CODE DE SIMULATION DE FORAGE (GLCU) : Analyse,Profilage,Perspectives

Olfa Haggui

▶ To cite this version:

Olfa Haggui. CODE DE SIMULATION DE FORAGE (GLCU) : Analyse,Profilage,Perspectives. [Rapport de recherche] E-425.pdf, MINES ParisTech - PSL Research University; Centre de recherche en informatique - MINES ParisTech - PSL Research University. 2017. hal-01691629

HAL Id: hal-01691629
https://minesparis-psl.hal.science/hal-01691629

Submitted on 24 Jan 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

CODE DE SIMULATION DE FORAGE (GLCU)

Analyse - Profilage - Perspectives

Elaboré par : Olfa HAGGUI
Encadrée par : Claude TADONKI (CRI) et Olivier STAB (GEOSCIENCES)

Projet Carnot M.I.N.E.S

PACA

CRI – GEOSCIENCES

2017

Table de matières

Introduction 1

Analyse de GLCU 2

1. Présentation de GLCU 3
2. Environnement logiciel 4

    2.1 Environnement logiciel des architectures GPU(CUDA) 5

    2.2 L'environnement de développement graphique OpenGL 5

        2.2.1 Principe du pipeline 6

        2.2.2 Open GL Buffer 6

3. Analyse des résultats 8

Profilage total de GLCU 9

Profilage détaillé de GLCU 11

1. Analyse et profilage de calcul CUDA 12

    1.1 Calculateur d'occupation 14

    1.2 L'utilisation de la bande passante 16

    1.3 Optimisation d'accès mémoire 17

2 Analyse et profilage de calcul Full Open GL 17

3 Transfert de données (Open GL to CUDA) 21

Conclusion et perspectives 22

Introduction

Les applications de traitement d'image sont généralement soumises à des contraintes temporelles lors de la génération des résultats finaux. Ces applications sont qualifiées de temps réel. On les trouve maintenant aussi bien dans les produits grand public (traitement de signal téléphonie, pilotage automatique des voitures, appareil photo, équipements wifi et audio…) ainsi que dans les équipements industriels lourds (système de surveillance, système aérospatial, supervision médicale …). Le traitement de ces applications est de plus en plus complexe, ce qui implique une augmentation du temps d'exécution. De ce fait, le respect des contraintes de temps d'exécution demeure une tâche importante pour le système informatique. Dans ce contexte, les architectures GPU (Graphics Processing Unit) ont été proposées dont l'objectif d'allouer l'exécution des applications de traitement des images et du signal à un composant à part entière. Elles sont caractérisées par un nombre élevé d'unités de traitement et d'une hiérarchie de mémoire configurable. Ces caractéristiques permettent d'implémenter des applications avec un niveau de parallélisme important, résultant ainsi d'une diminution considérable du temps d'exécution. Dans ce contexte, les applications industrielles explorent la possibilité d'utiliser les cartes graphiques (GPU) pour augmenter considérablement les vitesses de calcul pour atteindre une meilleure performance.

Notre travail s'articule autour d'une bibliothèque de forage développent par le centre de Géosciences de l'école des mines de MINES-ParisTech. Les concepteurs (Stab et Kostas) ont développés une librairie (GLCU) permettant d'extraire d'une scène 3D les distances points à points entre 2 surfaces irrégulières et d'appliquer des calculs sur ces distances pour évaluer l'interaction des surfaces.

L'objectif principal de ce travail est l'évaluation et l'amélioration de la bibliothèque GLCU au niveau du temps d'exécution pour les différentes versions de GLCU. En fait, différentes améliorations ont été déjà effectuées dans des travaux existants. De ce fait, ce travail focalise essentiellement sur une analyse et un profilage détaillé de code de GLCU.

Pour exposer ce travail, ce manuscrit est articulé généralement autour de trois parties. La première partie présente une analyse de l'environnement logiciel et matériel de l'architecture GPU de NVIDIA et de l'Open GL. La deuxième partie présente une analyse et un profilage total de code. La troisième partie présente un profilage détaille de code. Finalement, on clôture par une conclusion qui rappelle l'objectif, les différents apports de ce travail et les perspectives envisagées.

I. Analyse de GLCU

1. Présentation de GLCU

GLCU est une librairie accélérée sur GPU de calcul de l'interaction entre deux surfaces mobiles. En fait, lors de la simulation de forage l'empreinte laissée par l'outil permet d'étudier la surface du milieu foré, d'ajuster les paramètres de coupes et de valider le processus d'usinage ou de forage. GLCU se base sur une interface de programmation graphique permettant de faire des rendus 3D, OpenGL, ainsi que sur une interface de calculs sur carte graphique, CUDA. GLCU est décomposé en deux parties, la première partie utilise l'OpenGL pour calculer les matrices de distances (zbuffer) entre les surfaces animées d'une scène 3D à l'aide d'une caméra et la deuxième partie applique des calculs prédéfini aux zbuffer afin de calculer l'empreinte à l'aide des unités de calcul (des kernels) CUDA [1,2,3].

L'objectif principale de la bibliothèque CLCU est de :

- Extraire d'une scène 3D les distances points à points entre 2 surfaces irrégulières
- le calcul d'empreinte pour le forage
- Augmenter considérablement les vitesses de calcul du trou foré

2. Environnement logiciel

2.1 Environnement logiciel des architectures GPU(CUDA)

CUDA (Compute Unified Device Architecture) est une plateforme logicielle proposée par NVIDIA, offrant un modèle de programmation permettant l'ordonnancement parallèle des GPUs de NVIDIA. La plateforme CUDA permet aux développeurs d'utiliser C comme un langage de programmation de haut niveau. Elle assure l'accès à différentes mémoires de la hiérarchie de la carte ainsi que la synchronisation des traitements affectés à chaque SP.

- **Kernel** est une fonction écrite en langage C, exécutable par plusieurs threads en parallèle sur le GPU.
- **Les threads** sont répartis en block et les blocks en grilles et ils disposent tous d'identifiants uniques. L'appel à un kernel se fait en spécifiant le nombre de threads, de blocks et de grilles sur lequel il sera exécuté.
- **Warps**

Pour manipuler les threads, le multiprocesseur fait recours à l'architecture (single instruction multiple transaction).SIMT. C'est ainsi l'unité SIMT d'un multiprocesseur qui créée, planifie et exécute les threads par groupes de 32 threads, appelés warps. Les threads

d’un warp assurent l’exécution en parallèle du même ensemble d'instructions du kernel. Ces threads sont exécutés d'une façon indépendante [7].

Figure 1: Organisation globale de l’architecture Maxwell (GeForce GTX980)

- **Bloc / Grille**

Un bloc est un ensemble de threads pouvant être exécutés simultanément ou en série sans ordre particulier, il peut contenir de 64 jusqu'à 1024 threads. Une grille est un ensemble de blocs parallèles organisés comme illustré dans la figure 2. Le nombre de blocs de thread dans une grille est généralement dicté par la taille des données en cours de traitement et par le nombre de processeurs dans le système.

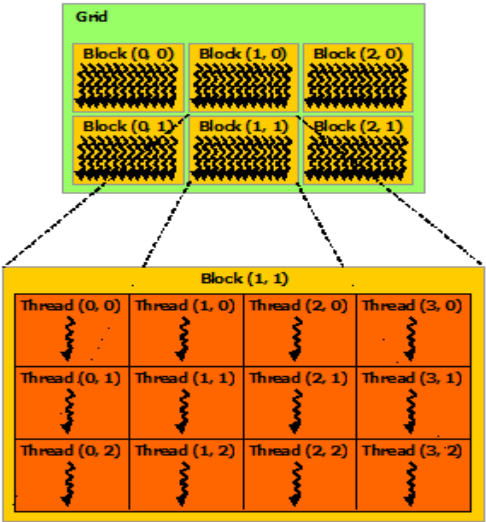

Figure 2 : Illustration d’une grille de blocs de threads (NVIDIA)

Chaque bloc au sein de la grille peut être identifié par un indice accessible à l'intérieur du kernel par la variable prédéfinie blockIdx. La dimension du bloc de thread est accessible dans le kernel par la variable prédéfinie blockDim. Les blocs s'exécutent de façon indépendante sans ordre particulier. Le nombre de blocs par grille ne dépend pas du nombre de processeurs disponibles mais des données à manipuler.
CUDA permet de limiter le nombre de blocs et le nombre de threads par bloc lors de l'appel du kernel dans le programme principal. Le nombre de block pouvant être manipulés en simultané par un multiprocesseur encore appelé nombre de blocks actifs par multiprocesseur dépend des ressources disponibles pour ce multiprocesseur. En d'autres termes, le nombre de registres nécessaires par thread, mais aussi la taille de la mémoire partagée utilisée par block, le tout pour un kernel donné, vont avoir un impact direct sur le nombre de block maximum pouvant être gérés par multiprocesseur.

2.2 L'environnement de développement graphique OpenGL

OpenGL (Open Graphics Library) est une bibliothèque graphique très complète qui permet aux programmeurs de développer des applications 2D, 3D assez facilement. OpenGL pose les spécifications d'une API (Interface de Programmation) pour le dessin 2D et 3D .Elle décrit un ensemble de fonction qui manipule des primitives géométriques (points, vecteurs, polygones) en vue de les projeter sur l'écran tout en considérant l'éclairage, l'orientation, les ombres et les couleurs... Ces fonctions peuvent être implémentées en logiciel (mode compatibilité) ou d'une manière câblé sur des processeurs spécialisés dans le traitement graphique (GPU) [14].

2.2.1 Principe du pipeline

Pipeline Open GL est une succession des opérations généralement réalisées par une carte graphique au rendu d'un lot de données (maillages et textures) sur un tampon (Buffer), la plupart du temps celui-ci est ensuite affiché à l'écran. Ces étapes sont séquencées dans le pipeline OpenGL qui est constitué de quatre principales étapes:

1) Traitement des vertex.
2) Rastérisation.
3) Traitement des fragments.
4) Assemblage des fragments.

Les étapes de traitement des vertex et des fragments sont programmables grâce au langage GLSL (OpenGL Shading Language). C’est un langage qui permet d'écrire des programmes appelés shaders qui sera insérés dans le pipeline de traitement d'OpenGL (sur GPU) à l'exécution du programme principal []. En Fait, il existe deux types de shader: la première qui traitent les vertex et les données qui leurs sont associées (Vertex Shader) et la deuxième traitent les fragments, et les données qui leurs sont associés, générés par la rastérisation des primitives spécifiées au début du pipeline OpenGL (Fragment Shader).

2.2.2 OpenGL Buffer

Open GL est une plateforme qui gère différents type des buffers pour les transferts des données puisque elle manipule des primitives géométriques (points, vecteurs, polygones). Il existe trois types des buffers :

- **FBO**

L'architecture d'objet de tampon de frame (FBO) est une extension d'OpenGL pour faire un rendu flexible hors écran, y compris le rendu à une texture. En capturant des images qui seraient normalement dessinées sur l'écran, elles peuvent être utilisées pour implémenter une grande variété de filtres d'image et des effets de post-traitement. Le FBO est utilisé dans OpenGL pour son efficacité et sa facilité d'utilisation. Un FBO contient une collection de destinations de rendu; couleur, profondeur et stencil buffer. Ces tampons logiques dans un FBO sont appelés images pouvant être attachées par framebuffer, qui sont des tableaux 2D de pixels qui peuvent être attachés à un objet framebuffer. Il existe deux types d'images fixables par framebuffer; **images de texture** et images **renderbuffer**. Si une image d'un objet de texture est attachée à un framebuffer, OpenGL exécute "render to texture". Et si une image d'un objet renderbuffer est attachée à un framebuffer, OpenGL exécute "rendu hors écran". Le diagramme suivant montre la connectivité entre l'objet frame buffer, l'objet texture et l'objet render buffer.

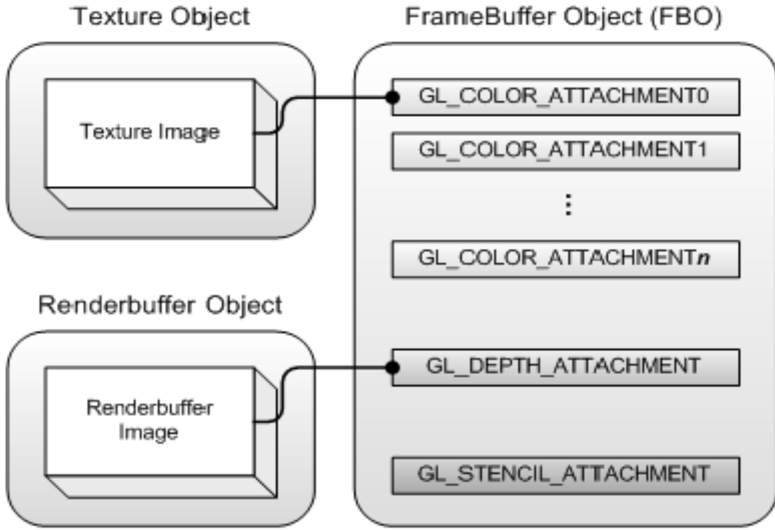

Figure 3 : Diagramme de Connectivité entre FBO, texture et Renderbuffer [15]

- **VBO**

Un vertex Buffer Object (VBO) est une mémoire tampon qui contient les informations relatives aux vertex : Position, couleur, normales. L'avantage des VBO c'est que les données sont directement chargées dans la mémoire de la carte graphique et restent accessibles jusqu'à la suppression du VBO contrairement à la méthode de chargement directe des vertex utilisée dans Listing 2.1 et par GLCU pour charger les triangulations.

- **PBO**

L'extension OpenGL pixel_buffer_object est très proche de vertex_buffer_object. Il élargit simplement l'extension vertex_buffer_object pour stocker non seulement les données de vertex, mais aussi les données de pixels dans les objets tampon. Cet objet tampon stockant des données de pixel s'appelle Pixel Buffer Object (PBO). Le principal avantage de PBO est le transfert rapide de données de pixels vers et depuis une carte graphique via DMA (Direct Memory Access) sans interruption des cycles CPU. Et l'autre avantage du PBO est le transfert DMA asynchrone. Comparons une méthode de transfert de texture classique avec un objet tampon pixel.

3. **Analyse des résultats**

GLCU génère des images ".raw" correspondant à différents types de résultats et stocke l’image dans un fichier. Le nom du fichier indique le type de résultat souhaité. Tableau ci-dessous résume les résultats obtenus selon différentes types des inputs

Tableau 1 : Les différents résultats de GLCU

Output	type	Input	Description
	Top	data/A04170.off data/blades_A04170_3.off	l’animation des polyèdres contenus dans des fichiers.
	face	data/A04170.off data/blades_A04170_3.off	l’animation des polyèdres contenus dans des fichiers en vue de face.
	Top	data/A04170.off	l’animation des polyèdres contenus dans des fichiers
	Top	data/blades_A04170_3.off	l’animation des polyèdres contenus dans des fichiers.
	Bottom	data/A04170.off data/blades_A04170_3.off	l’animation des polyèdres contenus dans des fichiers en vue de bas

	Top	data/A04170.off data/blades_A04170_3.off	Modèle géométrique d'une empreinte Image raw
	Top	data/A04170.off data/blades_A04170_3.off	Modèle géométrique d'une empreinte Surface triangulée représentant l'empreinte (dans une direction donnée)
	Top	data/A04170.off data/blades_A04170_3.off	Zbuffer : C'est une image qui peut être transformée simplement en modèle géométrique 3D (en surface triangulée).(zbuf.raw)
	top	data/A04170.off data/blades_A04170_3.off	Zbuffer cumulé : C'est une image qui peut être transformée simplement en modèle géométrique 3D (en surface triangulée) (zbuf.raw)

II. Profilage total de GLCU

Au sein de la GLCU, la première partie du traitement qui consiste à projeter la géométrie de l'outil et à extraire les valeurs du z-Buffer est entièrement réalisée avec OpenGL. En revanche la seconde partie qui consiste à traiter ces données est effectuée avec CUDA. En fait, GLCU contient différents type de calcul afin d'améliorer le code : glcu, glcuread, Glcpu, glcu_vbo et Glu_glut.

- **Glcu** : calcul sur GPU avec copie du zbuffer directement sur GPU.
- **Glcuread**: calcul sur GPU avec copie du zbuffer via le port PCI express .
- **Glcpu** : calcul sur CPU.
- **Glcu_vbo** : l'ajout d'un buffer VBO pour réduire l'accès au RAM.
- **Glcu_glut** : l'ajout de mode d'affichage glut.

Nous présenterons ici un moyen de profilage total de toutes les versions de GLCU, on utilisant deux cartes graphiques, Geforce GTX 980 avec 2048 cœurs CUDA et 4 Go de mémoire et Quadro P 5000 avec 2560 cœurs CUDA et 16 Go de mémoire. Comme le montre les résultats affichées dans les tableaux ci-dessous (2,3), on peut varier la résolution pour la carte Quadro P5000 jusqu'à 18000x18000 ce qui n'est pas le cas pour la carte Geforce GTX 980, avec laquelle, on ne peut pas utiliser des résolutions de grandes tailles et ceci est dû aux caractéristiques de chaque carte.

Tableau 2 : Temps de simulation pour Geforce GTX 980

Résolution	GLCU	GLCU_VBO	GLCU_GLUT	CLCU_READ	GLCPU
100	0.733	2.33	1.624	0.803	5,125
500	1.673	2.67	2.61	10.79	24,015
1000	3.12	4.67	4.05	78.71	91,99
2000	8.40	9.452	7.58	621.096	743,071
4000	42.22	49.15	29.77	5147.63	5854,19
6000	128.096	151.63	69.77	-	-
8000	315.5	340.17	-	-	-

Tableau 3 : Temps de simulation pour Quadro P5000

Résolution	GLCU	GLCU_VBO	GLCU_GLUT	CLCU_READ	GLCPU
500	1.40	0.85	1.53	1.52	3.39
1000	1.93	1.40	2.50	3.26	17.34
2000	5.77	4.72	5.50	12.21	124.58
4000	26.73	26.98	19.35	75.20	970.93
6000	86.94	85.14	51.34	250.27	3254.7
8000	203.99	202.37	110.9	586.58	7734.18
10000	399.59	392.11	210.1	1077.68	15478.2
12000	692.16	688.89	355.46	1370.25	-
14000	1101.58	1088.2	599.74	1782.94	-
16000	1680.27	1682.31	829.04	2464.26	-
18000	2481.91	2514.12	1299.4	3485.08	-
20000	3678.72	3534.09	1931.4	-	-
22000	-	4059.31	2769.5	-	-

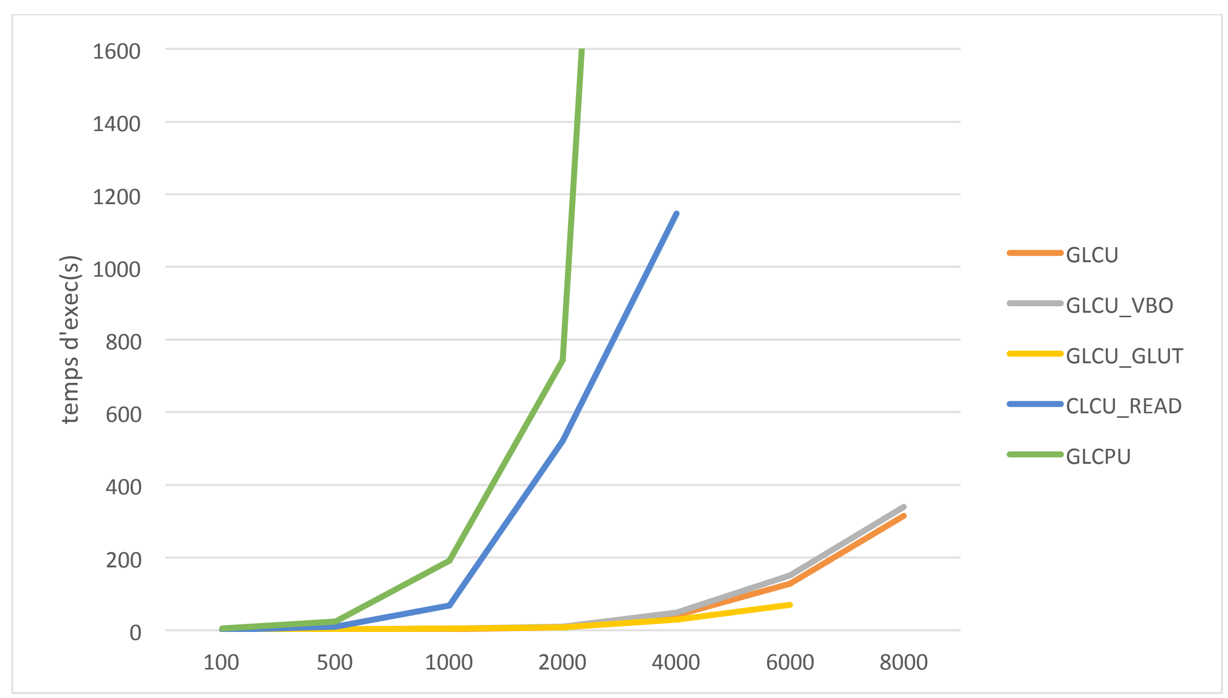

Figure 4 : Temps de calcul (Geforce GTX 980)

Nous avons effectué une analyse du code source de GLCU pour les différents GLCPU, GLCU, GLCUREAD, GLCUVBO et GLCUGLUT pour les deux cartes graphique Geforce GTX 980 et Quadro P500. Pour la première Catre, le temps de simulation pour GLCUREAD est presque parait à celle de GLCPU et avec l'utilisation des VBO la réduction des transferts entre la RAM GPU et CPU a été fortement réduite comme montre la dans la figure 4. Mais avec la présence d'une résolution maximale aux alentours de 8000 × 8000 le programme plante parce que CUDA tente d'allouer un espace plus grand que celui qui est disponible sur la carte graphique. Par contre avec la deuxième carte la résolution dépasse 18000x18000 due à l'espace mémoire (16 Go) libre sur la carte et avec aussi réduction considérable de temps de calcul sur CUDA comme le montre la figure 5.

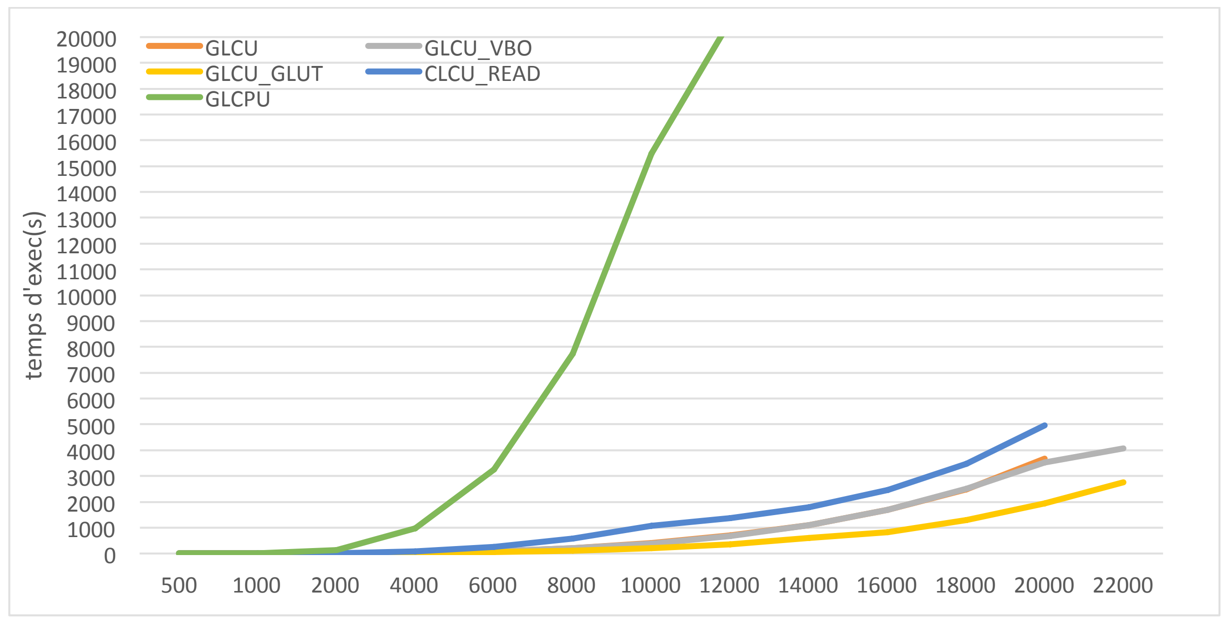

Figure 5: Temps de calcul (Quadro P 5000)

III. Profilage détaillé de GLCU

1. Analyse et profilage de calcul CUDA

Après les calculs des distances entre les surfaces animées d'une scène 3D en open GL. Les distances stockées dans un zbuffer sont ensuite fournies à des unités de calculs spécifiques programmées en CUDA (des kernels) pour le calcul de l'empreinte.

Dans cette partie, nous allons valider les paramètres et les techniques d'optimisation utilisées pour les kernels. L'expérimentation consiste à comparer les valeurs initiales, par rapport aux valeurs mesurées afin de vérifier que notre programme est optimal. On commence tout d'abord de vérifier le nombre de threads et le nombre de bloque optimal pour notre kernel afin d'atteindre le temps d'exécution optimal de kernel.

1.1 Calculateur d'occupation

La CUDA Occupancy Calculator vous permet de calculer l'occupation multiprocesseur d'un GPU par un kernel CUDA donné. L'occupation multiprocesseur est le rapport des warps actives au nombre maximal de warps pris en charge sur un multiprocesseur du GPU.

Occupancy = Active Warps / Maximum Active Warps

Chaque multiprocesseur dispose d'un ensemble de N registres disponibles pour les programmes de threads CUDA. Ces registres sont une ressource partagée qui est attribuée

entre les blocs de thread exécutés sur un multiprocesseur. Le compilateur CUDA tente de minimiser l'utilisation du registre afin de maximiser le nombre de blocs de threads qui peuvent être actifs dans la machine simultanément. Si un programme essaie de lancer un kernel pour lequel les registres utilisés par thread fois la taille du bloc de thread est supérieur à N, le lancement échouera.

Les principaux facteurs qui influent sur l'occupation sont l'utilisation de la mémoire partagée, l'utilisation du registre et la taille du bloc de thread. Pour Calculer l'occupation on utilisant occupancy calculator et les caractéristiques de notre carte graphique Geforce GTX 980 .

```
Device : "GeForce GTX 980"
  CUDA Driver Version / Runtime Version          7.5 / 7.0
CUDA Capability Major/Minor version number:    5.2
  Total amount of global memory:                 4095 MBytes (4294246400 bytes)
  (16) Multiprocessors, (128) CUDA Cores/MP:     2048 CUDA Cores
  GPU Max Clock rate:                            1216 MHz (1.22 GHz)
  Memory Clock rate:                             3505 Mhz
  Memory Bus Width:                              256-bit
  L2 Cache Size:                                 2097152 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536, 65536), 3D=(4096, 4096,
4096)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 2 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
```

Les figures (5,6 et 7) ci-dessous présentes chacun l'occupation selon le nombre maximum de thread par block, l'utilisation de la mémoire partagée et l'utilisation du registre. Le triangle rouge montre le nombre de warps optimal pour notre kernel.

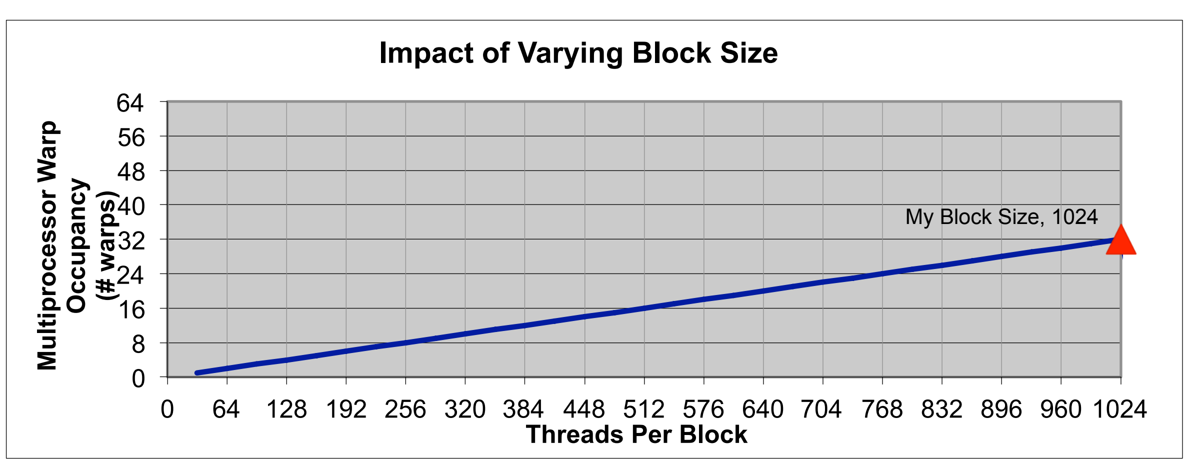

Figure 6 : Taille de Bloque

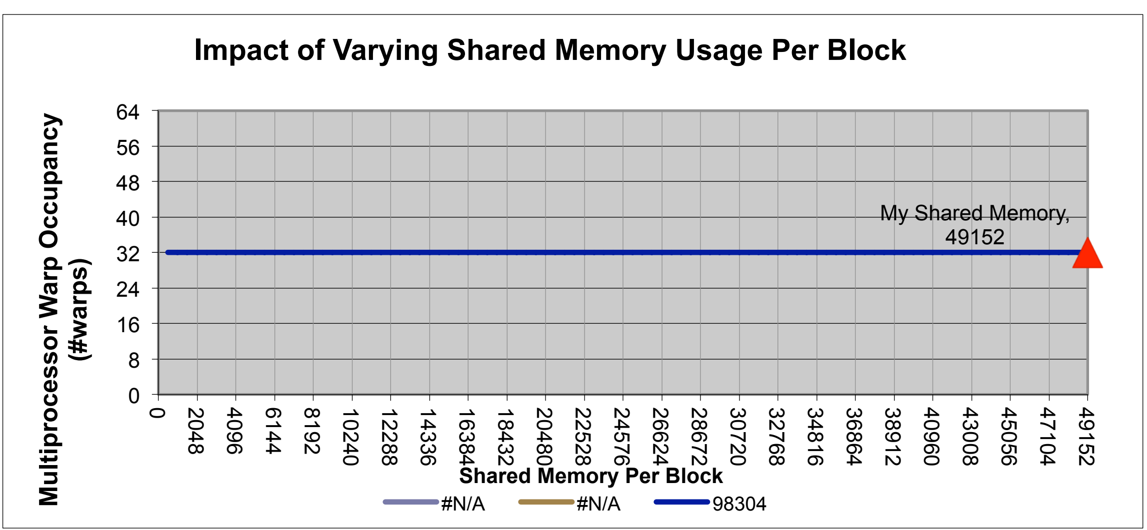

Figure 7 : l'utilisation de la mémoire partagée

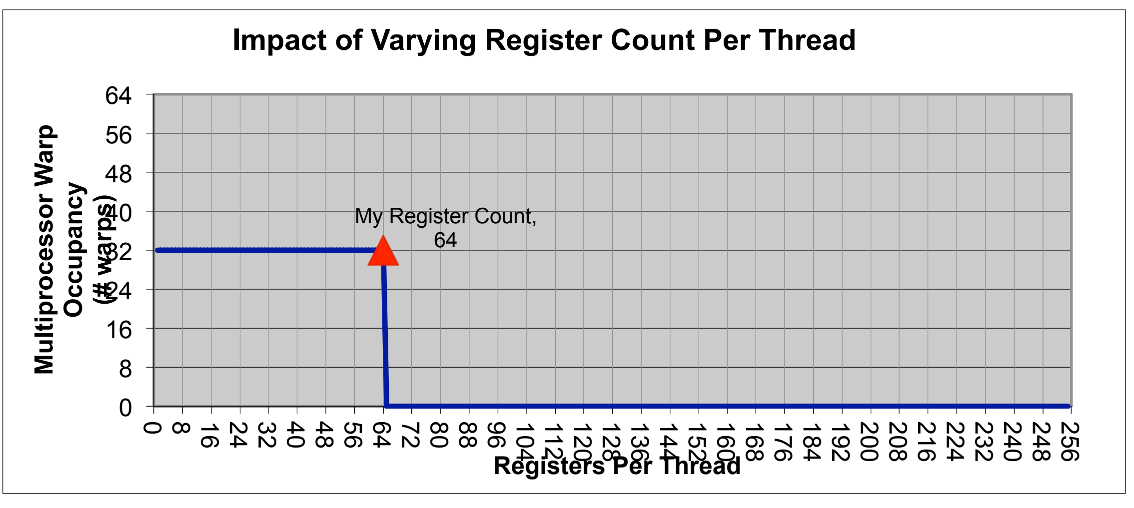

Figure 8 : L'utilisation de registre

Une occupation plus élevée ne signifie pas nécessairement une performance supérieure. Si un kernel n'est pas limité à la bande passante ou limité à la latence, l'augmentation de l'occupation n'augmentera pas nécessairement les performances.

En plus de calculateur, on a mesuré le temps de calcule de kernel par frame à l'aide de la fonction cudaEventElapsedTime() pour différents nombre de threads et différents nombre de blocks comme le montre l'exemple suivant.

```
cudaEvent_t start,stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start);
update_stack<<<blocks, threads>>>(stack, z_buffer, pixel_size);
cudaEventRecord(stop);
cudaEventSynchronize(stop);
float milisec=0;
cudaEventElapsedTime(&milisec,start,stop);
totalmilisec+=milisec;
printf("Cumulated time kernel : %4lf \n",totalmilisec);
```

Le tableau ci-dessous résume les différentes mesures pour le nombre de threads(16,32,64,256,1024) et le nombre de blocs varie de 4,8,16,32,64,256 jusqu' a 1024 blocs.

Tableau 4 : Temps de calcul de kernel en fonction de nombre de bloque et nombre de threads par bloque.

nombre de blocks	Nombre de threads				
	16	32	64	256	1024
4	6.42	3.42	1.598	0.423	0.131
8	3.092	1.588	0.804	0.224	0.084
16	1.552	0.890	0.412	0.128	0.0741
32	0.793	0.795	0.220	0.086	0.0751
64	0.415	0.221	0.123	0.758	0.075
256	0.133	0.0829	0.0757	0.75	0.072
1024	0.0983	0.0768	0.0767	0.0737	0.072

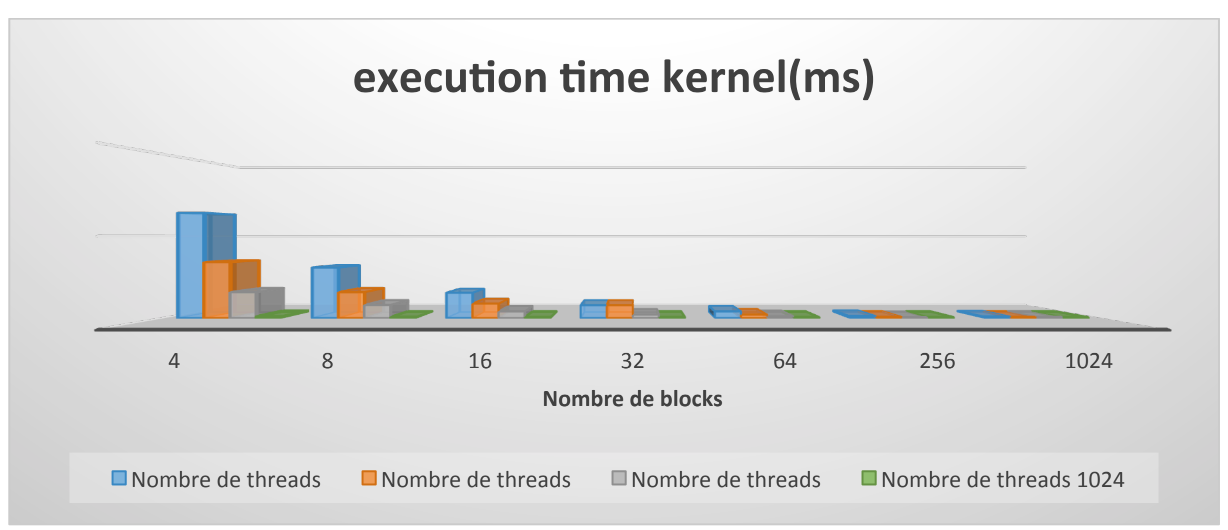

Figure 9 : temps d'execution de kernel

Comme montre dans la figure ci-dessus le temps d'exécution de kernel est fortement réduit pour les nombres de threads 1024 c'est à dire dim3 block (32,32). En fait, la dimension de bloc et de grille selon le calcul de l'occupancy et La solution utilisée est déjà optimale pour les kernels avec dimension de bloc (32,32)

1.2 L'utilisation de la bande passante

Parmi les règles d'optimisation dans le calcul CUDA est la bonne utilisation de bande passante .tableau ci-dessous montre l'efficacité de la bande passante par rapport la bande passante théorique pour les deux cartes graphiques.

Résolution	Bandwidth mesurée (GB/s) Geforce GTX 980	Bandwidth théorique (GB/s) Geforce GTX 980	Bandwidth mesurée (GB/s) Quadro P 5000	Bandwidth théorique (GB/s) Quadro P 5000
500	143,55	224	202.90	288
1000	175,12	224	210.43	288
2000	162,84	224	266.20	288
4000	182,10	224	250.86	288

8000	179,12	224	271.46	288
16000			279.64	288

1.3 Optimisation de temps d'accès mémoire

Comme nous avons décrit précédemment, le code CUDA prend en considération pratiquement toutes les règles d'optimisation de temps de calcul de kernel : Définir un nombre de blocks de 2 à100 fois égal au nombre de multiprocesseurs, définir plusieurs warps par multiprocesseurs et la bonne utilisation de bande passante. En plus, Le temps d'exécution est réparti en temps de calcul et le temps d'accès mémoire ce qui nécessite alors l'optimisation de temps d'accès mémoire surtout avec la présence de grand nombre de données. Passer par la mémoire partagée pour minimiser l'accès à la mémoire globale comme le montre l'exemple suivant.

```
void __global__ find_min_k(float *input, float *output, unsigned int nb_elt) {
extern __shared__ float sdata [];
```

2. Analyse et profilage de calcul Full Open GL

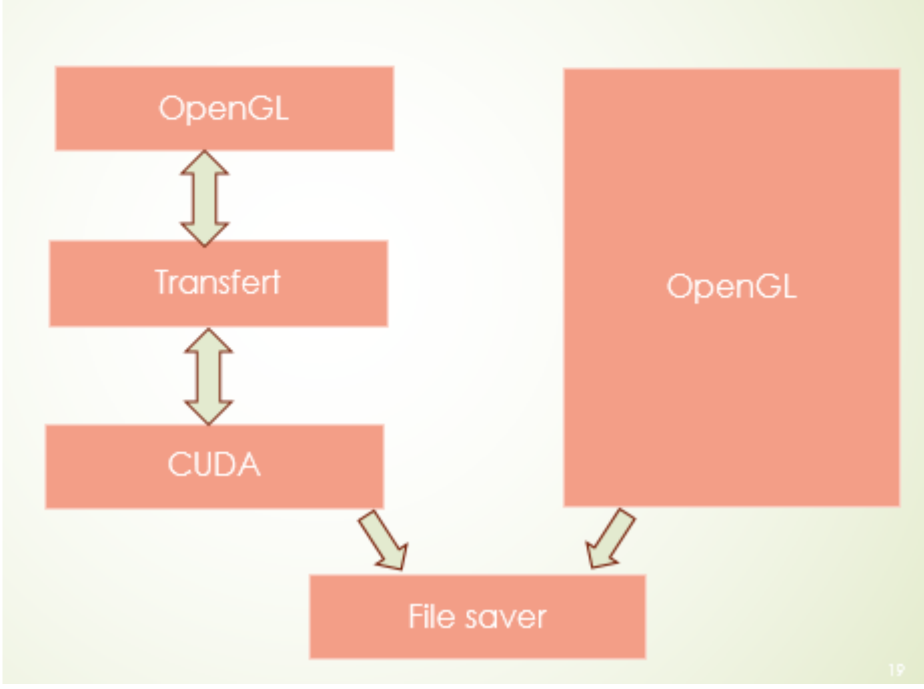

Figure 10: Organigramme simplifié du code GLCU

Un mode de calcul OpenGL a été ajouté à GLCU. Il permet de calculer l'empreinte directement à l’aide d'un shader (GLSL) et ce qui évite le transfert de données entre open GL et CUDA comme montre l’exemple ci-dessous. De ce fait, d'effectuer les calculs avec OpenGL, ce qui permet de profiter de l'accélération GPU. Le mode de calcul OpenGL a donc été ajouté à GLCU [saib]. GLCU peut ainsi être utilisé sur n'importe quelle plateforme compatible OpenGL tout en tirant les meilleures performances de parallélisation qu'elle puisse ouvrir grâce aux optimisations d'OpenGL

```
__global__ void update_stack(float *min, float *
   z_buffer) {
unsigned int tid = threadIdx.x + blockIdx.x *
   blockDim.x;

if (min[tid] > z_buffer[tid] || min[tid] == 255.0)
min[tid] = z_buffer[tid];

}
```

```
1  // texture precedente
2  uniform sampler2D readTex;
3  uniform vec2 resolution;
4
5  void main(){
6  // calcul des coordonees du fragment.
7  vec2 position = (gl_FragCoord.xy / resolution.xy);
8  // recupere la valeur du zbuffer actuel.
9  float z = gl_FragCoord.z;
10 // recupere la valeur de l’empriente precedente.
11 float zInput=texture2D(readTex, position).w;
12 // actualise la valeur de l’empreinte
13 gl_FragColor = vec4(z ,gl_Color.y , gl_Color.z , z<
      zInput ? z : zInput);
14 };
```

Pour le mode OpenGL, apporte un gain de performances proportionnel aux temps de simulation par rapport le mode CUDA pour différents nombre de frame. Par contre dans le mode CUDA, le gain de performances est minime, cela pourrait être dû à l'interopérabilité entre OpenGL et CUDA utilisée pour mapper le zbuffer calculé par OpenGL dans l'espace mémoire CUDA.

Le tableau ci-dessous et nous résume les résultats pour les deux cartes respectivement. La première colonne correspond aux résolutions utilisées, la deuxième colonne correspond à la mesure pour le mode de calcul CUDA pour 1000 et 2000 frames dont on va utiliser pour comparer. La troisième colonne correspond au mode full open GL ajouté pour différents nombre de frame aussi pour les deux cartes graphiques respectivement.

Frame	Open GL				CUDA			
	1000 (GTX 980)	2000 (GTX 980)	1000(Quadro P 5000)	2000(Quadro P 5000)	1000(GTX 980)	2000 (GTX 980)	1000(Quadro P 5000)	2000(Quadro P 5000)
Résolution								
500	0.84	1.32	1.08	1.10	1.44	2.35	1.22	1.17
1000	0.90	1.57	1.12	1.08	1.57	2.57	1.36	1.24
2000	1.59	2.25	1.31	1.20	2.06	3.52	1.66	2.60
4000	4.27	6.26	2.93	2.91	6.48	8.30	3.10	5.44
6000	9.25	14.12	5.81	6.23	10.81	17.05	6.01	11.11
8000	15.60	27.05	9.97	11.15	18.28	29.66	10.09	19.41
10000			14.22	17.44			15.33	29.78
12000			19.92	28.01			22.19	36.89
14000			24.15	35.66			29.83	43.74
16000			35.48	44.58			39.7	53.10
18000			41.12	66.93			52.5	76.19

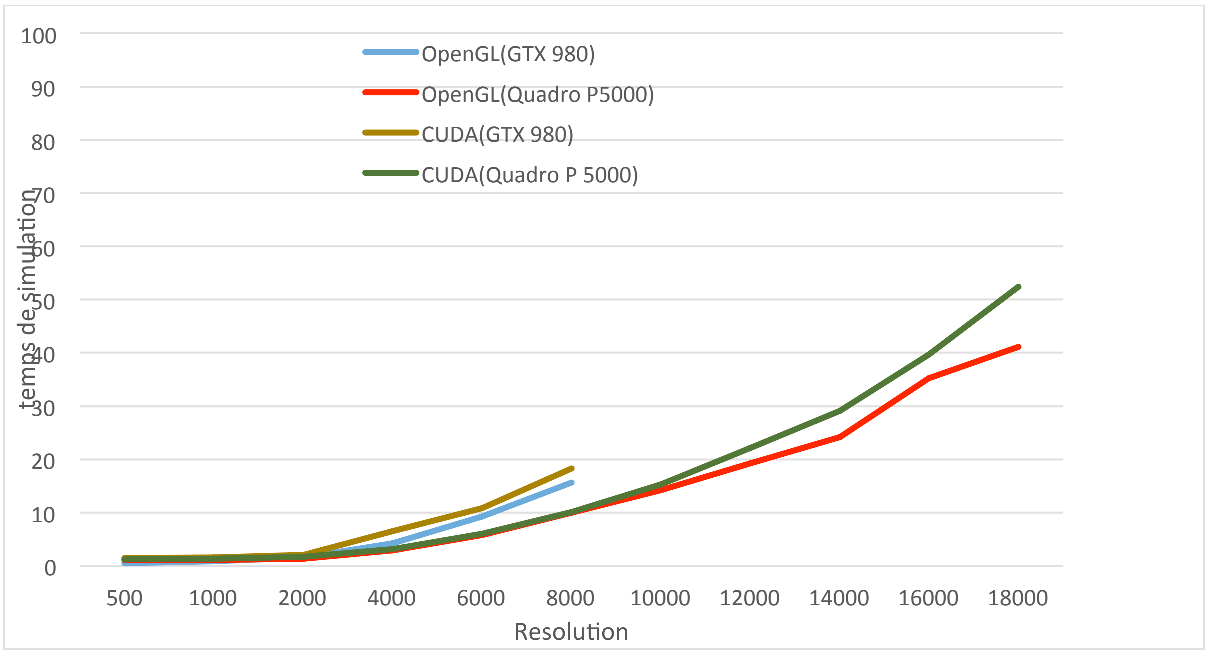

Figure 11: Temps de simulation pour 1000 frames

Les figures 11 et 12 présentent les résultats de la simulation en mode CUDA et Open GL respectivement pour un nombre de frame fixe (1000 et 2000) et pour deux cartes graphiques. Le mode OpenGL apporte une amélioration des performances proportionnelle à la durée de la simulation et à la résolution par rapport le mode CUDA.

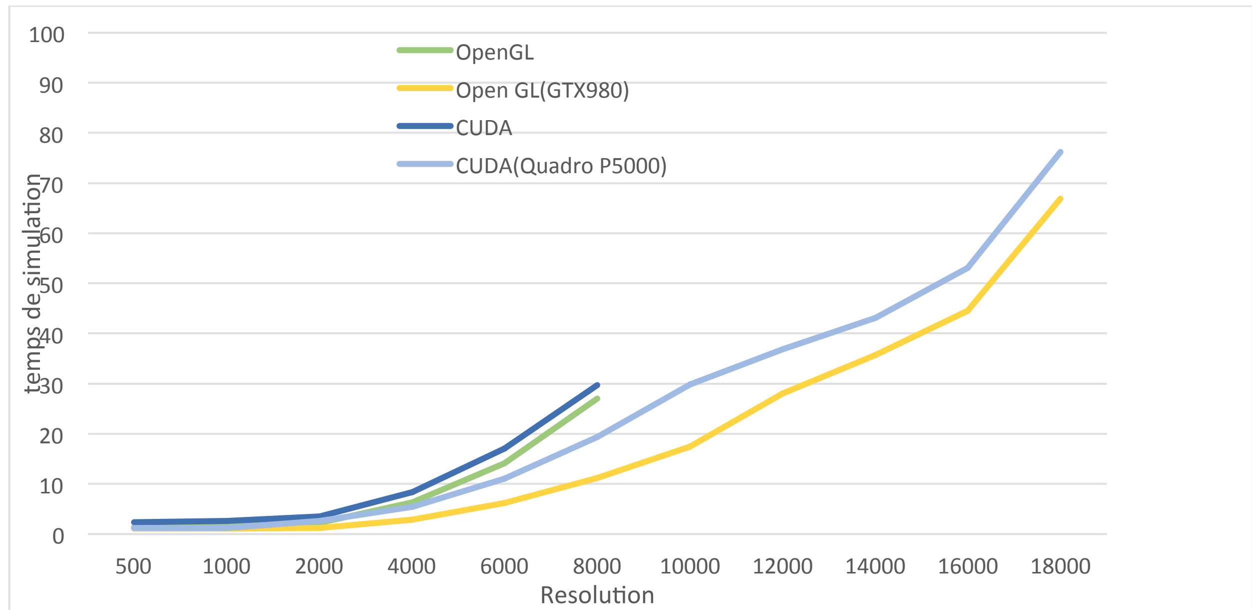

Figure 12: Temps de simulation pour 2000 frames

De même, on a choisi de mesure le temps de simulation pour différents nombre de frame (1000, 2000, 4000, 8000,100000) pour la carte graphique Quadro P5000 comme le montre la figure 13.on constate que le temps de simulation augmente proportionnel au nombre de frame pour les différents résolutions .

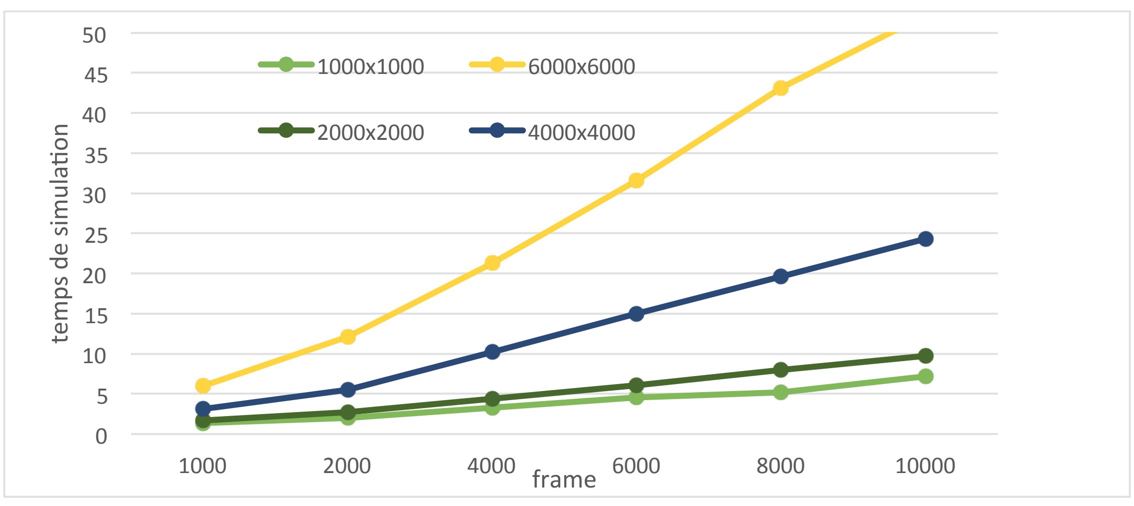

Figure 13 : temps de simulation pour différents nombre de frame

3. Transfert de données (Open GL → CUDA)

Le Transfert a pour rôle de récupérer le **zbuffer** calculée nativement par OpenGL, puis, le transmettre à CUDA pour y effectuer des calculs. Le contenu CUDA dispose de fonctions (Map/UnMap) qui permettent de représenter les buffers d'OpenGL dans l'espace mémoire CUDA .Le programme commence par créer le Buffer sous OpenGL et alloue l'espace mémoire, ensuite il fait appel à une fonction CUDA qui permet de récupérer un pointeur vers cette zone mémoire. En fait, les architectures GPU dispose des hiérarchies mémoire complexe .De ce fait, on a besoin dans notre programme de connaitre l'espace mémoire à allouer pour stocker les triangles de la frontière et de connaitre les index où enregistrer les vertex (sommets des triangles) pour éliminer les problèmes d'allocation mémoire.

Pour valider notre analyse, on a choisi deux cartes graphiques différents au niveau de taille mémoire (4 et 16 Go) pour évaluer notre mesure de transfert de Zbuffer de Open GL a CUDA et lancement les calculs CUDA sur le Zbuffer, puis l'enregistrement de résultat de ces calculs. Le tableau ci-dessous résume le temps de transfert de données et le temps de calcul de l'empreinte pour les deux cartes graphiques respectivement.

Tableau 5 : temps de transfert

Résolution	CUDA Gefroce GTX 980	Transfert Geforce GTX 980	CUDA Quadro P5000	Transfert Quadro P5000
500	0,029	0,032	0,018	0,22
1000	0,181	0,23	0,118	0,149
2000	1,288	1,751	0,842	1,115
4000	10,06	14,18	6,609	8,834

6000	34,26	48,37	22,441	29,91
8000	82,173	115,69	54,43	73,54
10000			106,4	146,66
12000			184,04	254,25
14000			293,32	409,19
16000			437,29	614,05

Temps de transfert de donnes entre open GL et CUDA est élevé par rapport temps de calcul de l'empreinte le plus important, concerne l'influence du rapport temps de calculs sur temps de transferts, sur les performances du traitement sur GPU. En effet, au travers de plusieurs exemples, nous avons montré que, malgré la parallélisation massive offerte par les cartes graphiques modernes, les performances d'un code de calcul sur GPU peuvent s'écrouler si les temps de transferts sont trop grands par rapport aux temps de calcul comme le montre la figure 14.

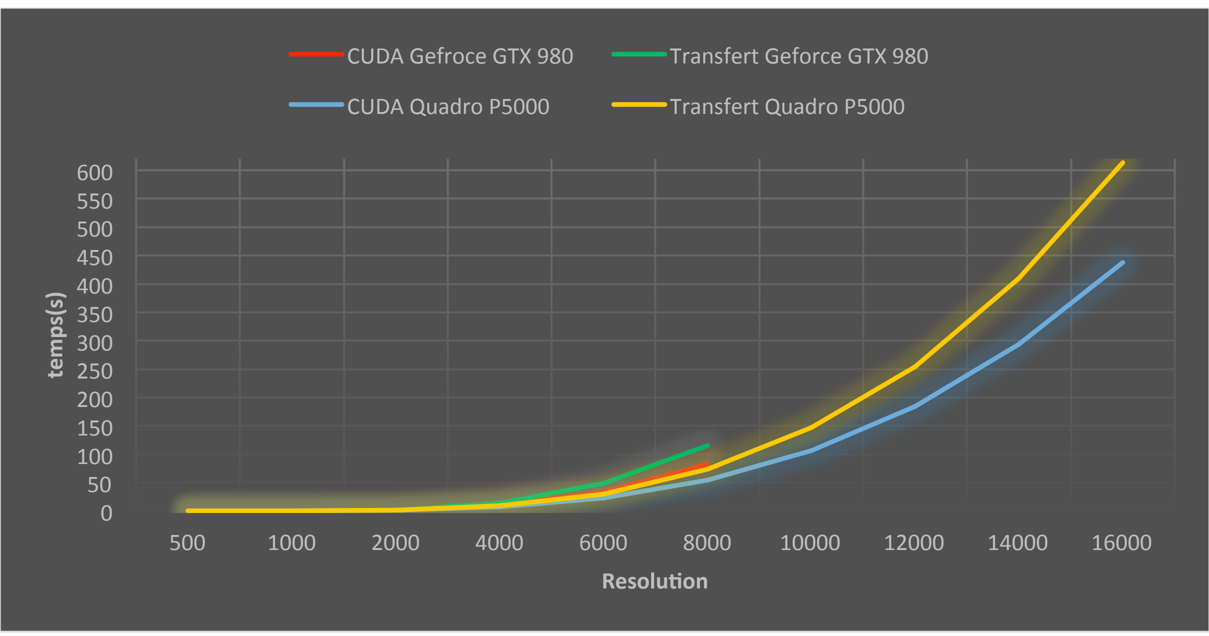

Figure 14 : temps de transfert par rapport le temps de calcul de l'empreinte

Conclusion et perspectives

Nous avons effectué une analyse et profilage détaillée du code source de GLCU. En fait le code est apporté déjà quelques améliorations, dont la réduction des transferts entre la RAM GPU et CPU avec l'utilisation des VBO, transformation de calcul de l'empreinte (kernels CUDA) au des shader en GLSL afin d'éliminer le transfert de données mais malgré tous les améliorations le temps de simulation reste toujours important et avec une résolution limité (ne dépasse pas 8000x8000 pixel).De ce fait, on a implémenté le code de GLCU sur une autre carte graphique plus performante que celle GTX980 avec 4 fois de taille mémoire et plus de nombre de cœurs afin d'éteindre des résolutions maximale (18000x18000).

Dans nos futurs travaux, on peut utiliser plus des buffers (FBO) afin de réduire le temps de transfert de données entre les calculs open GL et les calculs CUDA .on peut envisager aussi le portage de tous les calculs au mode CUDA sous forme de kernels pour éliminer les problèmes de transfert de données et accélérer les calculs ce qui nécessite dans ce cas des changements globale de code source.

Bibliographie :

[1] Dyken, C., Ziegler, G., Theobalt, C., Seidel, H.-P., 2008. High-speed Marching Cubes using HistoPyramids, in: Computer Graphics Forum. Wiley Online Library, pp. 2028–2039.

[2] Kim, Y.J., Varadhan, G., Lin, M.C., Manocha, D., 2004. Fast swept volume approximation of complex polyhedral models. Comput.-Aided Des. 36, 1013–1027. doi:10.1016/j.cad.2004.01.004

[3] Kostas, T., Stab, O., 2016. GPU approach of drilling simulation.

[4] Lee, S.-K., Ko, S.-L., 2002. Development of simulation system for machining process using enhanced Z map model. J. Mater. Process. Technol. 130, 608–617.

[5] Lee, S.W., Nestler, A., 2012. Virtual workpiece: workpiece representation for material removal process. Int. J. Adv. Manuf. Technol. 58, 443–463. doi:10.1007/s00170-011-3431-2 Lorensen, W.E., Cline, H.E., 1987.

[6] Marching Cubes: A High Resolution 3D Surface Construction Algorithm, in: Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH '87. ACM, New York, NY, USA, pp. 163–169. doi:10.1145/37401.37422

[7] CUDA C Programming Guide (2012), http://developer.nvidia.com/cuda/cuda-downloads.

[8] NVIDIA Fermi Compute Architecture Whitepaper (2009), http://www.nvidia.com/content /pdf/fermi_white_papers/nvidia_fermi_compute_architecture_whitepaper.pdf

[9] Saib, H., Stab, O., 2016. Construction d'un modèle géométrique 3D dans un pipeline OpenGL.

[10] Stein, A., Geva, E., El-Sana, J., 2012. CudaHull: Fast parallel 3D convex hull on the GPU. Comput. Graph. 36, 265–271. doi:10.1016/j.cag.2012.02.012

[11] Sud, A., Otaduy, M.A., Manocha, D., 2004. DiFi: Fast 3D distance field computation using graphics hardware, in: Computer Graphics Forum. Wiley Online Library, pp. 557–566.

[12] Weinert, K., Du, S., Damm, P., Stautner, M., 2004. Swept volume generation for the simulation of machining processes. Int. J. Mach. Tools Manuf. 44, 617–628. doi:10.1016/j.ijmachtools.2003.12.003

[13] Zhang, W., Majdandzic, I., 2010. Fast triangle rasterization using irregular z-buffer on cuda

[14] OpenGL programming guide : the official guide to learning OpenGL, version 4.3 / Dave Shreiner, Graham Sellers, John Kessenich, Bill Licea-Kane ; the Khronos OpenGL ARB Working Group.---Eighth edition. pages cm Includes index. ISBN 978-0-321-77303-6