Raytracing

CHPS0802

CLEMENT JOURD'HEUIL

ELIOT CALDERON Y MORA

SUPERVISE PAR M. ALIN FRANÇOIS

Table of Contents

Introduction

Dans le cadre du projet de la matière CPHS0802, programmation GPU. Le projet qui nous a été confié est la réalisation d'un algorithme de Raytracing, optimisé pour une utilisation sur GPU Nvidia avec comme langage de support Cuda. Ce modèle de rendu utilise une version simplifiée de la gestion de la lumière avec un modèle de Phong. Nous pouvons ainsi rendre des objets 3D dans une scène.

L'objectif de ce projet est de générer une image d'une scène, mais également de rendre une vidéo avec des objets en mouvement avec un nombre d'images par seconde défini manuellement.

Nous allons dans la suite de ce rapport présenter le fonctionnement du lancer de rayons, ensuite la structure de notre programme avec les objets et les fonctions utilisées. Pour nous assurer du bon fonctionnement des opérations, des tests unitaires ont été mis en place. Après cela, nous présenterons notre implémentation parallélisée pour GPU avec des résultats. Nous finirons par une conclusion.

Raytracing

Le Raytracing est une technique de calcul permettant le rendu d'une scène virtuelle depuis la vision d'une caméra elle aussi virtuelle placée dans cette scène. Le procédé simplifié consiste en l'envoi de rayons depuis la caméra, lors d'une intersection entre un rayon et le plan d'un objet, le pixel de l'image qui correspond à un rayon prend la couleur de l'objet.

L'intersection entre un rayon et un triangle est calculée via l'algorithme de Möller-Trumbore. Il permet de détecter efficacement si un rayon coupe un triangle, et à quelle distance. Une fois l'intersection trouvée, on calcul la couleur du pixel avec un modèle de Phong.

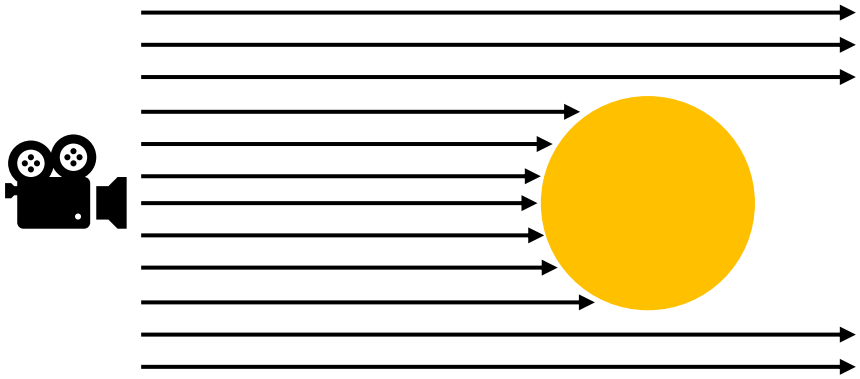


Figure 1 - Schéma lancé de rayons depuis une caméra

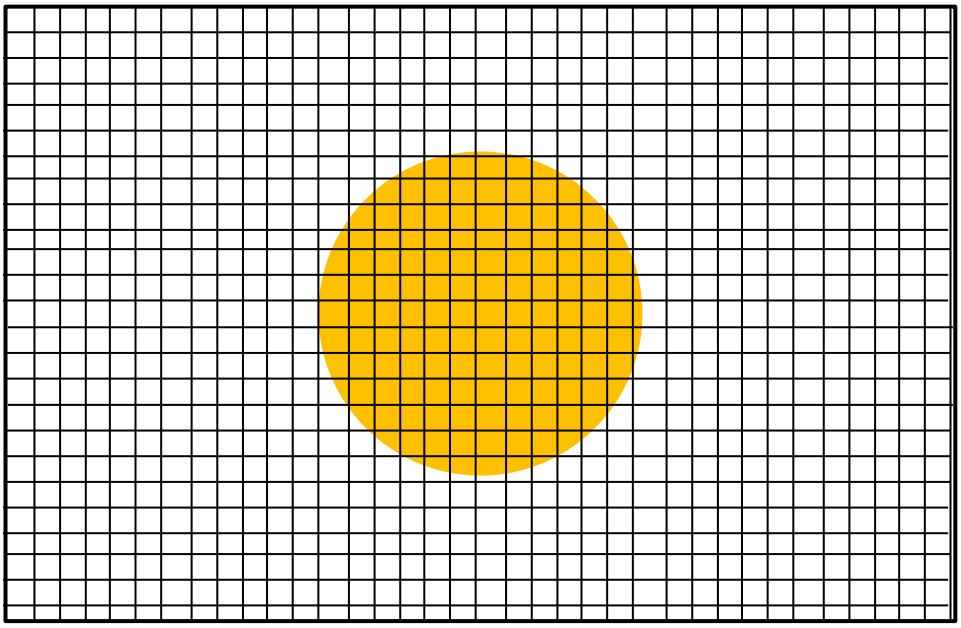


Figure 2 - Représentation de la vision de la caméra après le lancer de rayons

Structure du programme

La scène

Une scène est composée d'un vecteur de formes qui seront rendues, pour ce faire il faut une caméra placée et orientée dans la direction voulue et de plus un vecteur de lumières permettant ainsi de faire le rendu de la scène avec une ou plusieurs lumières.

Les formes

Notre classe principale pour gérer les objets qui seront rendus dans la scène est la classe abstraite « Form », elle est héritée par toutes les autres classes qui seront des objets rendus comme une surface ou un volume. Cette classe contient donc deux fonctions permettant de faire les calculs de rendu : « intersection » qui retourne un booléen nous permettant de savoir si un rayon touche une surface et « getNormal » qui nous retourne la normale d'un plan.

Sphère

Cet objet est créé mathématiquement dans la scène, il n'y a pas de maillage, seule les formules de la sphère permettent de déterminer si une intersection est effectuée avec un rayon. Une autre version avec des triangles a été créer pour le rendu sur GPU.

Triangle

C'est la forme principale pour créer tout type d'objets, le triangle est un plan en trois points qui peuvent être assemblés pour former des objets plus complexes.

Cube

Le seul volume disponible. Il est composé de 12 triangles rectangles isocèles. Pour faire le rendu d'un cube, il a été décidé d'insérer dans le vecteur de formes de la scène, tous les triangles du cube, ainsi on ne fait pas le rendu d'un cube mais plutôt de 12 triangles.

La lumière

Pour faire l'éclairage de notre scène, nous utilisons un algorithme de Phong. Celui-ci nécessite un matériau qui contient les paramètres pour les 3 types de lumières, soit l'ambiante, la diffuse et la spéculaire.

Test unitaires

Des tests unitaires avec la bibliothèque **Google Test** ont été effectués afin de s'assurer du bon fonctionnement des différentes fonctions de calculs. Nos tests permettent de vérifier que les opérations réalisées sur des vecteurs 3D sont corrects comme le produit scalaire (dot), le produit vectoriel (cross), tester les différents types d'opérateurs surchargés. Des tests similaires sont effectués sur les fonctions de notre classe « Point3D » et pour finir nous vérifions si la fonction qui lance un rayon se dirige dans la direction souhaitée.

Ces tests sont effectués sur nos classes qui sont utilisées par la version séquentielle du programme de Raytracing, mais également sur les structures utilisées pour le programme parallèle sur GPU.

Parallélisation

Cet algorithme de Raytracing a été optimisé pour une utilisation GPU, sur des cartes graphiques Nvidia, l'outil qui nous servira pour cela est le langage de programmation Cuda. Nous avons décidé que la partie à paralléliser serait le rendu des objets de notre scène.

Gestion mémoire

Pour commencer nous déclarons et initialisons sur l'host la scène avec caméra, lumières et les objets à rendre qui sont ajoutés à la scène. Pour pouvoir être utilisé sur notre GPU, nous effectuons un transfert de mémoire avec d'abord une allocation mémoire « cudaMalloc » sur le device puis on copie de la mémoire host vers la mémoire device avec un « cudaMemCopy ». À la fin nous copions l'image du device au host.

Kernel du rendu

Le rendu de la scène est fait via le kernel, chaque thread se charge d'un groupe de pixels. On commence par envoyer un rayon depuis la caméra qui se charge d'un pixel (x, y) de l'image. On cherche dans notre tableau d'objets l'intersection avec le plus proche, si une intersection est trouvée, le pixel prend la couleur de l'objet.

Résultats

Tous les tests ont été effectués dans une scène de 1920 x 1080, cette scène est composée d'un cube effectuant une rotation et de 2 sources de lumière. Les vidéos ont une durées de 2 sec et les résultats obtenus sont faits pour différentes fréquences d'images de vidéo souhaitées, soit 30, 60, 120, 144, 165 et 240 fps. On retrouve en abscisse le nombre de Fps souhaité pour la simulation et en ordonnée, le temps de génération en seconde ou le nombre de Fps de la génération.

Par exemple pour lire le premier graphique, nous avons lancé un rendu d'un cube qui tourne pour au finale avoir une vidéo en 30 Fps, en séquentiel le programme à mis 25 sec au total pour générer toutes les images et les enregistrer, dans ce temps on compte aussi les échanges mémoires, cette description correspond au premier point de la courbe.

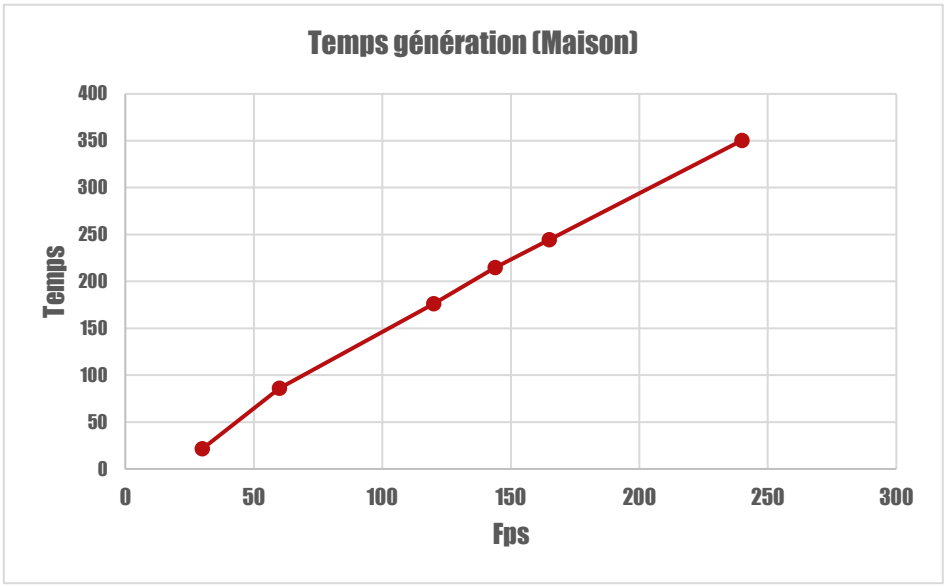
À présent pour lire le deuxième graphique, si on prend le premier point de la courbe, nous voulions faire une vidéo en 30 Fps et le programme a généré en temps réel 1,395 images par seconde.

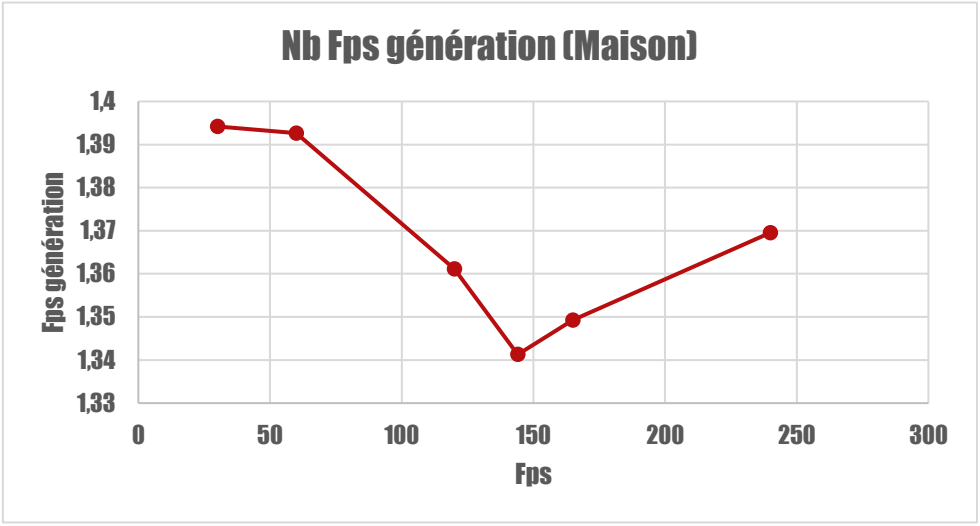
Pour le temps de calcul en parallèle nous avons compté le temps d'exécution global total mais aussi le temps d'exécution dans le kernel, donc la sauvegarde des images n'est pas prise en compte et les échanges mémoires ne le sont pas non plus.

Les machines utilisés pour obtenir ces résultats, sont pour la partie séquentielle, une machine personnelle et Romeo, on remarque que les temps d'exécutions sont très similaires. Pour la parallélisation nous avons fait les essais sur cette même machine personnelle qui comporte une RTX 4050 portable et Juliet.

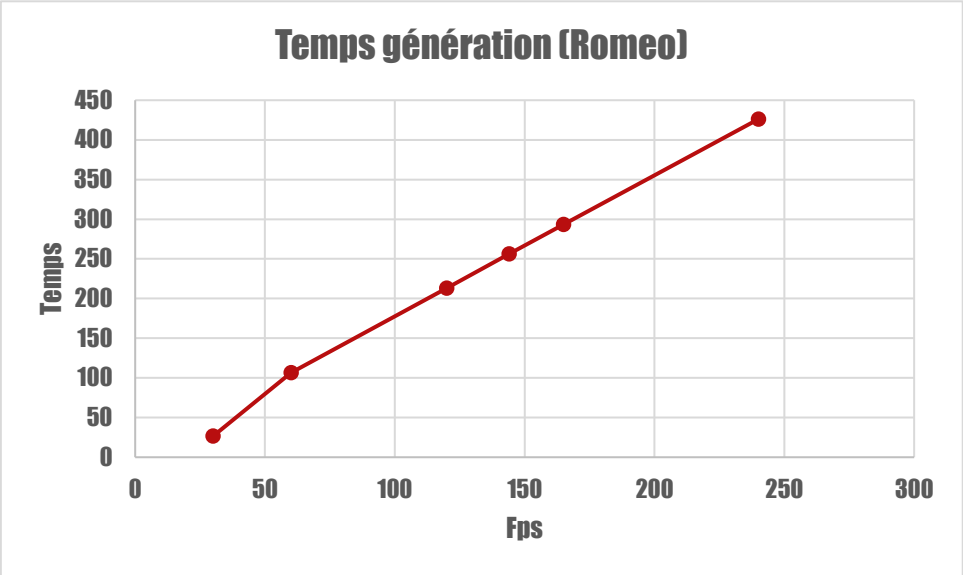
Séquentiel (CPU)

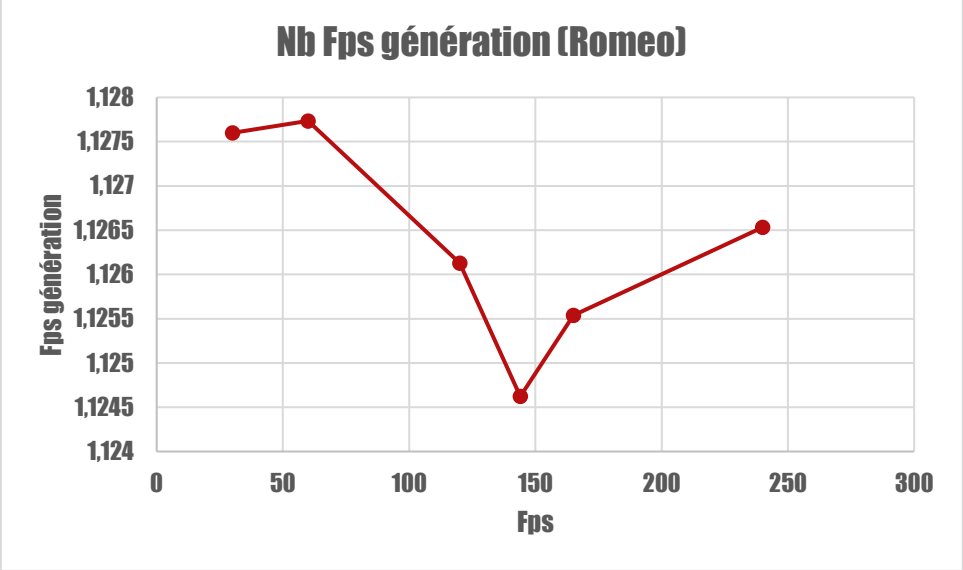
Machine personnelle





Romeo

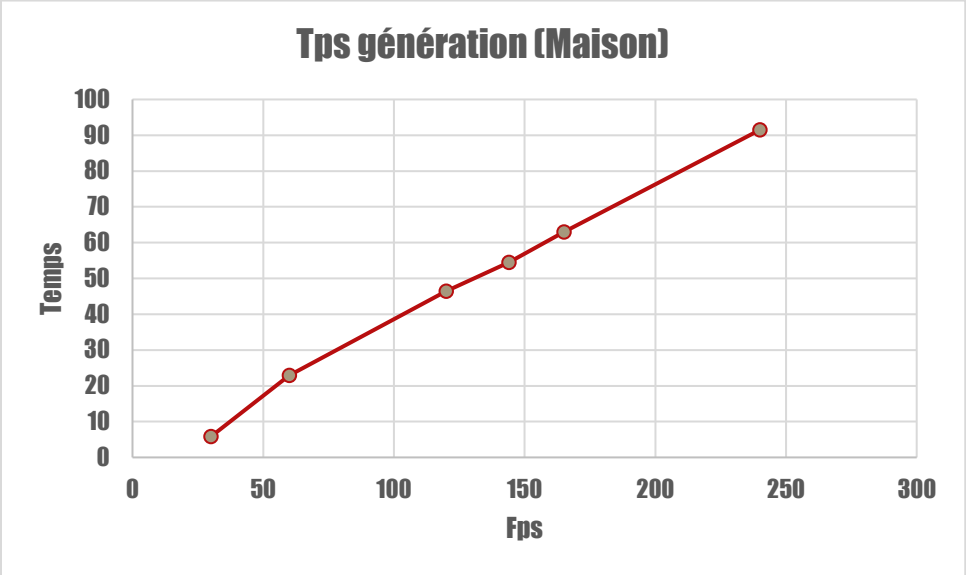

Temps génération (Romeo)
Temps
450
400
350
300
250
200
150
100
50
0
0
50
100
150
200
250
300
Fps

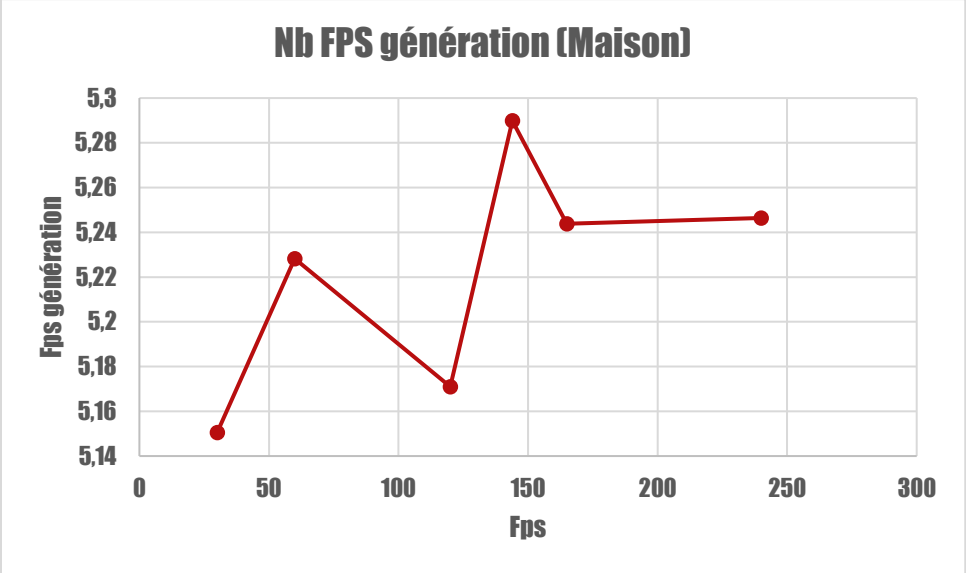

Nb Fps génération (Romeo)
Fps génération
1,128
1,1275
1,127
1,1265
1,126
1,1255
1,125
1,1245
1,124
0
50
100
150
200
250
300
Fps

Parallèle (GPU)

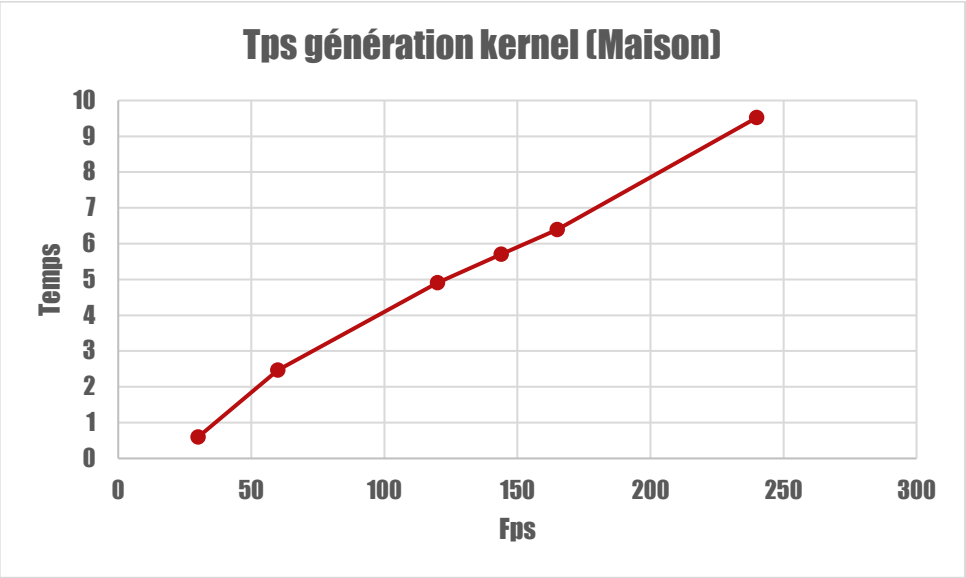
Machine personnelle

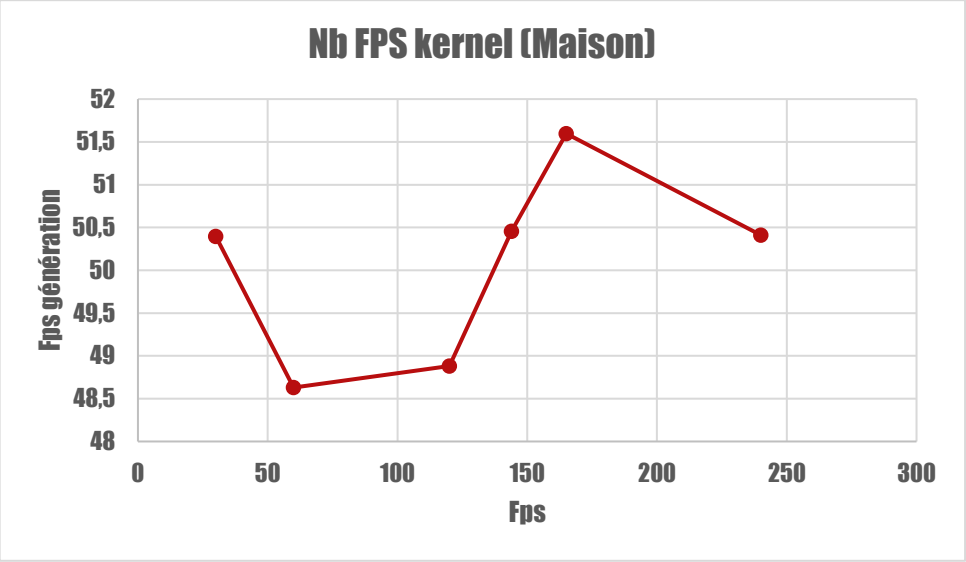
Temps Total





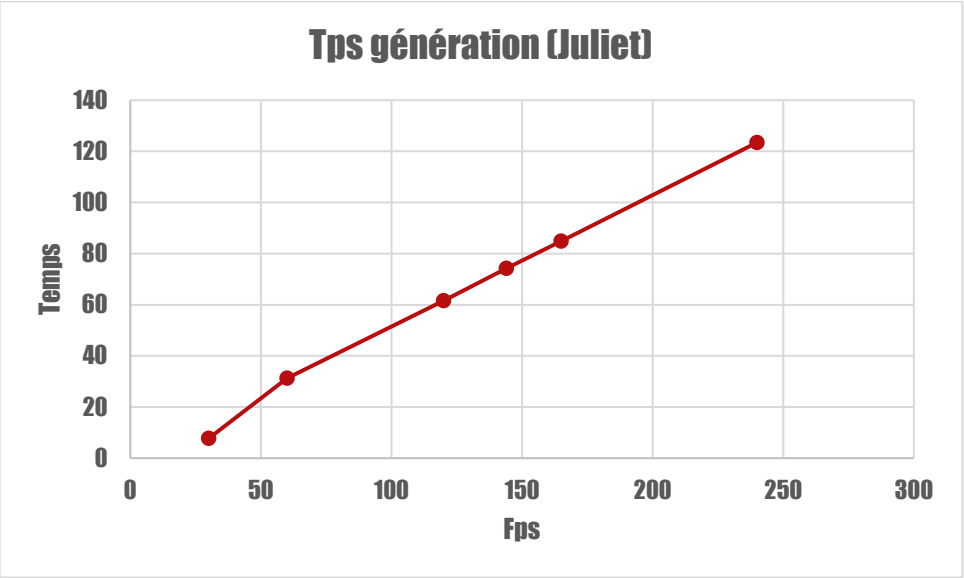
Temps kernel

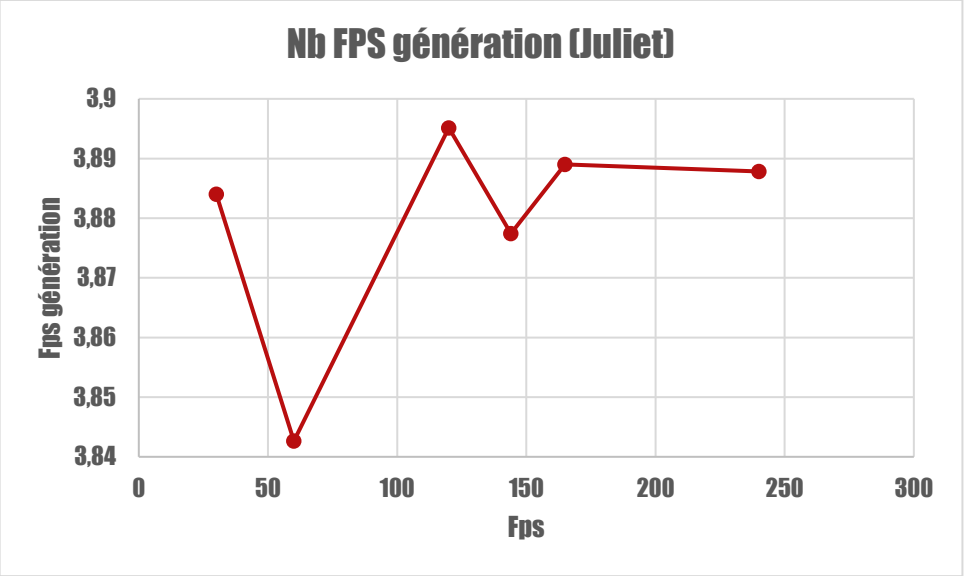




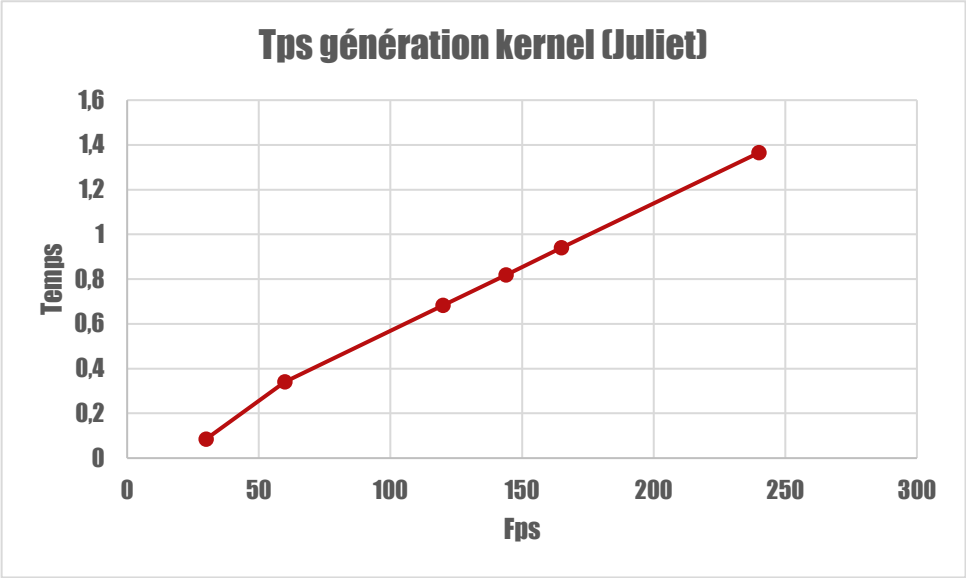
Juliet

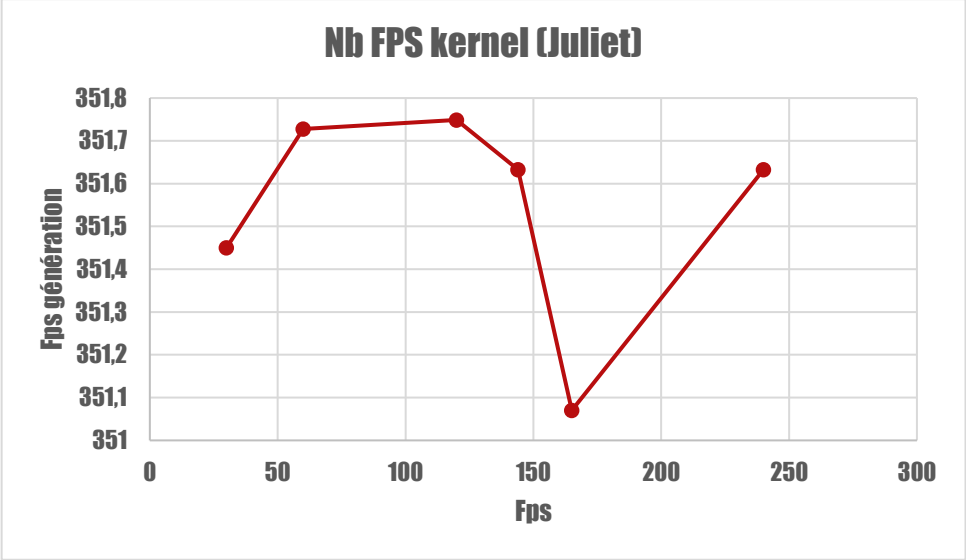
Temps total





Temps kernel





On remarque que sur l'ensemble des graphiques la génération des Fps n'est parfois pas consistante pour un nombre de Fps souhaité, mais cela reste négligeable par rapport à l'échelle des valeurs (ci-dessus par exemple, on passe de 351,6 fps de génération pour une vidéo souhaitée à 144f fps à 351,1 fps de génération pour une vidéo souhaitée en 165 fps).

Conclusion

En conclusion, la parallélisation du Raytracing s'est fait via un kernel sur le rendu un pixel par thread de la scène. On remarque une accélération de la génération néanmoins les échanges mémoires entre le GPU et le CPU (device, host), ainsi que l'écriture des images ppm nécessaire à la génération de la vidéo ralentit l'exécution du Raytracing sur GPU. Cela se remarque par le temps passé dans le kernel comparé au temps de l'algorithme seul.

De plus, après avoir observé le comportement du GPU pendant l'exécution du programme, on remarque que ce dernier n'est pas utilisé à 100% (20-40% du GPU seulement sont utilisés), ce qui implique que la parallélisation pourrait être meilleure en utilisant toutes les capacités du GPU. Nous avons expérimenté différentes tailles de grilles de threads, les performances n'ont pas été significativement impactées.

Référence

Repository GitHub : https://github.com/ElCald/Raytracing_Cuda