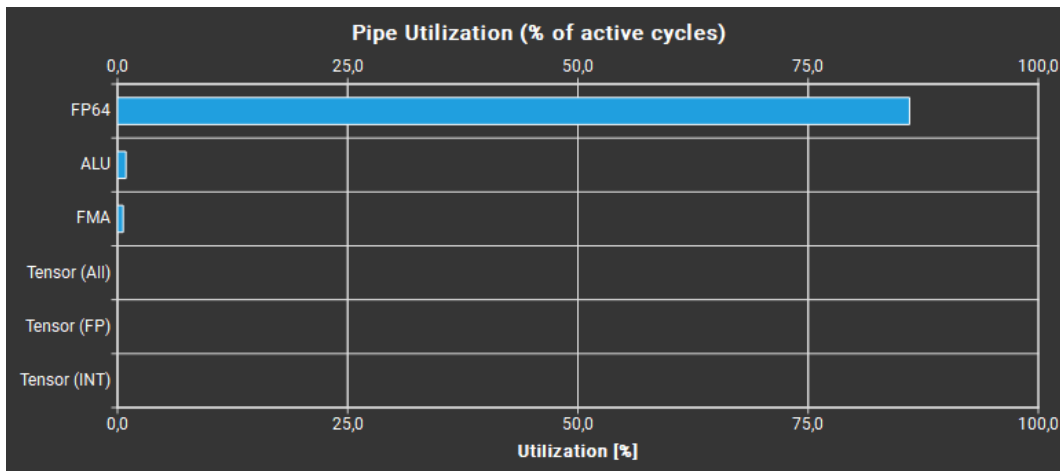


FIGURE 7 – Utilisation des Pipeline sur Nsight avant optimisation

De plus, on apprend que le rapport de vitesse pour les calculs sur des float plutôt que sur des doubles à performances maximales est de 64 :1 sur notre type de GPU. Concrètement notre GPU est 64x moins rapide sur ce type de calcul que si nous utilisions des float.

”The ratio of peak float (fp32) to double (fp64) performance on this device is 64 :1. The kernel achieved close to 0% of this device’s fp32 peak performance and 64% of its fp64 peak performance. If Compute Workload Analysis determines that this kernel is fp64 bound, consider using 32-bit precision floating point operations to improve its performance. See the Kernel Profiling Guide for more details on roofline analysis.”

Dans ce projet, le but n’étant pas d’avoir une image extrêmement précise, nous pouvons nous permettre de passer tout les calculs utilisant des doubles en float. Nsight nous indique un gain de performance d’environ 66% sur ce problème. Nous sommes passés de 39.70ms à 1.14ms en temps d’exécution. Ce qui nous donne un gain de performance de

$$\frac{39.70 - 1.14}{39.70} = 97\%$$

De plus, en analysant ce kernel, Nsight indique que maintenant, la pipeline la plus utilisée est FMA(Fused Multiply-Add), cette pipeline s’occupe des opérations sur les flottants. Certes, la pipeline FMA est la plus utilisée mais le travail est maintenant équilibré, on remarque que l’ALU(Arithmetic Logic Unit), qui va gérer les opérations logiques est aussi très utilisée. Ce qui est cohérent avec notre gain de performance car on manipule maintenant des nombres 32 bits.

2.6 Optimisation complète

Cependant, après une analyse de notre kernel optimisé, Nsight continu d’indiquer un problème que nous n’avons pas traité dans la partie précédente. Nous baserons cette optimisation sur le document Analysis-Driven Optimization: Analyzing and Improving Performance with NVIDIA Nsight Compute, Part 2 de Bob Crovella

”Compute is more heavily utilized than Memory : Look at the Compute Workload Analysis section to see what the compute pipelines are spending their time doing. Also, consider whether any computation is redundant and could be reduced or moved to look-up tables.”

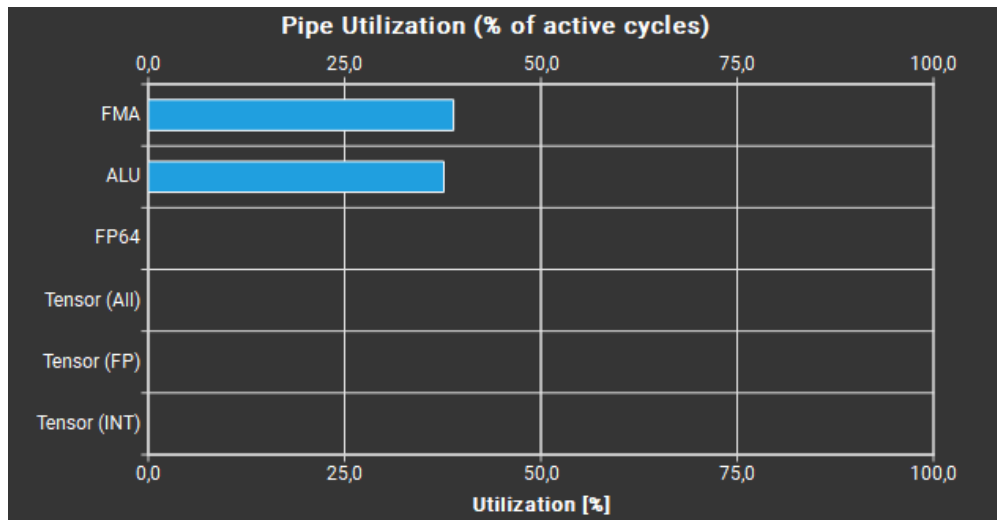


FIGURE 8 – Utilisation des Pipeline sur Nsight après optimisation

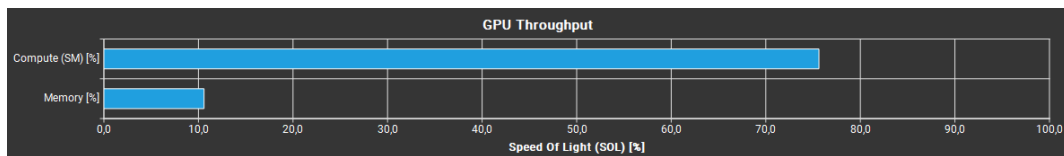


FIGURE 9 – Utilisation de la capacité du GPU pour le kernel optimisé par analyse

On nous dirige donc vers la section "Compute Workload Analysis" en nous suggérant que l'exécution d'opérations étant beaucoup plus élevée que l'utilisation de la mémoire est gênant, mais ici aucun problème n'est indiqué par Nsight. Quelque chose peut-ependant nous interpeller ici, en se penchant sur le second graphe :

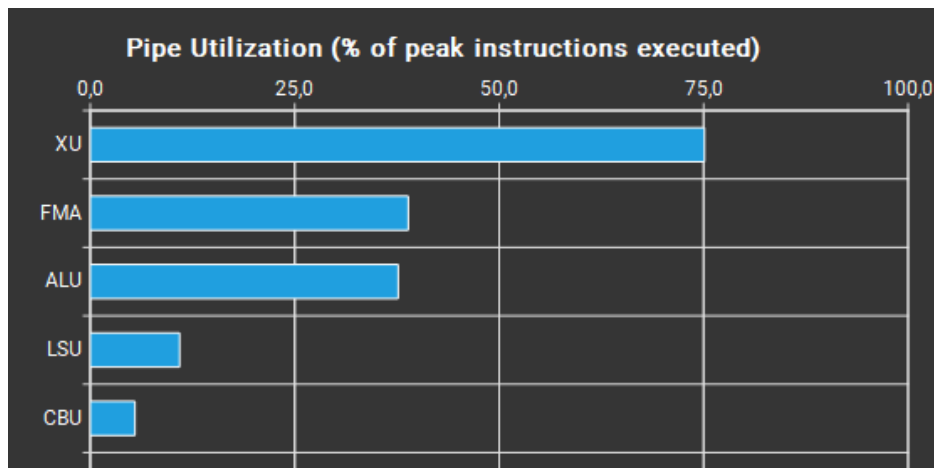


FIGURE 10 – Utilisation de la pipeline LSU pour le kernel optimisé par analyse

On remarque ici que la pipeline LSU(Load-Store Unit) est peu utilisée, ce qui suggère donc peu d'interactions avec la mémoire comme précisé par Nsight. En regardant la section "Memory Workload Analysis" on peut consulter la charte des accès mémoire :

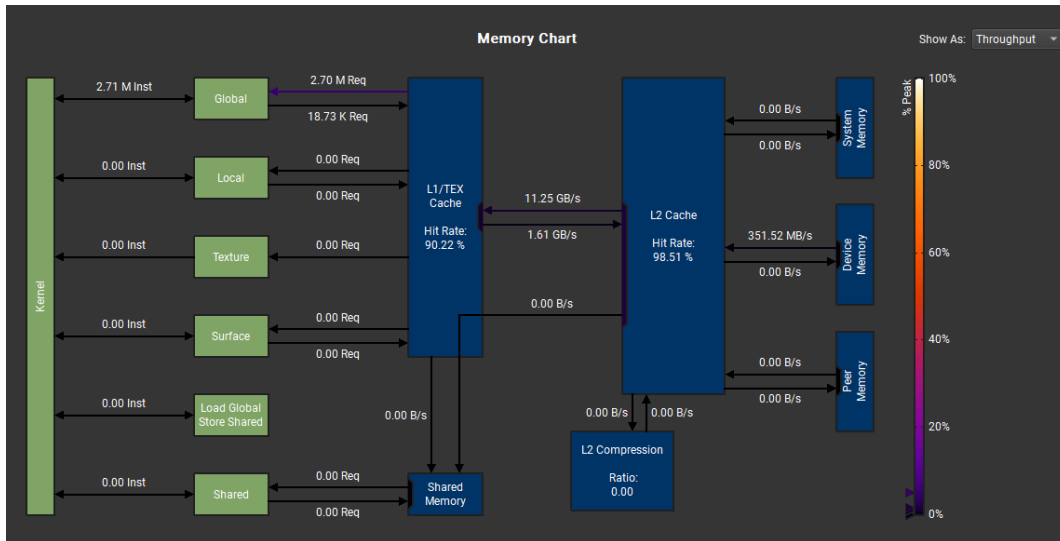


FIGURE 11 – Charte des accès mémoire pour le kernel optimisé par analyse

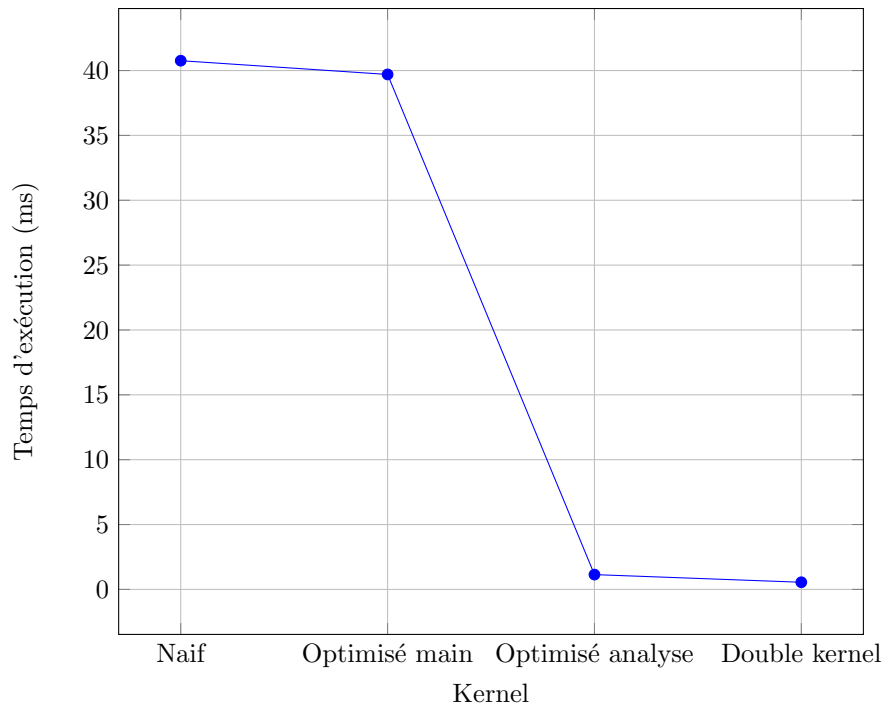
On remarque qu'on utilise absolument pas les capacités de la mémoire de notre carte graphique. On passerait donc à côté d'un gain de performance (non précisé ici). Notre solution a donc été d'utiliser les accès mémoire en divisant le travail sur deux kernel distincts : Le premier (kernel_angle_calc) pour le calcul des angles le second (kernel_view_test_tab) pour déterminer si un point est visible. Avec cette méthode on passe de 1.14 ms à 0.55 ms ce qui nous donne un gain de performance de

$$\frac{1.14 - 0.55}{1.14} = 52\%$$

Après analyse on remarque qu'on utilise beaucoup plus les accès mémoire et la pipeline LSU ce qui est donc cohérent avec notre optimisation théorique. (Voir Annexe) 5.1.

En résumé, voici un graphe des temps d'exécution durant les différentes optimisations :

Temps d'exécution en fonction du kernel



3 Réduction d'image

Dans cette partie, on va chercher à travailler sur une image de plus petite taille pour réduire le nombre de calcul total à faire. Le pixel choisi qui représentera le bloc de pixels sera celui qui aura la hauteur (donc la valeur de pixels) la plus grande.

Durant l'ensemble de l'exercice on se placera dans le cas où on passe de l'image **1.input.ppm** à une image de taille 10×10 . Cependant, le programme fonctionne pour toutes les tailles tant que l'image de sortie est de taille plus petite ou égale à la taille de l'image d'entrée.

3.1 Version séquentielle

Algorithme 2 Réduction

```
1: procédure REDUCTIONIMAGE(Image entree,Entier coupeX,Entier coupeY) :
2:   Pour i allant de 0 à hauteur, faire :
3:     Pour j allant de 0 à hauteur, faire :
4:       indexSortieX  $\leftarrow$  hauteur/coupeX
5:       indexSortieY  $\leftarrow$  largeur/coupeY
6:       valeur_pixel  $\leftarrow$  sortie[indexSortieX + largeurSortie  $\times$  indexSortieY]
7:       Si valeur_pixel < entree[i + largeur  $\times$  j] alors
8:         sortie[indexSortieX + largeurSortie  $\times$  indexSortieY]  $\leftarrow$  entree[i + largeur  $\times$  j]
9:       Fin si
10:    Fin pour
11:  Fin pour
12:  renvoyer sortie
13: Fin procédure
```

3.2 Version GPU non optimisé

Pour cette première version, on veut avoir la valeur maximale parmi les différents pixels. Or, il est impossible de modifier plusieurs fois la même valeur (en l'occurrence d'un élément d'un tableau) en parallèle sans créer de problèmes d'accès concurrents. Pour cela, il faut utiliser la fonction **atomicMax**. Cependant il est impossible d'utiliser cette fonction sur des *unsigned char*. Après avoir essayé de redéfinir la fonction **atomicMax** sur des *unsigned char* sans succès, il a été nécessaire de changer de méthode.

Plutôt que de passer pas des *unsigned char*, on utilise un tableau d'entier non signé classique sur 32 bits, que l'on utilisera pour le calcul et une fois cette étape terminée, on réaffectera notre tableau temporaire à notre tableau de *char*.

3.3 Version GPU optimisé

Dans cette version, on va tenter de réduire le nombre d'instructions bloquantes et d'utiliser les notions vu en cours.

3.3.1 Utilisation de la mémoire partagée

Ici, il ne semble pas pertinent d'utiliser la mémoire partagée, car l'objectif est d'obtenir une sous tableau de valeur maximale d'un plus grand tableau. Cependant, tous les threads ne travaillent pas forcément sur la même case du tableau donc on ne peut pas utiliser le maximum de la même manière qu'en TD. On pourrait modifier le nombre de threads par blocs de sorte à ce que chaque bloc correspondent à un pixel de la sous-image mais cela ne fonctionnerais que pour des cas très spécifiques car le nombre de threads par bloc doit être une puissance de 2.

3.3.2 Utilisation de mémoire constante

Une autre possibilité est d'utiliser la mémoire constante, plus petite que la mémoire partagée mais qui est visible pour tous les blocs et pourrait donc stocker l'image de sortie. Cependant cette mémoire constante est accessible uniquement en lecture et non en écriture, ce qui rend son utilisation impossible.

3.3.3 Optimisation par analyse

Maintenant que nous avons un kernel fonctionnel, nous allons lancer une nouvelle analyse pour voir si il est possible de l'optimiser.

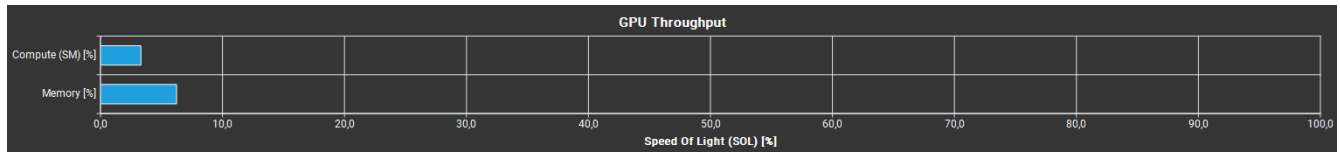


FIGURE 12 – Utilisation de la capacité du GPU pour le kernel réduction d'image naïf

On voit que les capacités du GPU que cela soit en mémoire ou en puissance de calcul sont sous utilisées. On peut confirmer notre hypothèse avec l'utilisation des pipeline qui est extrêmement faible.

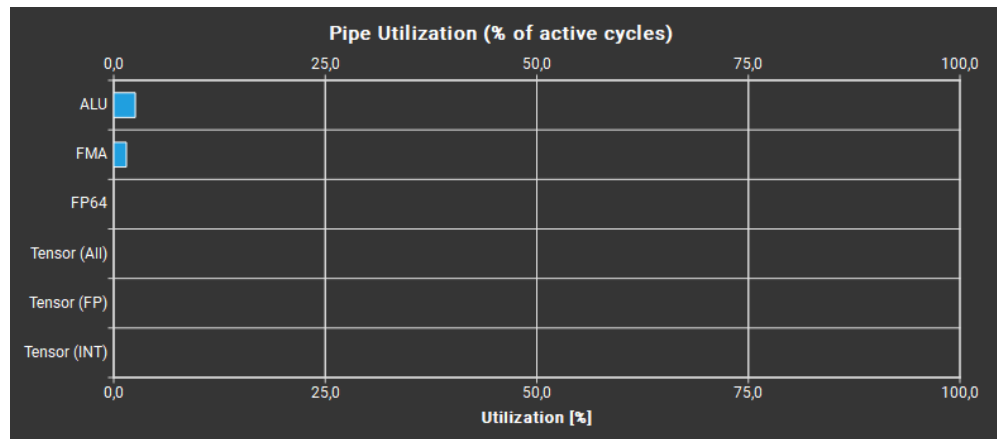


FIGURE 13 – Utilisation des Pipeline pour le kernel réduction d'image naïf

On constate cette faible utilisation par rapport à toute l'exécution du kernel à cause d'une fonction sur laquelle on repose entièrement.

```
1 if (xi < outWidth && yj < outHeight){
2     uint32_t pixValue = dev_inPtr[indexIn];
3     atomicMax(&dev_outPtr[indexOut], pixValue);
4 }
```

Code 1 – Les trois dernières lignes du kernel naïf tiled

Cette fonction est *atomicMax(valeur1, valeur2)*. En effet, on utilise cette fonction pour éviter l'accès concurrent de plusieurs threads à une valeur partagée. De ce fait, cette opération atomique est dite "bloquante" c'est à dire que les threads devront attendre leur tour avant de pouvoir mettre à jour la variable partagée qui est ici le pixel stocké dans le tableau. Pour pouvoir avoir de meilleures performances, nous allons faire en sorte de réduire l'impact de l'opération atomique sur les performances du kernel. Pour se faire, nous allons éviter les appels redondants. Voici notre proposition :

```
1 if (xi < outWidth && yj < outHeight){
2     uint32_t pixValue = dev_inPtr[indexIn];
3     if(pixValue > dev_outPtr[indexOut]){
4         atomicMax(&dev_outPtr[indexOut], pixValue);
5     }
6 }
```

Code 2 – kernel naïf tiled avec impact d'atomicMax réduit

Ici le second *if* à la ligne 4 permet de diminuer le nombre de fois que la fonction *atomicMax* est utilisé. En effet, comme les threads s'exécutent en parallèle, la valeur de l'élément du tableau de sortie va augmenter au fur et à mesure. Cependant, si un thread a mis une valeur assez grande avant que d'autres threads n'est pu évalué le *if*, on ne passe pas dans la condition et cela ne change pas la valeur finale, car cela ne pourra jamais être la valeur maximum.

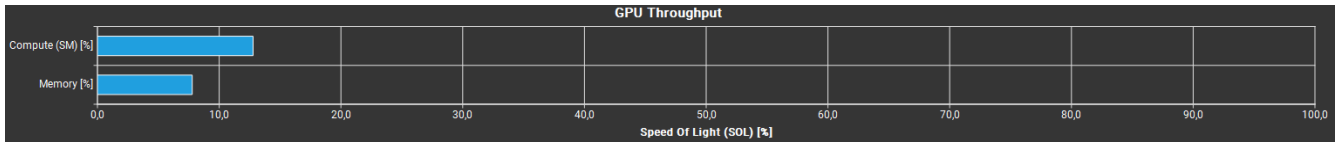


FIGURE 14 – Utilisation de la capacité du GPU pour le kernel réduction d'image optimisé

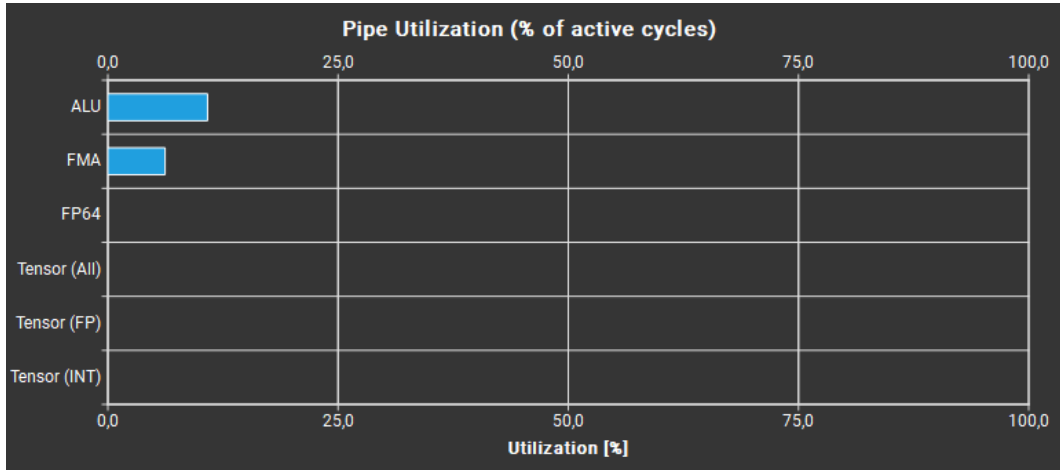


FIGURE 15 – Utilisation des Pipeline pour le kernel réduction d'image optimisé

Après analyse, on note une amélioration sur l'utilisation faite de la puissance de calcul du GPU, cependant, les valeurs restent faible. Regardons tout de même le gain de temps d'exécution. Nous sommes passés de 0.15ms à 0.35ms en temps d'exécution. Ce qui nous donne un gain de performance de

$$\frac{0.35 - 0.15}{0.35} = 57\%$$

Le but final aurait été ici de s'affranchir complètement de l'utilisation d'*atomicMax* mais nous n'avons pas trouvé de moyen de réaliser cette optimisation.

4 Conclusion

Lors de la réalisation de ce projet nous avons pu appréhender le problème de la parallélisation appliqué à une image et des difficultés qui y sont liés. Nous avons pu expérimenter différentes stratégies et optimisations pour le résoudre. De plus, nous avons utilisé l'outil d'analyse d'NVIDIA : Nsight Compute. C'est à l'aide de cet outil que nous avons pu réaliser les optimisations sur nos kernel. En manipulant le logiciel nous en avons appris plus sur la manière dont fonctionne un GPU et CUDA dans son ensemble. En effet, les notions de pipeline et d'accès mémoire ce sont précisées. Mais encore, nous avons pu constater l'importance du choix d'un processeur graphique en fonction du traitement qu'il sera amenée à faire (FP32, FP64). Dans l'ensemble, ce projet nous aura permis d'acquérir de meilleures compétences en programmation GPGPU que cela soit dans le domaine de l'analyse de kernel, la parallélisation de traitement ou encore dans la programmation avec CUDA en elle même.

5 Annexes

5.1 Analyse du double kernel avec Nsight

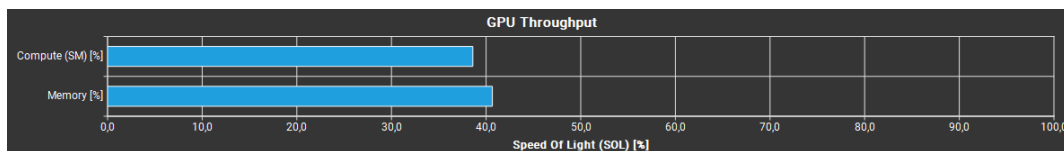


FIGURE 16 – Utilisation de la capacité du GPU pour kernel_angle_calc

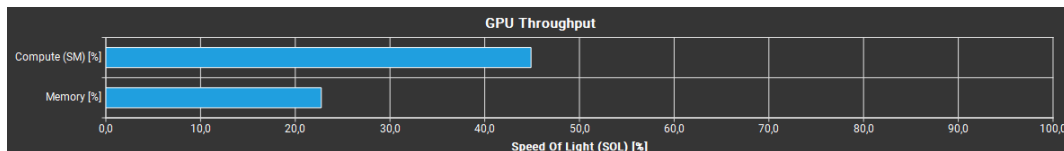


FIGURE 17 – Utilisation de la capacité du GPU pour kernel_view_test_tab

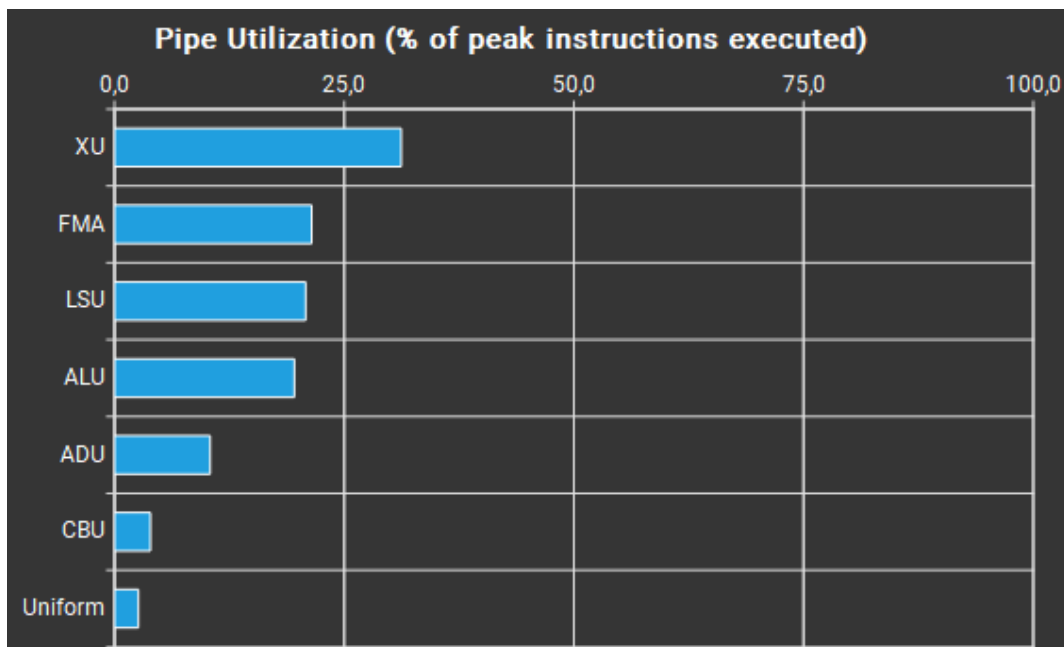


FIGURE 18 – Utilisation de la pipeline LSU pour kernel_angle_calc

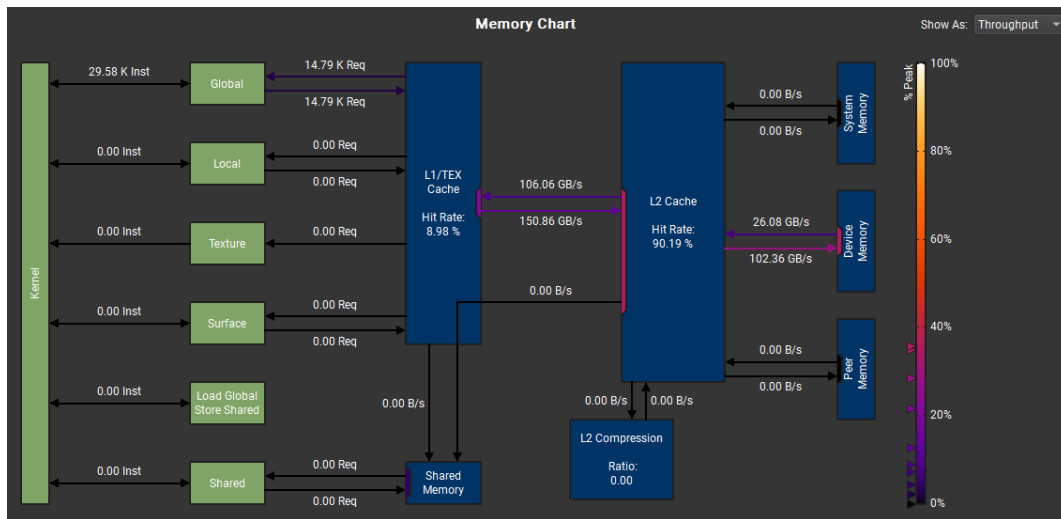


FIGURE 19 – Charte des accès mémoire pour kernel_angle_calc

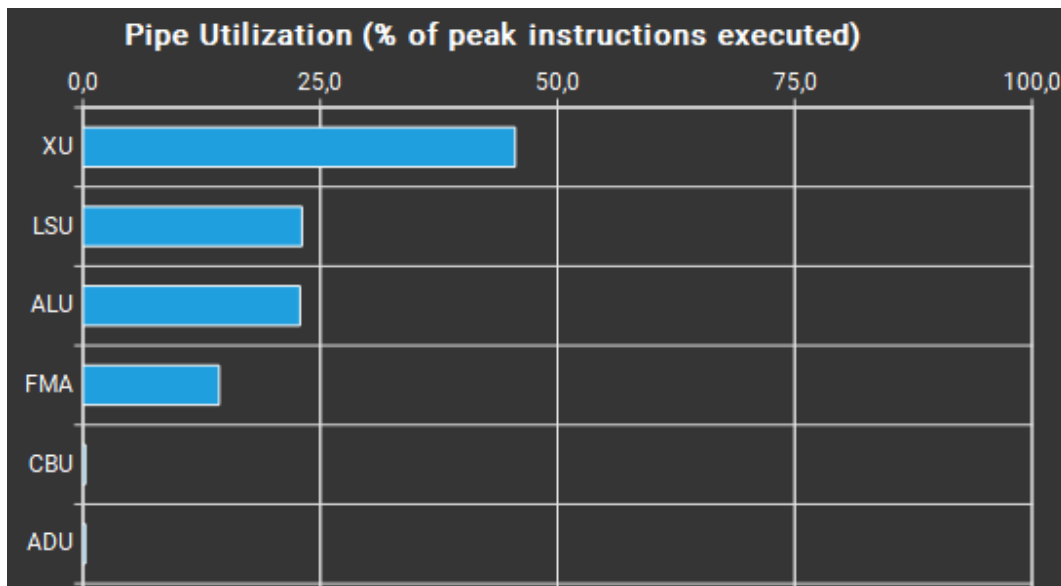


FIGURE 20 – Utilisation de la pipeline LSU pour kernel_view_test_tab

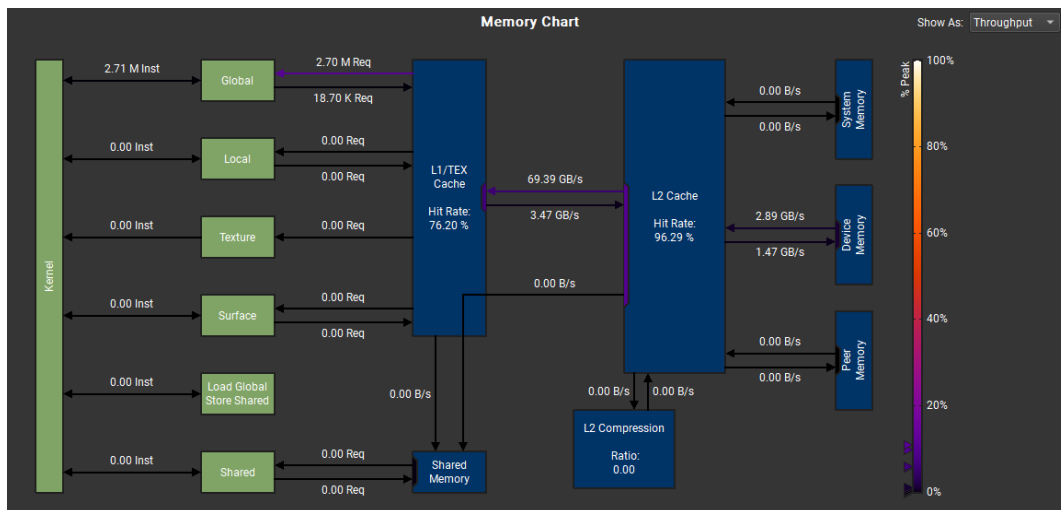


FIGURE 21 – Charte des accès mémoire pour kernel.view_test_tab