SIMSURF
Vers une simulation réaliste des états de surfaces par calculs massivement parallèles sur processeurs graphiques

Félix Abecassis
École Normale Supérieure de Cachan

Rapport de Stage
Février - Juillet 2012
EPITA CSI 2012

Laboratoire Universitaire de Recherche en Production Automatisée
Laboratoire de Mécanique et Technologie
61, avenue du Président Wilson – 94 235 Cachan cedex – France
Tél. +33 1 47 40 22 15 – Fax. +33 1 47 40 22 20
felix.abecassis@gmail.com – http://www.lurpa.ens-cachan.fr/

Table des matières

Chapitre 1

Introduction

Ce rapport de stage détaille les travaux effectués dans le cadre de mon stage de fin d'études de l'École Pour l'Informatique et les Techniques Avancées (EPITA), majeure Calcul Scientifique et Image (CSI). Ce stage s'est déroulé à l'École Normale Supérieure de Cachan au sein du Laboratoire Universitaire de Recherche en Production Automatisée et en collaboration avec le Laboratoire de Mécanique et Technologie situé sur le même campus. Le travail de ce projet s'inscrit dans le cadre du projet SIMSURF.

1.1 ENS Cachan

1.1.1 Présentation

L'École Normale Supérieure de Cachan (ENS Cachan) est une grande école française la fonction publique faisant partie du réseau des Écoles Normales Supérieures (avec l'ENS Ulm et l'ENS Lyon). C'est une école de la fonction publique dispensant une formation de haut niveau par la recherche et formant des enseignants pour les universités et les classes préparatoires.. L'ENS Cachan est constituée de 17 départements d'enseignement, 14 laboratoires et 4 instituts de recherche en sciences fondamentales, en sciences pour l'ingénieur et en sciences humaines et sociales. L'école compte environ 1320 étudiants normaliens et 750 étudiants venant d'autres universités en France ou à l'étranger. Les étudiants sont encouragés à suivre le cursus universitaire classique de recherche : Licence, Master puis Doctorat, de nombreux étudiants passent également l'examen d'agrégation. 250 doctorants et 70 postdoctorats sont actuellement à l'ENS Cachan.

L'ENS Cachan fait partie du pôle de recherche et d'enseignement supérieur UniverSud Paris et sa migration sur le campus du plateau de Saclay est en cours de préparation.

1.1.2 LURPA

Le Laboratoire Universitaire de Recherche en Production Automatisée (LURPA, www.lurpa.ens-cachan.fr) fait partie du Département de Génie Mécanique de l'ENS Cachan. Les deux axes principaux de recherche sont :

- Prise en compte d'une géométrie réaliste et cohérente dans les activités de conception, de fabrication et de contrôle. La recherche est réalisée au sein de l'équipe Géo3D.
- Amélioration de la sûreté de fonctionnement et des performances des systèmes automatisés discrets. La recherche est réalisée au sein de l'équipe Ingénierie des Systèmes Automatisés (ISA).

Le stage s'est déroulé au sein de l'équipe Géo3D puisque l'objectif du projet est la simulation réaliste des états de surface. Le travail a été effectué dans les locaux du LURPA. L'encadrement a été assuré par Christopher Tournier, professeur des Universités, et Sylvain Lavernhe, maître de conférences.

1.1.3 LMT

Le Laboratoire de Mécanique et Technologie (LMT, www.lmt.ens-cachan.fr est une Unité Mixte de Recherche commune à l'ENS Cachan, au CNRS et à l'Université Pierre et Marie Curie. Les axes de recherches concernent la modélisation des solides et des structures : mécanique des matériaux, mécanique expérimentale, simulation numérique et calcul haute performance. Le LMT possède un centre de calcul composé de 48 nœuds de calcul (528 cœurs) et de 8 nœuds graphiques (NVIDIA GeForce GTX 260).

L'encadrement a été assuré par Pierre-Alain Boucard, professeur à l'ENS Cachan, anciennement professeur des Universités à l'IUT de Cachan.

1.2 Projet SIMSURF

1.2.1 Présentation

Le projet dans lequel s'est inscrit le stage est intitulé *SIMSURF : vers une simulation réaliste des états de surfaces par calculs massivement parallèles sur processeurs graphiques*. Ce projet est financé par l'institut FARMAN, cet institut finance les projets rassemblant plusieurs laboratoires de l'ENS Cachan.

Les produits manufacturés à l'heure actuelle possèdent des surfaces qui interagissent avec l'environnement afin d'assurer des fonctions techniques : étanchéité, frottement, propriétés optiques, écoulement hydro ou aéro dynamique. Ces fonctions ne peuvent être assurées que si l'aspect de la surface possède une géométrie déterminée. Pour diminuer le temps de conception d'une nouvelle pièce et réduire les coûts, des logiciels de FAO sont utilisés pour simuler l'aspect de la surface usinée, moins de prototypages réels sont requis. Actuellement, les algorithmes de simulation implémentés dans les logiciels comme CATIA utilisent des représentations idéales des outils d'usinage (par exemple une sphère ou un cylindre) et ne permettent donc pas de prendre en compte la géométrie réelle de l'outil ainsi que ses éventuels défauts à cause de l'usure. De plus, ces algorithmes ne sont pas temps réel et donc ne permettent pas de choisir une portion de la surface sur laquelle on souhaite obtenir plus de précision (simulation multi-échelle).

1.2.2 CUDA

Le LURPA possédait au début du stage une maquette basée sur l'algorithme appelé **Z-buffer**. Cependant, le temps de calcul excessivement long de la simulation restreignait son utilisation à une très petite portion. Il a donc été décidé d'utiliser l'architecture CUDA développée depuis quelques années par NVIDIA. CUDA permet d'utiliser les capacités de calcul massivement parallèle des cartes graphiques pour implémenter des algorithmes coûteux mais exhibant un fort parallélisme. Ces algorithmes peuvent n'avoir aucun rapport avec l'affichage 3D (General-Purpose Processing on Graphics Processing Units : **GPGPU**). En utilisant la puissance de cette architecture, le temps de calcul peut être largement accéléré par rapport à une implémentation CPU, même multicore. Le gain de performance varie selon l'application scientifique et oscille généralement entre un facteur 2

et un facteur 100. Le site de NVIDIA propose une présentation d'applications scientifiques accélérées par l'utilisation de CUDA : www.nvidia.fr/object/cuda_apps_flash_new_fr.html.

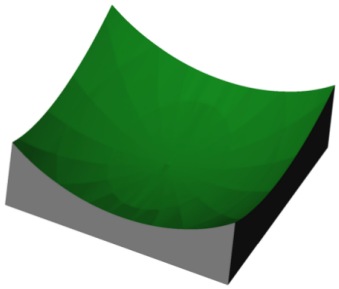
Le LMT possède une bonne expérience du calcul haute performance notamment avec CUDA, par contre le LURPA ne possédait aucune expertise dans ce domaine. Une formation continue à CUDA a été nécessaire pour comprendre les subtilités de son architecture et comment optimiser efficacement.

1.2.3 Objectifs

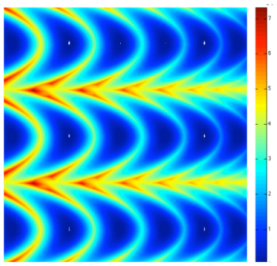
Les objectifs du projet, définis dans le document de départ se trouvant en annexe de ce rapport, étaient :

- Le découpage intelligent du problème pour exploiter au maximum l'architecture CUDA.
- La visualisation des résultats de simulation sans échanges de données avec des logiciels externes.
- La prise en compte de plusieurs échelles de simulation et le passage en temps réel de l'une à l'autre. L'utilisateur doit pouvoir zoomer sur une partie de la surface et obtenir immédiatement une nouvelle visualisation avec plus de précision.
- Fournir un démonstrateur avec les fonctionnalités ci-dessus.
- La corrélation entre simulation et états de surface réels, c'est-à-dire la comparaison des résultats de simulation avec l'aspect réel de la surface après usinage.

Ce document initial présente une comparaison des algorithmes de simulation du logiciel CATIA et du prototype du LURPA. Cette comparaison est illustrée dans la figure suivante, extraite du document cité :



Simulation CATIA : zone de 100x100 mm, 30min de calcul. Le maillage utilisé par la simulation est visible !!



Simulation Nbuffer sur 1 cpu : zone de 5x5 mm, 10min de calcul, écarts géométriques simulés de l'ordre du micromètre.

1.3 Plan

Dans un premier temps, nous présenterons le principe de fonctionnement de l'architecture massivement parallèle CUDA. Ensuite, nous étudierons l'algorithme utilisé lors du stage pour simuler l'état de surface d'une pièce après usinage. Puis, nous analyserons les différentes stratégies de parallélisation que nous avons experimentées afin d'exploiter au maximum les forces de l'architecture CUDA tout en évitant de tomber dans ses faiblesses, nous présenterons également différentes optimisations introduites dans le projet afin d'accélérer l'exécution CPU et GPU. L'algorithme choisi effectuant de nombreux calculs sur nombres flottants, nous étudierons sa stabilité numérique et comment elle a été améliorée. Enfin, nous détaillerons l'architecture choisie pour le démonstrateur du projet ainsi l'implémentation de la partie visualisation.

Chapitre 2

Configurations

Dans ce chapitre seront présentées les configurations matérielles utilisées tout au long de ce rapport pour évaluer les performances de nos algorithmes. Le développement a été effectué sous un environnement Linux.

2.1 Configuration ENS Cachan

Cette configuration est celle fournie par l'ENS Cachan pour le développement du projet. Ce ne fut pas la configuration initiale lors du début du stage. La configuration initiale (pendant 1 mois et demi) était bien moins puissante : une machine double-cœur et une carte graphique GTX 260. À la réception de la nouvelle configuration, celle-ci fut installée immédiatement, aucun benchmark de ce rapport n'a été effectué sur la première configuration.

2.1.1 Composants

Les caractéristiques techniques de la machine sont les suivantes :

- **CPU :** Intel Xeon X5650 (2.67GHz), 6 cœurs physiques, 12 cœurs virtuels (**hyperthreading**), 12 Mo cache L2 (Smart Cache), SSE 4.2, ISA `x86_64`.
- **RAM :** 8 Go DDR3.
- **GPU 1 :** NVIDIA Quadro FX 1800.
- **GPU 2 :** NVIDIA Quadro 4000.

La première carte graphique Quadro est utilisée uniquement pour l'affichage OpenGL, les kernels CUDA sont exécutés sur la Quadro 4000, bien plus efficace pour cette tâche. De plus, cette carte n'étant pas utilisée par le **X Window System**, il n'y a pas de limite de temps d'exécution pour les kernels.

2.1.2 Prix

Les deux configurations utilisées dans ce rapport sont très différentes, nous allons donc nuancer les écarts de performances observés en les rapportant au prix de chaque composant. Les prix ont été trouvés sur amazon.com en août 2012 et ne tiennent donc pas compte des éventuelles réductions dont aurait pu bénéficier l'ENS lors de l'achat de cette machine.

Cette configuration est vendue comme permettant le calcul haute performance sur une machine de bureau, le prix individuel élevé des composants le reflète :

- **Intel Xeon X5650 :** $1003.99

– **NVIDIA Quadro 4000 :** $703.63

2.2 Configuration personnelle

Cette configuration est personnelle, il s'agit de ma machine de bureau. Cette configuration est à la base une machine de jeu : le processeur est situé en entrée de gamme alors que la carte graphique était située en moyenne gamme lors de sa sortie.

2.2.1 Composants

– **CPU :** AMD Phenom II X4 955 (3.2 GHz), 4 cœurs physiques, 2 Mo cache L2, SSE 4a, ISA `x86_64`.
– **RAM :** 4 Go DDR3.
– **GPU :** NVIDIA GeForce GTX 560 Ti.

Ni la carte graphique ni le processeur ne sont orientés calcul haute performance. Le processeur possède moins de cœurs et un cache plus petit que le précédent. La carte graphique est orientée jeux vidéos : l'affichage 3D est très performant. Beaucoup de transistors sont utilisés pour le pipeline de rendu 3D, moins de place est disponible pour les composants pouvant exécuter des kernels CUDA.

2.2.2 Prix

– **AMD Phenom II X4 955 :** $148.99
– **NVIDIA Quadro 4000 :** $219.99

2.3 Comparaison des cartes graphiques

Nous allons maintenant présenter un comparatif des deux cartes graphiques utilisées.

2.3.1 Comparatif CUDA

Les caractéristiques essentielles affectant la performance CUDA sont listées dans le tableau suivant. Ces informations ont été récupérées en utilisant l'API CUDA et en consultant les spécifications sur le site de NVIDIA.

	Quadro 4000	GTX 560 Ti
CUDA capability	2.0	2.1
Mémoire globale	2048 Mo	1024 Mo
Multiprocesseurs	8	8
Cœurs CUDA	256	384
Fréquence d'horloge GPU	0.95 GHz	1.76 GHz
Fréquence d'horloge mémoire	1404 MHz	2100 MHz
Taille du bus mémoire	256 bits	256 bits
Bande passante mémoire	89.6 Go/s	128 Go/s
Taille du cache L2	512 Ko	512 Ko
Unified Addressing	✔	✔
ECC	✖	✖
OpenGL	4.1	4.1

Les cœurs CUDA (**CUDA cores**) étaient autrefois appelés **shader processors**. Il s'agit toujours du même composant matériel. Historiquement les premiers calculs sur GPGPU utilisaient les langages de **shader** de façon détournée, le langage CUDA a ajouté une abstraction permettant de programmer ces unités de traitement en utilisant des extensions au langage C.

La carte GTX semble supérieure sur tous les points malgré son prix inférieur. Nous verrons dans ce rapport que cette première impression est confirmée pour les calculs en simple précision, par contre en double précision la Quadro 4000 se révèle plus rapide. Les cartes de la gamme Quadro et Tesla ont nettement plus de mémoire que les cartes GeForce (jusqu'à 6 Go pour les cartes de la gamme Tesla). Cet écart peut faire la différence pour certaines applications de calcul scientifique, mais comme nous allons le voir dans ce rapport dans tous les cas que nous avons testés, l'occupation mémoire n'a jamais été un facteur limitant.

2.3.2 Bande passante mémoire

La bande passante mémoire a été calculée en utilisant le programme **bandwidthTest** du SDK CUDA. Ce programme utilise la routine CUDA `cudaMemcpy`.

	Quadro 4000	GTX 560 Ti
CPU → GPU	3241 Mo/s	3132 Mo/s
GPU → CPU	4009 Mo/s	2588 Mo/s
GPU → GPU	72322 Mo/s	110787 Mo/s

L'écart observé pour un transfert de l'accélérateur vers le processeur s'explique par le fait que le processeur utilisé est plus moderne. La différence pour un transfert interne à la carte graphique est expliquée par le fait que la GTX est une carte plus récente, ces chiffres sont cohérents par rapport aux spécifications fournies par NVIDIA.

2.3.3 Benchmarks SDK CUDA

Afin de comparer les deux cartes graphiques, nous pouvons utiliser certains exemples du SDK CUDA. Ces exemples peuvent donner un indicateur de performance de la carte graphique. Nous avons utilisé l'exemple d'implémentation de la simulation physique `nbody` en simple et double précision ainsi que l'algorithme de multiplication matrice-matrice de la bibliothèque **cuBLAS** (adaptation CUDA de la bibliothèque d'algèbre linéaire optimisée **BLAS**).

	Quadro 4000	GTX 560 Ti
nbody 32-bit	231.4 GFlop/s	475.9 GFlop/s
nbody 64-bit	115.8 GFlop/s	87.6 GFlop/s
cuBLAS (multiplication)	286.1 GFlop/s	583.8 GFlop/s

La GTX 560 est environ 2 fois plus rapide sur les deux exemples en simple précision alors qu'elle est 3 fois moins chère. La Quadro est une carte graphique professionnelle dédiée aux applications haute performance. Le calcul scientifique nécessite généralement de la double précision : la Quadro 4000 est plus efficace en double précision que la GTX 560 (environ 30% plus rapide).

Cette différence s'explique par la volonté de NVIDIA de segmenter le marché en proposant des cartes grand public d'un côté et des cartes professionnelles dédiées au calcul

haute performance de l'autre. La performance en double précision d'une carte GTX d'architecture `GF114` est environ 1/6 de sa performance simple précision alors que pour les Quadro le rapport est environ 1/2. La justification technique sous-jacente est que tous les cœurs CUDA ne peuvent pas exécuter des opérations 64-bit. Sur l'architecture `GF114` un cœur sur 3 est compatible 64-bit, et l'exécution s'effectue à une vitesse 1/2 de la vitesse 32-bit , nous avons donc un rapport de vitesse de 1/6 au total, ce que nous avons observé dans le benchmark précédent.

2.3.4 Fiabilité

Un autre point important à considérer est la fiabilité de la carte. En fonction de paramètres externes comme la température, des interférences électriques ou magnétiques, les rayons cosmiques, la radioactivité naturelle... un bit de la RAM peut spontanément être inversé (on parle de **soft error**).

Dans le domaine du calcul scientifique, une inversion d'un seul bit durant un calcul peut entraîner un résultat totalement faux. Au contraire dans le cas d'un affichage 3D, une inversion spontanée n'aura probablement aucune conséquence.

Les cartes de la gamme professionnelle Quadro ont été testées de manière plus poussée avant leur mise en vente, elles sont réputées plus fiables que les cartes GeForce. L'article Haque et Pande (2009) examine la fiabilité des cartes NVIDIA du projet de calcul distribué **Folding@Home** et confirme cette différence.

Cependant, l'utilisation de la commande `nvidia-smi` a montré que la GTX 560 est mieux refroidie que la Quadro 4000 : jusqu'à **30** degrés de différence lors d'une série intensive de calculs. L'avantage prétendu de fiabilité de la Quadro par rapport à GTX n'est pas assuré à cause d'un système de refroidissement insuffisant. De plus, la Quadro 4000 ne possède pas de mécanismes de correction d'erreur sur sa RAM (Error Correcting Code : **ECC**), si une erreur survient, elle ne sera pas corrigée.

Suite à ces observations, le choix de la Quadro 4000 ne semble pas être le meilleur pour du calcul scientifique simple précision. Si une fiabilité absolue est indispensable une carte avec ECC doit être choisie (Quadro 5000 minimum ou une carte de la gamme Tesla). Si la fiabilité est moins importante que la performance et si la simple précision suffit, une carte GeForce de type GTX 680 ou GTX 690 devrait être utilisée à la place.

Dans le cadre de notre application, une erreur sur un bit affectera au pire un seul brin de la grille. La fiabilité semble donc moins importante que pour d'autres applications de calcul scientifique où, par un effet boule de neige, le résultat final pourrait être radicalement différent.

Concernant le CPU utilisé à l'ENS, il remplit bien sa part du contrat en tant que processeur professionnel : il effectue un code correcteur d'erreurs sur plusieurs bits à la fois. Cette information est donnée par la commande Linux `dmidecode`.

Chapitre 3

CUDA

Dans ce chapitre nous allons présenter les principes du modèle de programmation CUDA. Les informations de ce chapitre sont issues de deux sources : NVIDIA (2012) et Farber (2011).

3.1 Extension C

D'un point de vue développeur, CUDA est une API pouvant être utilisée depuis le `C` ou le `C++` en proposant des constructions permettant d'étendre ces langages. Ces extensions permettent de définir des fonctions spéciales appelées **kernels** qui seront executées sur un accélérateur de type NVIDIA (seules les cartes NVIDIA supportent CUDA). Lorsqu'un kernel est appelé depuis du code CPU, cette fonction sera exécutée sur l'accélérateur de façon massivement parallèle, chaque instance de cette fonction est appelée **thread**.

Voici un exemple de code CUDA implémentant un kernel qui effectue une addition de deux vecteurs de `float` de taille N. Le résultat est stocké dans un troisième vecteur. Un kernel est déclaré comme une fonction C standard en utilisant l'attribut supplémentaire `__global__` :

```
__global__ void VecAdd(float* A, float* B, float* C)
{
  int i = threadIdx.x;
  C[i] = A[i] + B[i];
}

int main()
{
  ...
  // Kernel invocation with N threads
  VecAdd<<<1, N>>>(A, B, C);
  ...
}
```

Nous pouvons observer à travers cet exemple la force de l'API CUDA : le code exécuté sur l'accélérateur NVIDIA est écrit en utilisant du code `C` habituel. Cette stratégie rend le calcul haute performance accessible pour les développeurs et scientifiques familiers avec le `C`. Cette fonctionnalité contraste avec la difficulté d'accès du développement d'une carte FPGA ou d'un ASIC. Cependant, nous verrons par la suite que pour obtenir de bonnes performances, une connaissance avancée de l'architecture CUDA et de son modèle de programmation reste nécessaire.

Le nombre de processus à lancer pour cette fonction est paramétré dynamiquement depuis le code CPU en spécifiant la **configuration d'exécution** via la syntaxe CUDA

suivante : `<<<grid_dim, block_dim>>>`. `grid_dim` sont les dimensions de la grille de calcul et `block_dim` sont les dimensions d'un bloc de la grille.

Si nous lançons N threads pour effectuer l'addition élément par élément de nos deux vecteurs, nous devons lors de l'exécution du kernel pouvoir identifier chaque thread afin de déterminer l'indice des éléments à additionner. Dans notre exemple précédent, chaque thread possède un indice unique accessible depuis la variable native `threadIdx`. Cette variable est une structure à trois dimensions : (x, y, z). L'ensemble des threads de cet espace 3D forme ce que l'on appelle un **bloc**. Le choix des dimensions sur chaque axe est déterminé depuis le code CPU, au lancement du kernel, par la variable `block_dim`.

CUDA impose que tous les threads d'un même bloc doivent résider sur un même processeur de l'accélérateur. Cette limitation simplifie l'ordonnancement des threads CUDA et permet à tous les threads d'un même bloc d'échanger rapidement des informations en utilisant de la mémoire dite partagée. Par conséquent, il y a une limite sur le nombre total de threads par bloc. Sur les deux cartes graphiques utilisées, cette limite est de 1024.

Pour utiliser plus de threads plusieurs blocs doivent être alloués. Exactement comme pour les threads, les blocs peuvent être déclarés selon 3 dimensions. Cet ensemble de blocs s'appelle une **grille**. Les dimensions de cette grille sont définies par la variable `grid_dim` lors de l'exécution. La division en 3 dimensions de ces deux niveaux n'est pas nécessaire, mais permet de rendre l'indexage plus naturel dans le cas d'un kernel utilisant des vecteurs, des matrices ou des volumes. Les blocs CUDA peuvent correspondre alors respectivement à un sous-vecteur, un bloc d'une matrice ou un cube contenu dans un volume.

La division en grille puis en bloc est résumée dans la figure 3.1 provenant du guide CUDA de NVIDIA :

3.2 Compilation CUDA

CUDA offrant des extensions au langage `C`, nous ne pouvons utiliser un compilateur classique comme `GCC` pour compiler un kernel CUDA. Nous devons utiliser à la place un compilateur reconnaissant ces extensions. NVIDIA fournit le compilateur `NVCC` lors de l'installation de la suite d'outils CUDA. Depuis la version 4.1 de CUDA, ce compilateur est basé sur l'infrastructure LLVM. `NVCC` ne remplace pas un compilateur standard pour le code CPU et n'émet donc pas d'assembleur `x86`. Les deux tâches effectuées par `NVCC` sont les suivantes :

- Compilation des kernels en code assembleur **PTX** ou binaire **cubin**.
- Remplacement des appels à chaque kernel par des appels au runtime CUDA pour configurer puis exécuter ce kernel.

Une fois cette étape réalisée, `NVCC` appelle un compilateur hôte comme `GCC` pour compiler les fonctions qui seront exécutées sur le CPU. Le PTX généré par `NVCC` est un code assembleur intermédiaire, ce code est compilé de manière **Just-In-Time** avant l'exécution d'un kernel. Cette stratégie permet d'avoir un binaire portable, les optimisations spécifiques à l'accélérateur utilisé sont appliquées durant cette phase de compilation.

3.3 Architecture

L'efficacité du modèle de programmation CUDA réside dans son architecture massivement parallèle et extensible. Un calcul à effectuer est divisé en **blocs** de threads lors de la mise en place de la configuration d'exécution. Ces blocs sont divisés et exécutés par ce que l'on appelle un **streaming multiprocessor (SM)**.

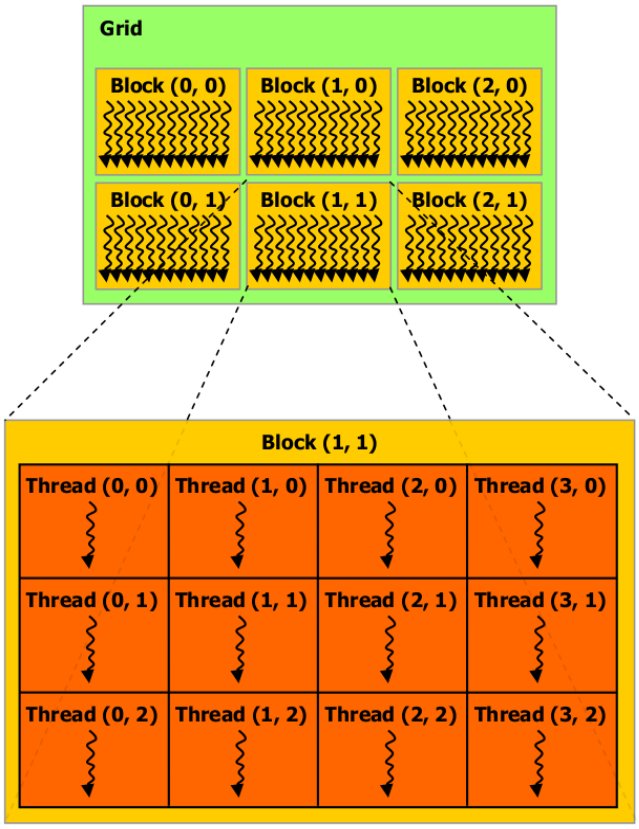


FIGURE 3.1 – Hiérarchie CUDA.

3.3.1 Multiprocesseur

Ces multiprocesseurs constituent la brique de base de l'exécution CUDA, l'extensibilité du modèle CUDA provient du fait que le nombre de multiprocesseurs peut être augmenté autant que nécessaire, le modèle d'exécution restera le même, mais les performances seront améliorées naturellement. Ce mécanisme permet à CUDA d'être un modèle **extensible** (on parle de **scalability**). Grâce à cette propriété, du code CUDA peut être lancé sans modifications sur une carte bas de gamme ou sur une carte haut de gamme spécialisée dans le calcul scientifique (comme une Tesla). Notons cependant que selon le **compute capability** de l'accélérateur, il sera peut-être nécessaire de réécrire partiellement le code.

Les threads d'un même bloc pouvant échanger des données via un espace mémoire appelé mémoire partagée (**shared memory**). Un bloc est donc une unité de parallélisme à grain plus élevé que le thread, mais offre un second niveau de parallélisation de l'algorithme puisque les threads d'un même bloc peuvent coopérer efficacement. Si ces blocs n'ont pas d'interactions entre eux, la **scalability** sera parfaite car toutes les exécutions seront indépendantes. Le modèle de blocs est donc le support de l'extensibilité CUDA.

3.3.2 Organisation d'un multiprocesseur

L'organisation interne d'un multiprocesseur sur une carte de génération **Fermi** (comme la GTX 560) est présentée sur le schéma 3.2. Ces informations proviennent du document NVIDIA (2009). Chaque cœur CUDA possède une unité arithmétique et logique (**Arithmetic Logic Unit, ALU**) et une unité de calcul flottant (**Floating Point Unit, FPU**). Au total, pour le multiprocesseur présenté, les composants suivants sont présents :

- **32 CUDA cores.**
- **16 Load/Store Unit.** Lecture et écriture des valeurs en RAM ou en cache.
- **4 Special Function Units**. Ces unités calculent des fonctions transcendantes, par exemple sinus, cosinus, racine carrée.
- **4 Texture Units**. Pour l'adressage et l'interpolation des textures en mémoire GPU.

Ces multiprocesseurs sont connectés au cache L2 puis à la mémoire globale de la carte (**DRAM**). L'accès à la DRAM est séparé en 6 partitions. Un composant d'interface relie le GPU au CPU par le bus PCI-Express. Le composant **GigaThread** est l'ordonnanceur global distribuant les blocs CUDA à exécuter entre les différents multiprocesseurs de la carte. Tant qu'il reste des blocs à exécuter, le moteur GigaThread alloue un ou plusieurs blocs par multiprocesseur. Cette technologie introduite dans la génération Fermi permet une meilleure distribution de la charge de travail et permet surtout à plusieurs kernels CUDA de s'exécuter de manière concurrente.

L'organisation générale de l'architecture Fermi est résumée sur la figure 3.3.

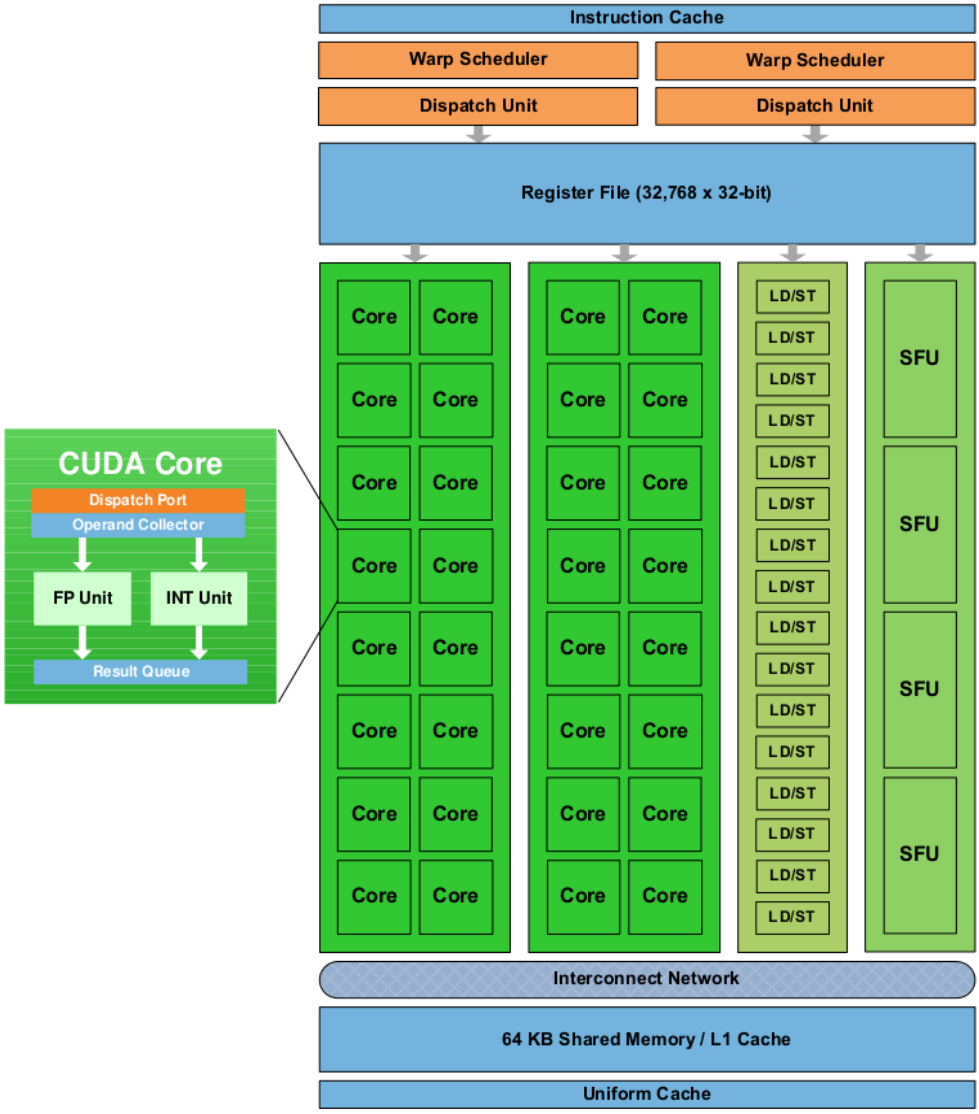


FIGURE 3.2 – Organisation de l'architecture Fermi.

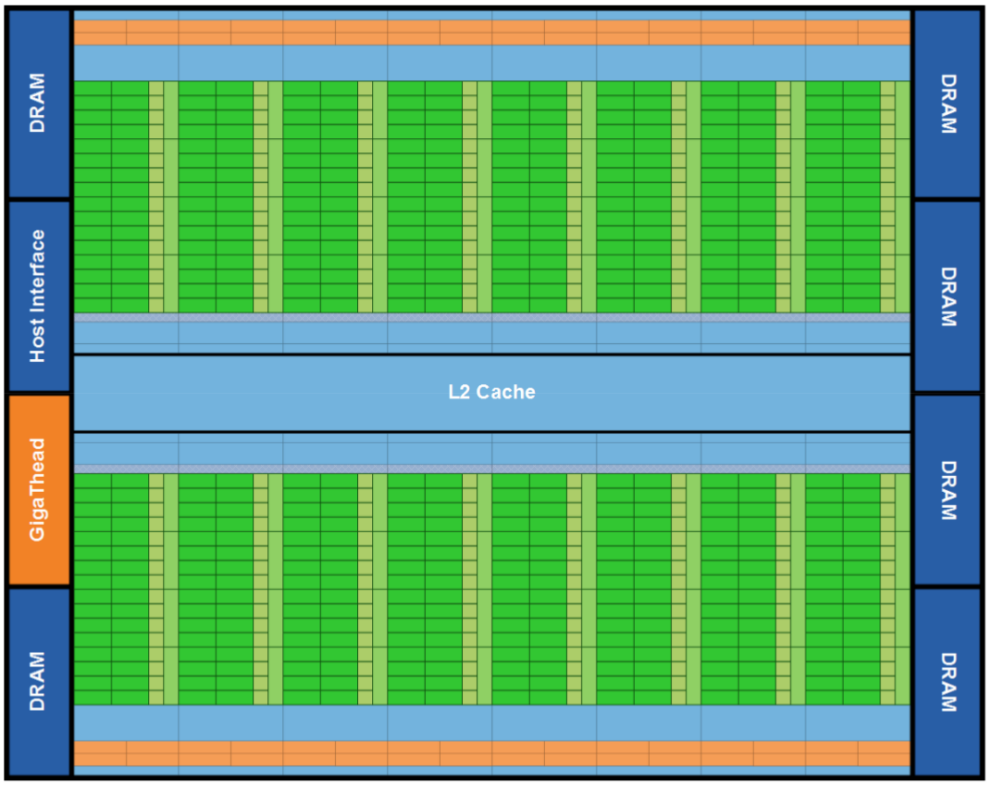


FIGURE 3.3 – Organisation globale de l'architecture Fermi.

3.3.3 Warp

L'architecture que nous avons présentée jusqu'à maintenant ne présente rien d'exceptionnel comparé à une architecture CPU classique. Le parallélisme est beaucoup plus important que sur CPU, mais chaque processeur GPU est beaucoup moins performant. La notion de warp est un élément central des performances CUDA, nous allons donc en expliquer le principe.

Un **w**arp est un ensemble de 32 threads consécutifs (par rapport à leur indice unique). Pour le moteur GigaThread, un bloc est l'unité élémentaire à ordonnancer et exécuter. Le principe est le même à ce niveau : le warp est l'unité élémentaire d'ordonnancement au sein d'un multiprocesseur. Nous avons donc 2 niveaux de parallélisme à des granularités différentes et régulés chacun par des ordonnanceurs. Comme nous pouvons le remarquer dans la figure 3.2, l'architecture utilisée possède 2 **Warp scheduler** et 2 **Dispatch Unit**. À chaque cycle (**tick**) d'horloge, 2 warps peuvent chacun exécuter une instruction simultanément.

Ordonnancement

Après exécution, certains warps peuvent être mis en attente. Par exemple lorsque l'instruction utilise une unité de calcul spéciale citée au-dessus (accès à une texture, accès à la DRAM, utilisation de la SFU), le warp doit attendre de recevoir le résultat. Au prochain cycle d'horloge, ce sera donc un autre warp qui sera exécuté. À un instant t, les warps pouvant être exécutés sont dits **actif**, les autres sont **en attente**. Un exemple d'exécution avec 2 **Warp Schedulers** est présenté dans la figure 3.4.

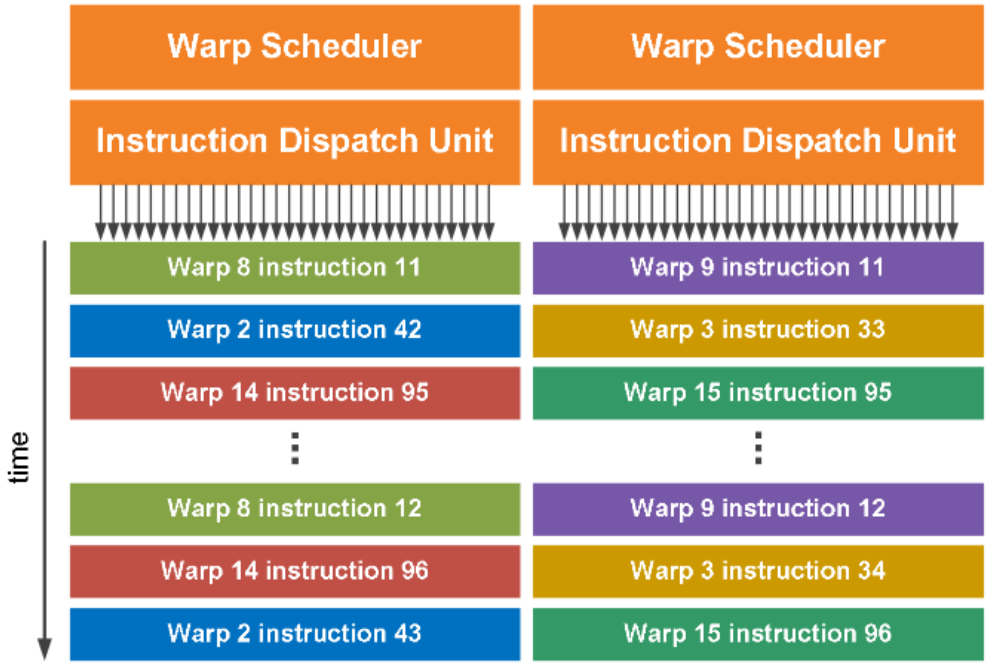


FIGURE 3.4 – Ordonnancement de warps avec deux **Warp Schedulers**.

Architecture SIMT

Pour qualifier cette architecture où un ensemble de threads exécutent la même instruction simultanément, on parle d'architecture **Single Instruction, Multiple Thread** (SIMT). Cette qualification est une extension de la **taxinomie de Flynn** qui introduit les instructions **Single Instruction, Multiple Data**. Les instructions SSE sont un exemple d'instructions SIMD.

Divergence de warps

Chaque thread possède cependant son propre **program counter** et ses propres registres. Les threads sont donc libres de prendre des chemins d'exécutions différents. Cependant, le mécanisme de warp implique que les performances maximales sont atteintes uniquement lorsque les 32 threads d'un même warp prennent le même chemin d'exécution, la même instruction est alors exécutée par tous les threads.

Si plusieurs threads ne suivent pas le même chemin d'exécution que les autres (par exemple suite à un `if` ou à une boucle `for`) , on parle de **divergence** de warp. S'il y a divergence, chaque branche doit être exécutée tour à tour, les performances sont donc divisées par 2. **Pour obtenir de bonnes performances, la divergence est un facteur important à prendre en compte.**

3.3.4 Compute Capability

Le **compute capability** est la capacité de calcul CUDA d'un accélérateur. Selon la version de l'accélérateur utilisé, certaines fonctions CUDA, comme les opérations atomiques, ne seront pas disponibles. D'autres changements sont purement architecturaux, mais peuvent grandement influencer les performances d'exécution. Par exemple au-dessous

de la version 2.0, il n'y a pas de cache sur la DRAM, tous les accès sont extrêmement couteux. Avant la version 2.0 il n'y a pas non plus d'opération de **Fused Multiply-Add** 64-bit et l'arithmétique en nombres entiers est effectuée sur 24 bits.

3.4 Mémoire

Nous avons expliqué comment fonctionnent la répartition et l'ordonnancement des tâches, il s'agit du premier pilier à la base des performances de CUDA. Le second pilier est la hiérarchie mémoire particulière présente sur la carte graphique. Plusieurs espaces mémoire sont disponibles pour le développeur, connaître leurs avantages et leurs limitations est important afin d'améliorer les performances. Nous pouvons classifier ces mémoires en 2 types : la mémoire résidant sur le multiprocesseur (**on-chip**), et la mémoire résidant sur l'accélérateur (**on-board**). Le premier type de mémoire étant dans le même circuit, la bande passante de transfert est nettement plus rapide.

Nous allons présenter brièvement les particularités de chacun de ces espaces mémoire.

3.4.1 Registres

C'est la mémoire la plus rapide disponible. Le nombre de cycles d'horloge nécessaires pour y accéder est inférieur à 1. La contrepartie est que c'est une ressource extrêmement limitée, La taille du banc de registres (**register file**) est de 32 Ko par multiprocesseur. Ces registres sont partagés entre tous les threads des blocs résidants actuellement sur le multiprocesseur. Il existe une limite du nombre de registres par thread de 63 pour les accélérateurs actuels. Si le nombre de registres par threads augmente alors le nombre maximum de threads pouvant résider sur un multiprocesseur diminue. Augmenter le nombre de registres peut ainsi diminuer les performances puisque moins de warps seront potentiellement exécutables simultanément, mais cela peut aussi permettre d'améliorer les performances en profitant de la bande passante très élevée des registres.

3.4.2 Mémoire partagée

La mémoire partagée est un espace de 16 ou 48 Ko présent sur chaque multiprocesseur et organisé en 32 **banques** de 32 bits de large chacune. Lorsque plusieurs threads d'un même warp accèdent à la même banque, il y a un conflit et les accès doivent être sérialisés. Une variable ou un tableau en mémoire partagée peut être déclaré en rajoutant l'attribut `__shared__`. On ne peut déclarer qu'un seul tableau en mémoire partagée par kernel. La taille du tableau peut être choisie statiquement ou dynamiquement.

3.4.3 Cache L1

Chaque multiprocesseur possède son cache L1. Il s'agit du même espace physique que la mémoire partagée. Cet espace a pour taille 64 Ko et peut être partitionné pour soit favoriser le cache soit la mémoire partagée (16/48 ou 48/16). Le cache L1 a une latence d'accès de 10 à 20 cycles d'horloge. Le cache L1 stocke aussi la pile d'exécution (pour les appels de fonction et les variables locales). Ce cache est uniquement utilisé pour optimiser les accès en lecture, les écritures en mémoire globale passent directement au cache L2.

3.4.4 Mémoire locale

La mémoire locale n'a pas d'existence physique, il s'agit d'un espace virtuel permettant de stocker les variables ne pouvant être stockées sur le banc de registres. Lorsqu'un kernel devrait utiliser plus de registres que la capacité d'un multiprocesseur, certaines variables sont placées en mémoire locale à la place. Depuis l'architecture Fermi, la mémoire locale est située sur le cache L1. Avant cela la mémoire locale résidait en mémoire globale, provoquant une pénalité potentielle de plusieurs centaines de cycles lors d'un accès à une variable de cet espace.

3.4.5 Cache L2

Le cache L2 est commun à tous les multiprocesseurs, sa taille est de 512 Ko pour l'architecture Fermi. Ce cache suit une stratégie **Least Recently Used**, contrairement au cache L1. Les données récemment utilisées y sont conservées. Ce cache est garanti comme étant toujours cohérent pour tous les multiprocesseurs. Les threads peuvent donc modifier une variable en cache L2, ce changement sera immédiatement visible pour tous les autres threads. Les opérations atomiques peuvent s'effectuer directement en cache L2 et n'invalident pas le cache.

3.4.6 Mémoire globale

Physiquement, il s'agit de la mémoire DRAM de l'accélérateur. C'est la mémoire la plus large, par exemple 2 Go sont disponibles sur la Quadro 4000. En contrepartie, c'est la mémoire ayant la latence d'accès la plus lente : plusieurs centaines de cycles. Les accès à la mémoire globale sont très pénalisants, mais il existe différents moyens d'optimiser ces accès :

- **Alignement :** les accès mémoire sont moins efficaces lorsque l'adresse n'est pas alignée. Pour forcer l'alignement, il peut être nécessaire d'introduire du padding dans certaines structures de données (des octets inutilisés sont rajoutés pour forcer l'alignement).
- **Coalescing** : lorsque des threads consécutifs d'un warp accèdent à des données consécutives en mémoire, ces accès peuvent être rassemblés pour diminuer le nombre de transactions mémoires.
- **Broadcast** : lorsque tous les threads d'un même warp veulent accéder à un même segment mémoire de taille 128 bits. Une seule transaction mémoire de 128 bits sera effectuée au lieu de 32 transactions, les données seront ensuite propagées à tous les warps. L'efficacité de cette opération est donc de 3200% !

De nombreuses techniques peuvent être utilisées pour minimiser ou optimiser les accès à la mémoire globale. Ces techniques sont présentées dans Farber (2011).

Pour le développeur, des segments de mémoire globale peuvent être alloués dynamiquement en utilisant la fonction `cudaMalloc`. Ces segments sont ensuite libérés par des appels à `cudaFree`.

3.4.7 Mémoire constante

C'est un espace mémoire de 64 Ko qui, comme son nom l'indique, ne peut pas être modifié. Pour les accélérateurs de version 2.0 et supérieur, l'intérêt de la mémoire constante est diminué à cause de l'introduction des caches et de l'instruction **LoaD Uniform (LDU)**.

3.4.8 Mémoire texture

La mémoire texture est un espace situé en mémoire globale. Chaque multiprocesseur possède 8 Ko de cache sur cette mémoire texture. Cette mémoire est optimisée pour des accès consécutifs proches spatialement. Des capacités additionnelles de filtrage sont disponibles, par exemple : interpolation 9-bit, **clamping** ou **wrapping** des coordonnées. Le principal désavantage de cette mémoire est l'absence de cohérence : une écriture d'une valeur de la texture n'est pas garantie d'être visible à tous les autres threads du même kernel.

3.4.9 Résumé

Le schéma 3.5 résume l'emplacement de chacun des espaces mémoire que nous avons décrit jusqu'à présent.

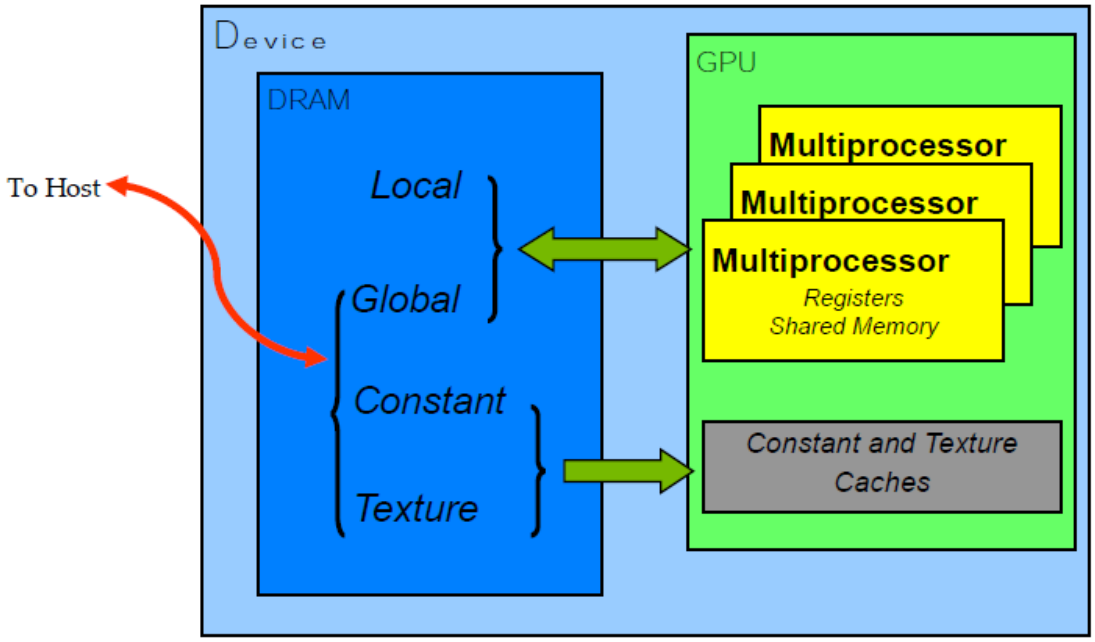


FIGURE 3.5 – Espaces mémoire CUDA.

Chapitre 4

Watchdog et timeout

4.1 Présentation

Un **watchdog**, ou **chien de garde** est un mécanisme matériel ou logiciel permettant de déclencher le redémarrage d'un système dans le cas où le temps d'exécution d'un programme dépasse une certaine limite. Tous les pilotes Nvidia incluent un watchdog qui est uniquement actif sur la carte graphique responsable de l'affichage vidéo (par exemple la carte graphique utilisée par le **serveur X**.

4.2 Intêrets

Un watchdog présente à la fois des avantages et des inconvénients.

L'avantage est que si un programme est bloqué à cause d'une erreur de programmation (par exemple une boucle infinie dans un kernel CUDA) il sera arrêté par le watchdog. Sans un watchdog, l'affichage resterait figé et seul un redémarrage complet du système pourrait résoudre le problème, cette procédure serait lente et donc non acceptable pour un environnement de développement. Nous pouvons noter au passage que le watchdog CUDA ne surveille que le temps d'exécution des kernels lancés, au contraire le mécanisme de protection mémoire sur la carte graphique n'est pas parfait, un redémarrage est parfois nécessaire après un dépassement mémoire dans un kernel.

L'inconvénient du watchdog est que s'il n'y a pas erreur de programmation, le watchdog se déclenchera de toute façon. Le timer étant de 10 secondes sur Linux, un calcul CUDA ne peut pas s'exécuter plus longtemps. Dans le cadre du calcul haute performance, ce comportement n'est pas acceptable pour une mise en production puisque 10 secondes est une durée extrêmement courte au vu de la complexité des calculs parfois réalisés.

4.3 Solution

Il existe plusieurs solutions au problème du déclenchement du watchdog CUDA. Nous en avons identifié 4 et nous allons présenter leurs avantages et leurs inconvénients dans cette partie.

4.3.1 Partitionnement statique

Une première possibilité est de partitionner l'ensemble des données à traiter en N sous-ensembles et de lancer N kernels CUDA au lieu d' un seul traitant l'ensemble des données.

La partition des données est effectuée statiquement, c'est-à-dire que la taille de chaque ensemble est fixée avant le lancement des N kernels. Sachant que lancer N kernels induit une surcharge de travail au niveau de l'ordonnanceur CUDA, la principale difficulté de cette approche réside dans le fait de réussir à trouver la bonne granularité de partition. Une granularité trop fine évitera assurément le déclenchement du watchdog mais le temps d'exécution total sera plus long. Au contraire, une granularité plus forte expose au risque de **timeout** mais le temps d'exécution ne serait pas augmenté.

Afin de s'adapter à la carte graphique utilisée, la taille des partitions peut être choisie en utilisant une heuristique dépendant des caractéristiques CUDA de l'accélérateur. Cependant ce choix ne peut pas assurer que le watchdog ne sera pas déclenché puisque le temps de calcul nécessaire pour chaque partition peut présenter une variance importante.

4.3.2 Partitionnement dynamique

Cette approche améliore la précédente en effectuant le partitionnement dynamiquement : après l'exécution d'un kernel, son temps d'exécution a un effet rétroactif (**feedback**) sur le partitionnement : si le timeout était trop proche la prochaine partition sera plus petite, dans le cas contraire elle sera plus grande.

Encore une fois cette approche ne peut pas garantir qu'il n'y aura pas de timeout. Si le watchdog se déclenche, 10 secondes sont alors perdues et il faut réinitialiser l'API CUDA, ce qui prend un temps non négligeable. De plus, cette approche est plus complexe à mettre en place et demande une bonne heuristique pour contrôler dynamiquement la partition des données.

4.3.3 Mode TTY

Comme nous l'avons expliqué ci-dessus, le watchdog n'est actif que pour les cartes graphiques utilisées par le serveur graphique. Sous Linux il est donc possible de désactiver le timeout en utilisant la station de travail en mode **TTY**. Cette solution n'est pas applicable pour Mac OS X et Windows. Pour cela nous devons basculer sur une fenêtre TTY en utilisant la combinaison de touches `CTRL+ALT+F1`, serveur graphique est ensuite arrêté en utilisant la commande `stop` suivie du nom du desktop manager :

```
$ sudo stop kdm
```

La session graphique étant maintenant fermée, nous pouvons lancer un kernel CUDA sans limites de temps depuis ce TTY. Le principal inconvénient de cette méthode est qu'il n'est plus possible d'utiliser OpenGL pour visualiser les résultats. Cette méthode est utilisée dans le projet lorsqu'un seul GPU est présent afin de lancer une série de tests fonctionnels ou de tests de performances.

4.3.4 Configuration double GPU

Si la visualisation est indispensable en même temps que le lancement d'un kernel CUDA (c'est le cas pour projet SIMSURF), alors la solution restante est d'utiliser une configuration comportant 2 cartes graphiques. Une des cartes (la moins puissante) est responsable de l'affichage vidéo et est donc utilisée par le serveur graphique. Cette carte est utilisée pour la partie visualisation de l'application en exécutant les appels de l'API OpenGL.

La deuxième carte graphique est utilisée uniquement pour les appels de kernels CUDA et n'est donc pas connectée à une entrée vidéo. Cette carte est donc considérée comme un accélérateur de calcul pur et non une carte graphique. Par exemple, les cartes NVIDIA de la gamme Tesla sont uniquement dédiées au calcul haute performance CUDA (en production dans un cluster par exemple). Ces cartes n'ont aucune sortie vidéo.

Au vu des désavantages des 3 précédentes méthodes, c'est celle-ci qui a été retenue durant le stage pour éviter l'intervention du watchdog. La configuration utilisée à l'ENS Cachan comporte 2 GPU : une Quadro FX 1800 pour gérer l'affichage et une Quadro 4000 pour le lancement des kernels CUDA.

Chapitre 5

Algorithme de simulation

5.1 Principe

Nous cherchons à réaliser la simulation numérique de l'usinage d'une pièce par une machine-outil. En situation réelle, pour réaliser un usinage nous avons besoin des éléments suivants :

- **Surface initiale :** la pièce est usinée par enlèvement de matière à partir d'un bloc brut ou d'une pièce issue d'un autre usinage.
- **Un outil :** la forme de l'outil influence la façon dont le bloc de matière sera creusé.
- **Une trajectoire :** afin d'obtenir les formes de la pièce voulue, l'outil doit suivre une trajectoire afin d'enlever la matière aux endroits souhaités.
  La trajectoire peut être de différents types :
  - 3 axes : l'outil se déplace dans l'espace selon les coordonnées (x, y, z). Son orientation reste inchangée.
  - 5 axes : l'outil se déplace dans l'espace selon les coordonnées (x, y, z). Son orientation peut varier à chaque position, l'orientation de l'outil est donnée par un vecteur directeur normalisé (i, j, k).

Si l'outil n'est pas symétrique autour de son axe, on doit prendre en compte la rotation de la broche. La valeur de la rotation angulaire à une position donnée sera notée ω. Plusieurs configurations outil/trajectoire peuvent se succéder pour usiner une seule pièce, par exemple on peut avoir une phase d'ébauche avec un premier outil puis une phase de finition avec un outil différent.

Afin de réaliser une simulation de ce processus, nous devons modéliser chacun de ces éléments afin d'obtenir une représentation manipulable numériquement :

- **Surface initiale :** elle est définie par ses dimensions en largeur, longueur et hauteur. Afin de représenter les variations de hauteur, il est nécessaire de discrétiser la surface. Seules les grilles régulières sont supportées dans le projet.
- **Outil :** pour que la simulation soit la plus précise possible, nous devons avoir une représentation pouvant tenir compte de toutes les particularités de l'outil de façon très fine. Nous avons donc choisi de représenter l'outil par un maillage de triangles.
- **Trajectoire :** la trajectoire de l'outil est discretisée afin d'obtenir des positions successives. La discrétisation des positions est effectuée en amont par un autre programme. Un positionnement 3 axes est représenté par un triplet des 3 coordonnées (x, y, z). La représentation d'un positionnement 5 axes nécessite de stocker également l'orientation de la normale de l'outil : un autre triplet (i, j, k) est nécessaire.

5.2 Données discretisées

Nous avons donc l'ensemble des données suivantes :

- Une grille de taille $N \times M$ représentant une discrétisation de la surface initiale. L'origine de la grille est placée à un point (gx, gy, gz). La grille étant régulière l'espacement entre points voisins est constante : d.
- Un maillage de T triangles représentant l'outil. Ce maillage peut être représenté en géométrie indexée ou par une liste de triplets de sommets.
- Un ensemble de P positions outils. Une position est représentée par un 3-uplet dans le cas d'une trajectoire 3 axes et par un 6-uplet dans le cas d'une trajectoire 5 axes. Si l'angle doit être pris en compte alors une valeur supplémentaire doit être rajoutée par position.

Une fois en possession de ces données numériques modélisant notre configuration d'usinage, nous devons choisir un algorithme permettant de représenter l'enlèvement de matière. De nombreux algorithmes existent pour simuler l'enlèvement de matière sur une surface, ces algorithmes ont des complexités et des précisions variables. Nous allons présenter deux algorithmes classiques dans le domaine : l'algorithme **N-buffer** et l'algorithme **Z-buffer**. Pour le projet SIMSURF nous avons choisi d'implémenter l'algorithme **Z-buffer**.

5.3 N-buffer

À chaque point de la grille est associé un segment orienté parallèlement au plan de la grille, ce segment est appelé **brin**. La hauteur du segment correspond à la hauteur de la surface à cette position. L'ensemble des brins forme ce que l'on appelle un **hérisson**. L'algorithme **N-buffer** utilise une technique de lancer de rayons pour simuler l'enlèvement de matière sur la surface : à chaque brin on associe un rayon ayant pour origine le sommet du brin et comme orientation la normale à la surface au point considéré. La hauteur de chaque brin est stockée dans un tampon (**buffer**) et à chaque brin est associé un vecteur suivant la normale, d'où le nom d'algorithme **N-buffer**.

Si un rayon réalise une intersection avec un triangle et que la hauteur de l'intersection est inférieure à la hauteur actuelle du brin, on parle de coupe ou de tonte du brin. La grille est alors mise à jour avec la nouvelle hauteur. Cette coupe correspond symboliquement à l'enlèvement de matière lorsqu'une partie de l'outil se trouve au-dessous de la surface initiale. Le principe de l'algorithme est présenté dans la figure 5.1, provenant de l'article original présentant cette méthode : Jerard *et al.* (1989).

Afin de simuler l'usinage complet de la pièce nous devons calculer pour chaque brin son intersection avec tous les triangles du maillage outil et dans toutes les positions. La complexité est donc :

$$N \times M \times T \times P \tag{5.1}$$

De nombreuses optimisations sont implémentables pour ne pas avoir à calculer l'intersection de chaque brin avec tous les triangles dans toutes les positions outil. Les techniques d'optimisation appliquées en ray tracing peuvent être envisagées.

5.3.1 Ray tracing

Le ray tracing est un algorithme classique de synthèse d'image utilisé pour obtenir des rendus photoréalistes par exemple simulant la réfraction et la réflexion de la lumière sur différents matériaux et dans différents milieux. Dans notre cas nous pouvons adopter une

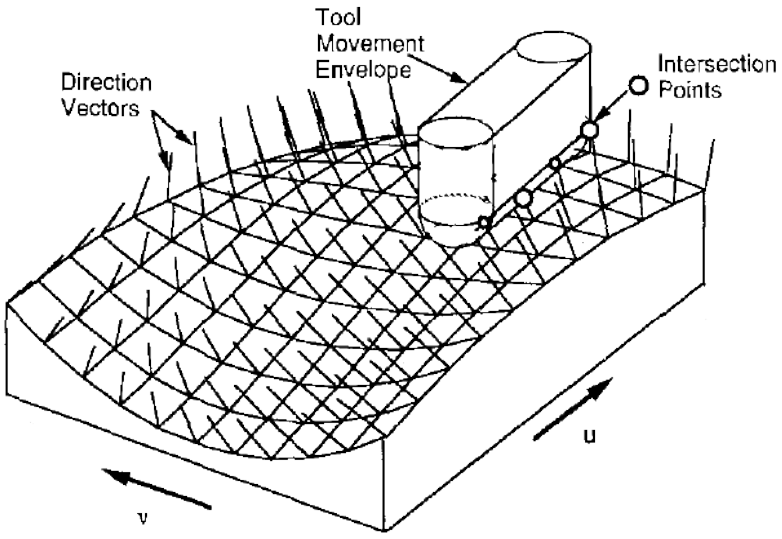


FIGURE 5.1 – Algorithme N-buffer

approche raytracing car un rayon est lancé pour chaque brin. Cette technique peut être accélérée en diminuant le nombre de tests par exemple en calculant d'abord l'intersection avec la boîte englobante de l'outil et en utilisant des structures de données adaptées comme une Bounding Volume Hierarchy (BVH) ou un arbre BSP (Binary Space Partitioning).

5.4 Z-buffer

L'algorithme **Z-buffer** est une variation de l'algorithme **N-buffer**. Le nom de cet algorithme fait référence à la technique utilisée en synthèse d'image pour déterminer la visibilité des éléments de la scène. L'algorithme du Z-buffer est appliqué en utilisant le hérisson au lieu de la fenêtre d'affichage.

Dans le cas du Z-buffer, tous les rayons sont orientés dans la même direction, parallèlement à la surface considérée (voir figure 5.2). Cette méthode est moins précise que la précédente, car on perd des informations sur l'allure de la surface, mais elle permet de nombreuses optimisations à la fois sur le nombre de tests d'intersection à effectuer et sur le nombre d'opérations nécessaires pour chaque test.

En utilisant les hypothèses supplémentaires de cette méthode, l'algorithme diminue pour chaque triangle le nombre de brins testés. Pour chaque triangle, sa projection sur la grille et boîte englobante 2D sont calculées comme illustré sur la figure 5.3 Les brins en dehors de la boite englobante (les cercles hachurés) ne sont pas testés, ils ne peuvent pas avoir d'intersection avec ce triangle. Une double boucle est ensuite effectuée pour tester l'intersection de chaque brin à l'intérieur de la boite englobante avec la projection 2D du triangle. Cette boîte englobante du triangle sera appelée **fragment** en référence à une technique similaire utilisée en synthèse d'images. Les brins peuvent avoir une intersection avec le triangle (en vert) ou non (en rouge). S'il y a intersection, la hauteur est calculée puis comparée avec la hauteur actuelle de ce brin. Plusieurs algorithmes d'intersection peuvent être utilisés, dans ce rapport nous présentons les différents algorithmes que nous avons successivement utilisés.

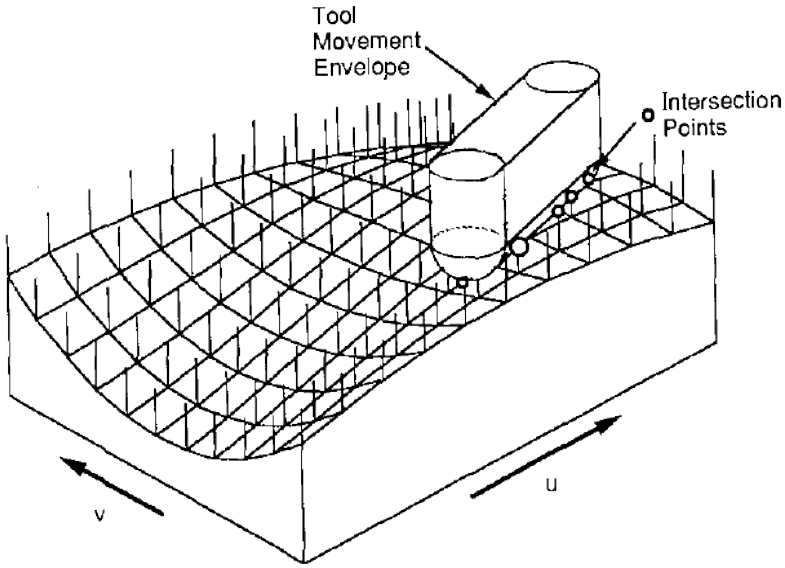


FIGURE 5.2 – Algorithme Z-buffer

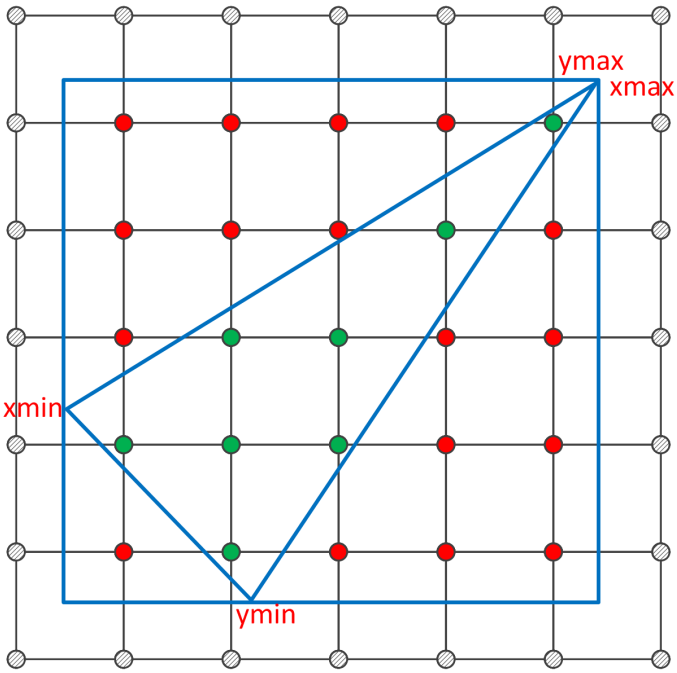


FIGURE 5.3 – Boite englobante 2D d'un triangle.

5.4.1 Pseudo-code

Le pseudo-code C++ de l'algorithme Z-buffer est le suivant :

```
void zbuffer(Path& path, Surface& surface, Tool& tool)
{
  // For each position.
  for (int p = 0; p < path.size(); ++p)
  {
    position pos = path[p];
    // For each triangle
    for (int t = 0; t < tool.size(); ++t)
    {
      triangle tri = tool[t];
      // Transform triangle: apply translation, rotation, transform
          matrix...
      transform_triangle(tri, pos);

      // Project triangle on surface and get fragment.
      domain2D domain = project(tri, surface);
      for (int y = domain.ymin(); y < domain.ymax(); ++y)
      {
        for (int x = domain.xmin(); x < domain.xmax(); ++x)
        {
          // Set up the ray.
          float3 origin = { x, y, surface(x, y) };
          float3 direction = { 0.f, 0.f, 1.f };

          float distance; // will store the distance if intersection.
          if (intersection(origin, direction, tri, distance))
            surface(x, y) = min(surface(x, y), distance);
        }
      }
    }
  }
}
```

5.5 Occupation mémoire

Nous allons maintenant étudier l'occupation mémoire sur la carte graphique de la simulation en fonction de différents paramètres comme la précision utilisée, la taille de la grille, la taille du maillage outil, le type d'usinage et le nombre de positions. Nous allons tout d'abord donner les formules permettant de calculer l'occupation mémoire (en octet) pour chaque paramètre pris indépendamment. Puis nous étudierons comment se combinent ces paramètres dans deux contextes différents de simulation : dans un contexte de macrogéométrie et un contexte de microgéométrie.

5.5.1 Précision

Pour stocker la hauteur des brins, les positions et le maillage de l'outil, nous pouvons choisir d'utiliser une précision sur 32 ou 64 bits (**float** ou **double**). Nous considérerons ici que pour une configuration donnée la précision utilisée est la même pour tous les paramètres. Nous dénoterons par S la taille d'une variable en virgule flottante.

- $S = 4$ en cas de simple précision.
- $S = 8$ en cas de double précision.

Sous CUDA et sur CPU il n'existe pas nativement d'autres formats (par exemple 16 ou 128 bits).

5.5.2 Grille

Nous considérons dans notre cas une grille carrée avec un espacement régulier des brins. Nous appellerons N la taille d'un coté de la grille. L'occupation mémoire d'une grille de taille N est calculée par l'équation suivante :

$$M_{grid} = S \times N^2 \tag{5.2}$$

5.5.3 Maillage outil

Il existe deux manières de stocker un maillage triangulaire : par une liste de triangles ou par une liste de sommets et une liste d'indices (géométrie indexée).

Liste de triangles

En représentant chaque triangle séparément, il est nécessaire de stocker les coordonnées 3D de chacun des 3 sommets. Un sommet est stocké sous la forme d'un triplet (x, y, z) soit 3 valeurs de type simple ou double précision. Certaines données sur le triangle peuvent être précalculées afin d'accélérer le temps de calcul de l'intersection, on peut aussi rajouter du padding sur la structure afin d'aligner les accès mémoire. On rajoute donc un facteur de padding P. L'occupation mémoire d'un triangle est donc

$$M = 3 \times 3 \times S + P \tag{5.3}$$

Si le maillage est constitué de T triangles, l'occupation mémoire totale est alors :

$$M_{mesh} = N \times M \tag{5.4}$$

Dans le cas d'un triangle simple précision et sans padding nous avons donc une taille mémoire de 36 octets.

Géométrie indexée

En géométrie indexée chaque nouveau triangle nécessite d'ajouter un sommet à la liste ainsi que 3 indices permettant d'identifier les sommets constituant le nouveau triangle. Pour T triangles on a donc $3 + (T - 1)$ sommets de type virgule flottant et $3 \times T$ indices de type entier. La taille d'un sommet est $3 \times S$. La taille d'un indice sera la taille d'une variable de type `int` soit généralement 32 bits (4 octets). Si le nombre de triangles est inférieur à 2^{16} on pourrait stocker les indices sur des variables de type `unsigned short` pour compresser la liste des indices. La taille mémoire du maillage est donc :

$$M_{mesh} = 2 + T + 3 \times S \times T \tag{5.5}$$

5.5.4 Positions outil

La taille de chaque position dépend du type d'usinage utilisé dans la simulation. On appellera p le nombre d'octets nécessaires pour représenter une seule position. Pour représenter l'ensemble des positions, le nombre d'octets requis est :

$$M_{path} = p \times P \tag{5.6}$$

3 axes

Une position 3 axes est représentée par un triplet (x, y, z). On a donc $p = 3 \times S$.

3 axes avec rotation

Une position 3 axes avec rotation est représentée par un triplet (x, y, z) et une valeur ω de type virgule flottante représentant la valeur angulaire. $p = 4 \times S$.

5 axes

Dans le cas d'une configuration 5 axes on rajoute un nouveau triplet (i, j, k) pour représenter le nouvel axe de l'outil. Ainsi : $p = 6 \times S$.

5 axes avec rotation

On a une valeur supplémentaire pour stocker l'angle de rotation. $p = 7 \times S$.

5.5.5 Contexte macrogéométrique

Dans un contexte de macrogéométrie, on cherche à déterminer l'allure générale de la surface usinée. La taille du maillage outil est donc faible car avec un maillage trop fin de nombreux triangles seront projetés entre les brins. Nous nous intéressons alors à l'occupation mémoire totale en fonction de la taille de la grille et du nombre de positions outil. Le graphique 5.4 illustre la courbe des isovaleurs de l'occupation mémoire (en Mo). Nous avons choisi une configuration de trajectoire 5 axes avec rotation au format simple précision, soit 28 octets par position outil.

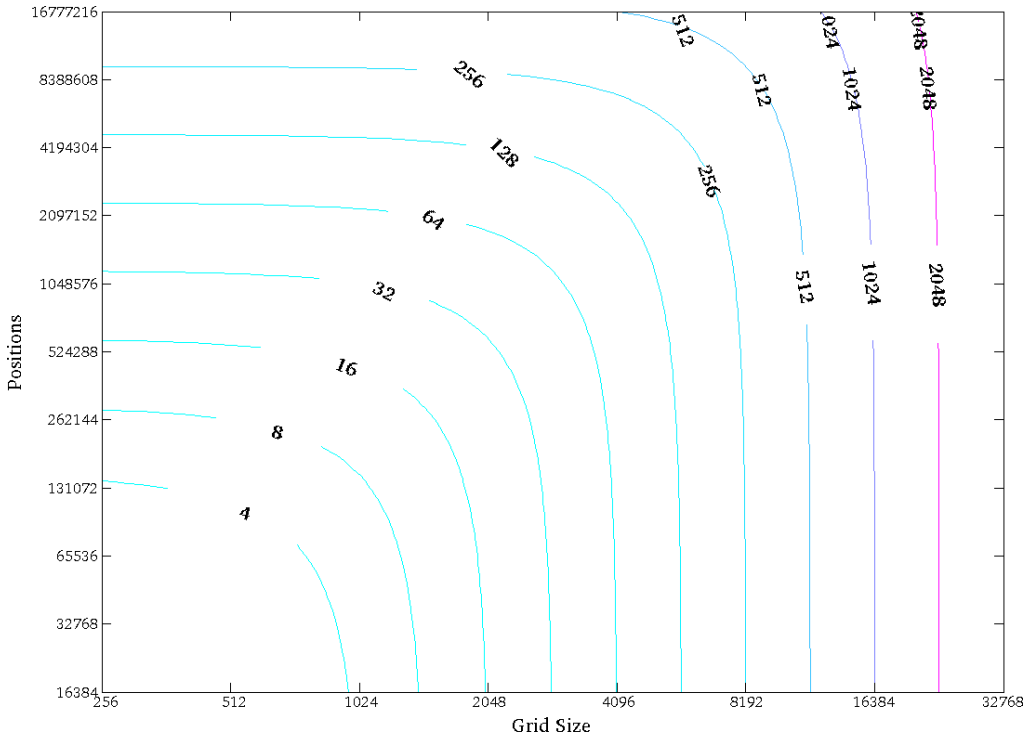


FIGURE 5.4 – Courbes d'occupation mémoire en contexte macrogéométrique.

5.5.6 Contexte microgéométrique

Dans un contexte de microgéométrie, on effectue les calculs sur une surface de taille réduite : le nombre de positions outil est peu élevé. On veut avoir la représentation la plus précise possible de la surface, on travaille alors avec un maillage très fin de l'outil. Dans cette configuration nous nous intéressons plutôt à l'occupation mémoire en fonction de la taille de la grille et de la taille du maillage outil. Nous avons tracé les isovaleurs de l'occupation mémoire dans le graphique 5.5. Le maillage est stocké sous la forme d'une liste de triangles, soit 36 octets par triangle. Nous avons fixé une borne sur le nombre de triangles à 1 million.

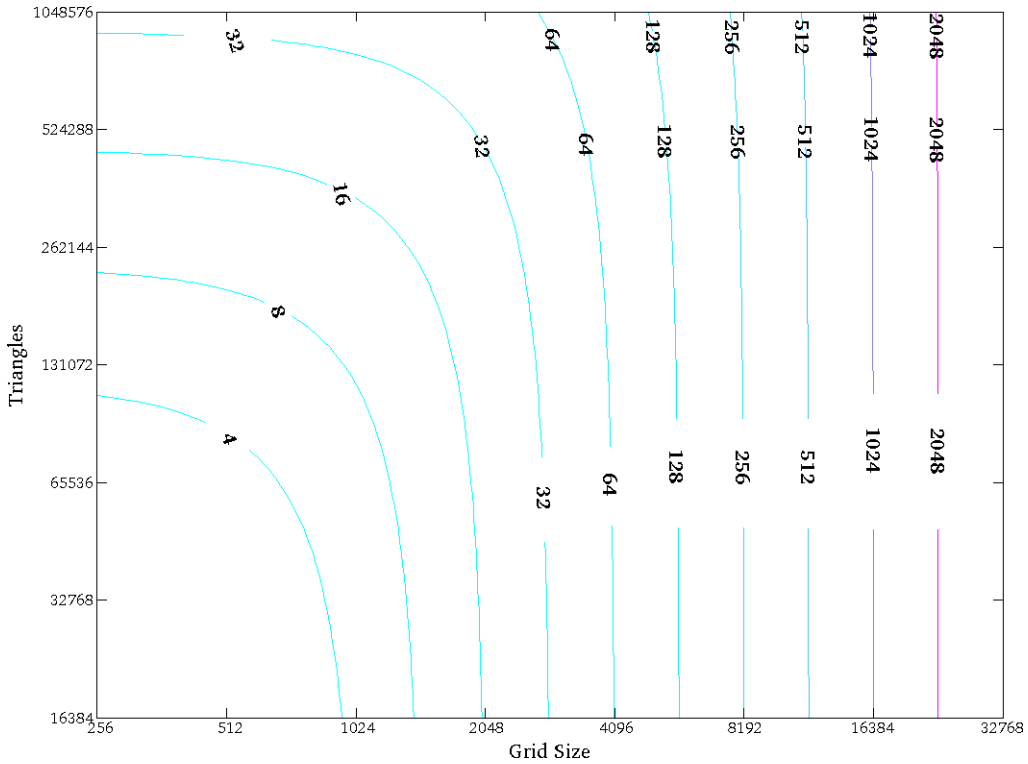


FIGURE 5.5 – Courbes d'occupation mémoire en contexte microgéométrique.

Chapitre 6

Parallélisation

Dans ce chapitre nous allons présenter comment les fonctionnalités de CUDA ont été utilisées pour paralléliser le calcul de la simulation dans le projet. Nous allons présenter différentes stratégies de parallélisation.

6.1 Synchronisation et conflit d'accès

6.1.1 Problème

En calcul parallèle, toute synchronisation implique une perte de parallélisme donc une dégradation des performances de l'application. Selon l'approche utilisée, lorsque plusieurs threads exécutent en parallèle l'algorithme Z-buffer il peut être nécessaire d'éviter les conflits d'accès lors de la mise à jour de la hauteur des brins. Nous avons pour cela deux possibilités :

- Utiliser des opérations atomiques. Tous les threads travaillent sur les mêmes brins en mémoire et leur mise à jour se fait de manière concurrente. En utilisant des opérations atomiques, l'état de la mémoire sera cohérent et déterministe après exécution. Les opérations atomiques ont un coût plus important que les accès mémoire habituels. Les problématiques liées aux opérations atomiques et l'impact sur les performances sont détaillées dans un autre chapitre.
- Utiliser du stockage propre à chaque thread. Chaque thread travaille sur une copie locale de la surface. Si la hauteur d'un brin change, uniquement la copie est mise à jour. Une fois toutes les exécutions terminées on réalise la fusion de toutes les copies locales (en calculant brin par brin le minimum entre toutes les copies locales) pour obtenir la surface finale.

6.1.2 CPU

Les deux stratégies sont applicables indifféremment pour une implémentation CPU. La deuxième stratégie requiert plus de mémoire, mais ce n'est pas un problème sous CPU avec les 8Go disponibles et grâce aux trois niveaux de cache (L1, L2, L3) disponibles sur le processeur Xeon utilisé. On peut tirer parti des instructions SSE pour réaliser la fusion entre les différentes surfaces.

6.1.3 GPU

Sur GPU des millions de threads sont lancés simultanément, la deuxième stratégie est donc inapplicable telle quelle puisque cela nécessiterait autant de copies de la surface

qu'il y a de threads. De plus, l'API CUDA propose nativement une instruction permettant d'effectuer une opération de minimum atomique en mémoire globale ou locale. La première stratégie est donc à privilégier sur le GPU.

6.2 Stratégie de parallélisation CPU

En utilisant l'API OpenMP la parallélisation peut se faire de façon très simple à l'aide d'une directive de compilation. L'avantage d'OpenMP est que cette API est supportée par les principaux compilateurs C et C++ existants, ainsi aucune dépendance externe à une bibliothèque n'est ajoutée. Le désavantage est que les possibilités de parallélisation sont limitées à des tâches simples. Paralléliser une boucle `for` sans dépendances entre les itérations se fait simplement par l'ajout d'une directive `pragma` :

```
#pragma omp parallel for
for (int i = 0; i < size; ++i)
```

6.3 Stratégies de parallélisation CUDA

6.3.1 Approche brin

À chaque thread CUDA on affecte un brin du hérisson. Il s'agit ici de l'approche ray-tracing. Un thread calcule l'intersection du brin avec l'outil dans toutes les positions.

Pour diminuer le nombre de tests d'intersection rayon-triangle on calcule d'abord l'intersection avec la boîte englobante de l'outil dans une position donnée. S'il y a intersection, on teste alors avec chaque triangle de l'outil. L'avantage de cette approche est qu'il n'y a pas besoin d'utiliser d'opérations atomiques puisqu'un thread gère son propre brin, en effet la hauteur minimale de coupe pour un brin est calculée par un seul thread.

6.3.2 Approche position

À chaque thread CUDA on affecte une position outil à traiter, ce thread applique l'algorithme Z-buffer pour chaque triangle de l'outil dans cette position. La granularité des tâches est élevée : si la quantité de triangles est importante, chaque thread va s'exécuter pendant une longue durée. Si les temps de calcul entre threads sont hétérogènes, certains threads d'un warp peuvent ne plus être actifs et donc du parallélisme est perdu.

Un thread pouvant affecter la hauteur de coupe de plusieurs brins par conséquent un brin peut être mis à jour par plusieurs threads. Ainsi nous avons des problèmes de conflits d'accès comme expliqué précédemment. Nous devons alors utiliser des opérations atomiques pour permettre une mise à jour concurrente de la hauteur des brins.

6.3.3 Approche triangle

À chaque thread CUDA on affecte un triangle de l'outil. Le thread applique alors l'algorithme Z-buffer pour ce triangle dans toutes les positions de la trajectoire. Dans le cas de l'usinage 3 axes les dimensions de la projection du triangle sur la grille restent constantes donc cela ouvre des possibilités d'optimisation. Cependant dans le cas du 5 axes nous n'avons plus cette propriété, de plus chaque position implique une nouvelle matrice de transformation devant être récupérée en mémoire, le nombre de transactions augmente alors de façon importante. Cette méthode est donc intéressante dans le cas d'une trajectoire 3 axes mais inapplicable dans le cas d'une trajectoire 5 axes.

Les mêmes remarques que l'approche précédente s'appliquent concernant la granularité des tâches et les conflits d'accès en mémoire globale.

6.3.4 Approche fragment

Dans cette approche chaque thread travaille sur un seul triangle dans une seule position. Le nom de **fragment** provient de la terminologie utilisée en synthèse d'images. La granularité est ici la plus fine possible : nous n'avons plus de risques d'un mauvais équilibre de la charge de travail. Cependant en contrepartie le nombre de threads est nettement plus élevé : la gestion et l'ordonnancement de milliards de threads impliquent une charge de travail supplémentaire pour le l'ordonnanceur de tâches CUDA et ainsi une augmentation importante du temps de calcul.

6.4 Répartition du travail

6.4.1 Problématique

Dans le cas d'une exécution parallèle, on est toujours limités par l'exécution la plus longue. Une exécution longue d'un thread peut garder tout un warp en otage, ou pire, tout un bloc. Si la granularité de la répartition des tâches est trop élevée, cette situation a davantage de risques de se produire.

6.4.2 Application aux stratégies CPU

L'API OpenMP propose à l'utilisateur de modifier la stratégie d'ordonnancement des différentes itérations d'une boucle entre plusieurs threads. Les différentes possibilités sont : **static, dynamic, guided, runtime**. L'utilisation de ces stratégies n'a pas apporté d'amélioration de performances, elles sont restées identiques voir légèrement dégradées.

6.4.3 Application aux stratégies GPU

L'objectif est de diminuer la granularité de chaque tâche. On peut par exemple essayer de diminuer la granularité d'un facteur G pour chacune des approches que nous avons décrit précédemment :

- **Approche position :** un thread applique l'algorithme Z-buffer pour $\frac{T}{G}$ triangles par position.
- **Approche triangle :** un thread applique l'algorithme Z-buffer pour $\frac{P}{G}$ positions par triangle.
- **Approche fragment :** La granularité de cette approche ne peut pas être plus fine.

6.4.4 Threads persistants

L'ordonnanceur de tâches CUDA présent sur chaque carte graphique est optimisé pour le traitement d'un ensemble de tâches ayant des complexités homogènes. Dans de nombreux cas, ce choix de design s'avère très efficace, cependant dans les cas où le temps d'exécution par tâche possède une variance importante, cette méthode peut ne pas être optimale. Par exemple un thread s'exécutant pendant très longtemps peut garder tout un warp ou tout un bloc actif.

L'article Aila *et al.* (2012) propose une méthode générique pour gérer l'ordonnancement des tâches de façon manuelle dans un kernel CUDA. Cette méthode fonctionne à l'aide d'une file de travail de manière similaire à l'option d'ordonnancement **dynamic** de

OpenMP. Un warp de threads accède à une variable globale et l'incrémente de manière atomique afin de déterminer les numéros des tâches devant être exécutées par ce warp. Avec cette méthode le nombre de threads lancés est constant : on lance exactement assez de threads pour occuper tous les multiprocesseurs de la carte graphique et la répartition des tâches est automatique : si un warp est inoccupé il va chercher un ensemble de tâches à exécuter.

L'implémentation de la méthode des threads persistants est donnée ci-dessous :

```
// Work counter for persistent threads.
__device__ int warpCounter = 0;

#define LOAD_BALANCER_BATCH_SIZE 32

__global__
void persistent_kernel(...)
{
  int tidx = threadIdx.x; // Lane index within warp.
  int widx = threadIdx.y; // Warp index within block.
 // Current ray index in global buffer.
  __shared__ volatile int nextRayArray[8];
 // Number of rays in the local pool.
  __shared__ volatile int rayCountArray[8];
  nextRayArray[widx] = 0;
  rayCountArray[widx] = 0;

  do
  {
    volatile int& localPoolRayCount = rayCountArray[widx];
    volatile int& localPoolNextRay = nextRayArray[widx];

    // Local pool is empty: fetch new rays from the global pool using lane 0.
    if (tidx == 0 && localPoolRayCount <= 0)
    {
      localPoolNextRay = atomicAdd(&warpCounter, LOAD_BALANCER_BATCH_SIZE);
      localPoolRayCount = LOAD_BALANCER_BATCH_SIZE;
    }

    // Pick 32 rays from the local pool.
    // Out of work => done.
    {
      int posidx = localPoolNextRay + tidx;
      if (posidx >= length)
        break;

      if (tidx == 0)
      {
        localPoolNextRay += 32;
        localPoolRayCount -= 32;
      }

      zbuffer(...);
    }
  }
  while (true);
}
```

Chapitre 7

Atomicité

7.1 Introduction

En programmation parallèle, l'atomicité est une propriété indispensable permettant de maintenir un état global cohérent entre les différents processus (threads) d'exécution lorsque ceux-ci accèdent simultanément à un même emplacement mémoire.

Une instruction est dite **atomique** si son exécution peut être considérée comme étant indivisible, c'est-à-dire qu'elle ne peut pas être interrompue par une autre instruction, même provenant d'un thread différent.

7.2 Ordonnancement

Lorsque plusieurs threads s'exécutent en parallèle sur une même machine, chaque cœur dispose de son propre banc de registres. Ainsi une variable résidant dans cet espace mémoire peut être modifiée sans impacter l'exécution des autres threads puisque la portée de ces variables est uniquement locale. Au contraire, la mémoire globale (RAM) est partagée entre tous les processus, une modification d'une valeur en mémoire globale sera visible par tous les autres processus. Lorsque plusieurs processus s'exécutent en parallèle sur des cœurs différents, les instructions processeur peuvent s'entrelacer de manière non prédictible.

7.3 Exemples de non-atomicité

7.3.1 Exemple 1

Nous allons montrer à travers un premier exemple que même pour une tâche simple (un compteur partagé), il est nécessaire d'utiliser des opérations atomiques ou une primitive de synchronisation afin d'obtenir un résultat final cohérent.

Considérons la fonction C suivante :

```
void incr(int* ptr)
{
  int value = *ptr;
  *ptr = value + 1;
}
```

Cette fonction peut se décomposer en 3 opérations élémentaires : un `read` d'une valeur en mémoire, un `add` pour incrémenter la valeur, et enfin un `write` afin d'écrire la nouvelle valeur en mémoire.

Si 2 threads exécutent cette fonction en parallèle, l'ordre d'exécution de ces 3 instructions pour chaque thread est indéterminé. Considérons par exemple la séquence suivante (les threads sont numérotés 1 et 2) :

1. $\texttt{read}_1$
2. $\texttt{add}_1$
3. $\texttt{read}_2$
4. $\texttt{add}_2$
5. $\texttt{write}_1$
6. $\texttt{write}_2$

Dans cet exemple, le thread 2 a lu l'ancienne valeur en mémoire avant que le thread 1 n'y écrive sa nouvelle valeur. A la fin de cette exécution, la valeur en mémoire a été incrémentée de 1 pour 2 appels de fonction. Ce cas est un exemple classique démontrant la nécessité d'utiliser des opérations atomiques afin de d'effectuer la lecture, l'addition et l'écriture de la variable en une seule instruction assembleur **indivisible**. Lorsqu'un résultat dépend de l'ordre d'exécution des instructions des différents threads, on parle alors de **race condition**.

7.3.2 Exemple 2

Un autre problème similaire peut survenir lorsque la lecture d'une variable en mémoire se décompose en réalité en 2 instructions `read` ou `write`. Par exemple la lecture et l'écriture d'une valeur de 64 bits en RAM peuvent être implémentées par deux opérations de 32 bits. Considérons le scénario où le thread 1 écrit en RAM une valeur de 64 bits et en même temps le thread 2 lit en RAM cette même valeur. Chaque thread doit exécuter deuxopérations élémentaires, voici un ordonnancement possible de ces quatre opérations :

1. $\texttt{write}_1(32:63)$, écriture des bits de poids fort
2. $\texttt{read}_2(32:63)$, lecture des bits de poids faible
3. $\texttt{read}_2(0:31)$, lecture des bits de poids faible
4. $\texttt{write}_1(0:31)$, écriture des bits de poids faible

À cause de l'entrelacement des instructions concurrentes, le thread numéro 2 lit une valeur de taille 64 bits particulièrement mise à jour par le thread 1. Cette valeur est totalement incohérente puisqu'elle combine les bits de poids faible d'une valeur avec les bits de poids fort d'une autre valeur.

Prenons l'exemple plus simple d'une opération de 4 bits divisée en deux opérations de 2 bits, la valeur initiale est 0001 (1) et le thread 1 essaye d'écrire la valeur 1100 (12). Avec la séquence d'exécution présentée ci-dessus, le thread 2 va lire la valeur incohérente 1101 (13).

7.4 Application dans SIMSURF

Les problèmes introduits par les deux exemples précédents surviennent lors de la parallélisation CPU ou GPU de l'algorithme N-buffer ou Z-buffer.

- Si plusieurs threads exécutent en parallèle la simulation et tentent de modifier de manière concurrente un même brin de la grille, alors on retrouve le même problème que l'**exemple 1** en remplaçant l'opération d'incrémentation par une opération de minimum.
- Dans le cas d'une implémentation double précision (64-bit) CPU ou GPU, la mise à jour de la hauteur d'un brin peut s'effectuer en 1 ou 2 instructions assembleur selon l'architecture utilisée. Lors d'écritures/lectures concurrentes, l'atomicité de l'opération utilisée doit porter sur l'intégralité des 64 bits de la valeur. Dans le cas contraire, un thread risque de lire une valeur partiellement mise à jour.

Sur la figure 7.1, on peut voir l'impact sur l'allure de la surface lorsque les opérations atomiques ne sont pas utilisées pour éviter les accès concurrents. Les multiples pics que nous voyons correspondent à une mise à jour concurrente de 2 threads pour un même brin.

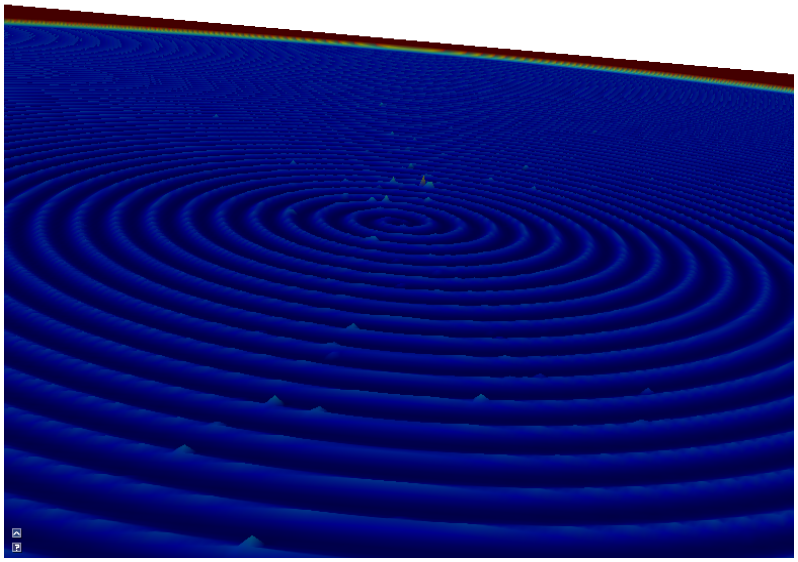


FIGURE 7.1 – Résultat de simulation sans atomicité.

7.5 Garantir l'atomicité

Les 2 points précédents peuvent être résolus de plusieurs manières que nous allons maintenant détailler.

7.5.1 Utilisation d'un verrou

Un verrou (**lock**) est une variable mémoire permettant de limiter l'accès à une structure de données ou un espace mémoire à un seul thread. L'accès contrôlé à cet espace mémoire s'appelle **section critique**. Lorsqu'un thread souhaite accéder à cette section critique, il tente d'acquérir le verrou. Il obtient le verrou si aucun autre thread n'est déjà en section critique, dans le cas contraire le thread demandeur est mis en attente. En sortie de section critique, le verrou est libéré pour laisser la place à un autre thread. Cette primitive de synchronisation est trop haut-niveau pour permettre une utilisation dans notre algorithme de simulation d'usinage, la surcharge de travail induite par la parallélisation ne serait pas acceptable.

7.5.2 Compare-And-Swap

Présentation

Le **compare-and-swap** est une instruction atomique utilisée pour la synchronisation d'un nombre arbitraire de processus. Cette instruction prend 3 paramètres : l'adresse mémoire, l'ancienne valeur qui devrait se trouver à cette adresse et enfin la nouvelle valeur à stocker en mémoire. Atomiquement l'instruction compare la valeur présente à l'adresse mémoire avec la valeur attendue, si les deux valeurs sont identiques alors la valeur en mémoire est remplacée par la nouvelle valeur, dans le cas contraire aucune opération n'est effectuée car la valeur a été modifiée par un autre processus après la lecture du processus courant. Le code C (non atomique) équivalent à l'opération est le suivant :

```
int compare_and_swap (int* reg, int oldval, int newval)
{
  int old_reg_val = *reg;
  if (old_reg_val == oldval)
     *reg = newval;
  return old_reg_val;
}
```

La valeur se trouvant initialement à l'adresse mémoire est toujours retournée, ainsi on peut vérifier si la valeur a été effectivement modifiée.

Implémentation du minimum atomique

Une opération de Compare-And-Swap à une **valeur de consensus** de $+\infty$, c'est-à-dire qu'on peut implémenter toutes les opérations atomiques avec cette opération. Ce qui nous permet d'implémenter des opérations de minimum atomiques à partir du compare-and-swap 32-bit et 64-bit. On utilise une boucle pour appeler l'opération atomique tant qu'elle échoue. Voici l'implémentation d'un minimum atomique sur nombres entiers avec compare-and-swap :

```
  void atomic_min(int* ptr, int new_val)
    {
      if (*ptr < new_val)
        return;
      int old_val = *ptr;

      int tmp;
```

```
    while ((tmp = compare_and_swap(mem, old_val, new_val)) != old_val)
    {
      // New value is smaller that the value we want to write: exit.
      if (tmp < new_val)
        break;
      old_val = tmp;
    }
  }
```

Implémentation CPU

Sur une architecture CPU `x86_64` le compare-and-swap est implémenté par l'instruction assembleur préfixée `lock cmpxchg`. Cette instruction peut traiter des opérandes de taille 32 ou 64 bits. En C++ on peut soit utiliser de l'assembleur `inline` pour utiliser cette instruction, ou alors plus simplement on peut utiliser les fonctions **builtin** de `GCC`. Ces fonctions sont détaillées à l'adresse suivante : gcc.gnu.org/onlinedocs/gcc-4.1.1/gcc/Atomic-Builtins.html. Dans notre cas nous utilisons la fonction suivante :

```
type __sync_val_compare_and_swap(type *ptr,
                                 type oldval,
                                 type newval)
```

Implémentation GPU

L'API CUDA propose 3 implémentations du compare-and-swap : 2 versions 32 bits (signé et non signé) et une version non signé 64 bits.

```
int
atomicCAS(int* address,
          int compare,
          int val);

unsigned int
atomicCAS(unsigned int* address,
          unsigned int compare,
          unsigned int val);

unsigned long long int
atomicCAS(unsigned long long int* address,
          unsigned long long int compare,
          unsigned long long int val);
```

Les versions 32 bits sont disponibles sur tous les accélérateurs de version 1.1 et supérieurs, et la version 64 bits est disponible à partir de la version 1.2.

7.5.3 Instruction dédiée

Certaines architectures disposent d'un ensemble d'instructions permettant de réaliser des opérations de façon atomique. Ces opérations sont généralement limitées : addition, soustraction, incrémentation, décrémentation, échange (swap), etc.

Pour une implémentation parallèle de notre simulateur, nous avons besoin d'une opération atomique permettant de réaliser un minimum sur une valeur flottante 32-bit ou 64-bit. Le jeu d'instruction implémenté par notre architecture CPU (`x86_64`) ne dispose pas d'instructions permettant de réaliser un minimum atomique, seul le compare-and-swap est donc utilisable. Concernant l'implémentation GPU, L'API CUDA propose les opérations citées précédemment et une fonction `atomicMin` :

```
int atomicMin(int* address, int val);
```

La mémoire pointée par la variable `address` peut soit résider en mémoire globale soit en mémoire partagée. Cette fonction est disponible pour tous les accélérateurs de version 1.1 minimum. Il n'existe pas de version s'exécutant sur des nombres flottants, mais ce problème peut être résolu en tirant parti de la représentation binaire **IEEE 754** et en utilisant la technique du **type punning**. Les détails de cette technique sont présentés dans une section ultérieure de ce rapport.

7.5.4 Benchmark

Le tableau suivant présente l'impact sur le temps de calcul de l'utilisation des opérations atomiques 32-bit et 64-bit. Ce benchmark a été réalisé sur la configuration GTX 560 et AMD Phenom II X4 955.

	Temps GPU (ms)	Temps CPU (ms)
sans atomicité (32-bit)	689.7	24773
atomicMin (32-bit)	690.3	✖
compare-and-swap (32-bit)	750.5	24984
sans atomicité (64-bit)	3035	27348
compare-and-swap (64-bit)	3072	27756

Nous pouvons observer qu'avec CUDA l'instruction spécialisée `atomicMin` est plus rapide que l'implémentation à base de compare-and-swap sous CUDA, cette opération est aussi rapide qu'un accès mémoire sans atomicité. Pour les implémentations CPU et GPU 64-bit, la surcharge de travail introduite par l'atomicité reste négligeable

Il est important de noter que pour une implémentation CPU parallèle d'autres mécanismes interviennent en plus du coût d'une opération atomique. Par exemple maintenir la cohérence des caches de chaque cœur implique une surcharge supplémentaire pour une implémentation multicore utilisant une structure de donnée partagée.

Chapitre 8

Intersection rayon-triangle

Les algorithmes Z-buffer ou N-buffer ont pour partie centrale le test d'intersection entre un rayon et un triangle. Ce test est effectué des milliards de fois par configuration. Prenons par exemple le cas d'une configuration utilisant un outil de 2000 triangles et 100000 positions. La taille du fragment de chaque triangle est en moyenne de 4×4. Le nombre de tests d'intersection est environ :

$$4 \times 4 \times 2000 \times 100000 = 3.2 \times 10^9 \tag{8.1}$$

En termes de nombre d'instructions assembleur exécutées, une écrasante majorité proviendra de ce test d'intersection. En effet, la transformation du triangle et le calcul du fragment sont négligeables en comparaison puisqu'ils ne sont réalisés qu'une seule fois par triangle. Ce test doit donc être optimisé en priorité si l'on souhaite améliorer les performances à la fois CPU et GPU de la simulation. Cependant il est important de noter qu'une diminution du nombre d'opérations n'implique pas forcément une amélioration du temps d'exécution. Le plus souvent les kernels CUDA sont limités par la bande passante mémoire (**memory-bound kernel**) et non par la capacité de calcul de la carte (**compute-bound kernel**).

Au cours du stage, l'implémentation de l'intersection a été modifiée plusieurs fois. Nous allons présenter ici les différentes versions par ordre chronologique, puis nous présenterons l'impact sur les performances qu'a eu chaque optimisation.

8.1 Intersection générale rayon-triangle

Cette méthode d'intersection provient de l'article Möller et Trumbore (1997). Cet article présente un algorithme simple et nécessitant peu de mémoire. Cette méthode n'exige pas de prétraitement des triangles ni de stockage de données supplémentaires comme la normale du triangle. Seulement les coordonnées des sommets sont nécessaires. De plus, cette méthode ne nécessite pas de calculer l'intersection entre le rayon et le plan dans lequel est inscrit le triangle. À la place, le calcul d'intersection est effectué dans le repère du triangle. Une transformation doit donc être appliquée à l'origine et à la direction du rayon. Nous allons maintenant présenter brièvement cet algorithme en suivant le fil de l'article cité.

8.1.1 Principe

Un rayon $R(t)$ ayant pour origine O et une direction normalisée D est définie par l'équation :

$$R(t) = O + tD \tag{8.2}$$

Le triangle à tester est défini par trois sommets V_0, V_1 et V_2. Un point $T(u, v)$ à l'intérieur d'un triangle est donné par l'équation suivante :

$$T(u, v) = (1 - u - v)V_0 + uV_1 + vV_2 \tag{8.3}$$

u et v sont appelés les coordonnées barycentriques du triangle. Ces coordonnées sont utiles dans de nombreuses applications en rendu 3D : application d'une texture, interpolation d'une normale, interpolation d'une couleur, etc. Les inégalités suivantes sont vérifiées afin que le point soit bien à l'intérieur du triangle :

1. $u \geq 0$
2. $v \geq 0$
3. $u + v \leq 1$

Calculer l'intersection entre le rayon $R(t)$ et le triangle équivaut à trouver un couple (u, v) tel que :

$$\begin{array}{rcl} R(t) & = & T(u, v) \\ O + tD & = & (1 - u - v)V_0 + uV_1 + vV_2 \end{array} \tag{8.4}$$

En réorganisant l'expression nous obtenons la forme matricielle suivante :

$$\left[\begin{array}{ccc} -D, & V_1 - V_0, & V_2 - V_0 \end{array} \right] \left[\begin{array}{c} t \\ u \\ v \end{array} \right] = O - V_0 \tag{8.5}$$

Nous pouvons donc obtenir les coordonnées u et v sur le triangle et la distance t du rayon en triangle en résolvant ce système d'équations. Géométriquement, cette équation correspond à une translation du triangle à l'origine puis à l'application d'une transformation pour que le triangle devienne unitaire en y et en z avec la direction du rayon alignée avec l'axe x. La figure 8.1, provenant de l'article original, illustre ce procédé :

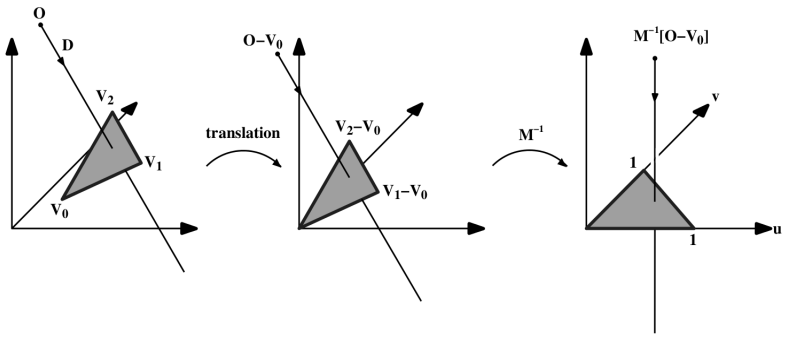


FIGURE 8.1 – Algorithme d'intersection rayon-triangle.

La matrice de transformation, M correspond à : $\left[\begin{array}{ccc} -D, & V_1 - V_0, & V_2 - V_0 \end{array} \right]$

Notons $E_1 = V_1 - V_0$, $E_2 = V_2 - V_0$ et $T = O - V_0$. En utilisant la règle de Cramer, la solution à l'équation précédente est :

$$\left[\begin{array}{c} t \\ u \\ v \end{array} \right] = \frac{1}{\left| \begin{array}{ccc} -D, & E_1, & E_2 \end{array} \right|} \left[\begin{array}{ccc} |T, & E_1, & E_2| \\ |-D, & T, & E_2| \\ |-D, & E_1, & T| \end{array} \right] \tag{8.6}$$

Le déterminant de trois vecteurs dans l'espace Euclidien de dimension 3 est donné par la formule :

$$\begin{vmatrix} A, & B, & C \end{vmatrix} = -(A \times C) \cdot B = -(C \times B) \cdot A \tag{8.7}$$

L'équation précédente peut ainsi être réécrite sous la forme :

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{(D \times E_2) \cdot E_1} \begin{bmatrix} (T \times E_1) \cdot E_2 \\ (D \times E_2) \cdot T \\ (T \times E_1) \cdot D \end{bmatrix} = \frac{1}{P \cdot E_1} \begin{bmatrix} Q \cdot E_2 \\ P \cdot T \\ Q \cdot D \end{bmatrix} \tag{8.8}$$

Avec $P = D \times E_2$ et $Q = T \times E_1$. Nous pouvons calculer ces expressions séparément et les réutiliser pour éviter d'effectuer des opérations redondantes.

8.1.2 Algorithme

L'algorithme suivant, écrit en C++, est adapté de l'article mentionné. Afin d'optimiser les performances, cet algorithme est paresseux (**lazy**), c'est-à-dire que les calculs ne sont effectués que lorsque c'est nécessaire. Par exemple, il n'est pas nécessaire de calculer v si u ne vérifie pas l'inégalité présentée. La variable `EPSILON` assure la stabilité numérique de l'algorithme, ce test élimine les rayons parallèles au triangle en comparant le déterminant à un petit intervalle autour de zéro.

Comme expliqué dans ce rapport, l'algorithme d'intersection est implémenté par une classe de type **policy**. L'algorithme de simulation utilisé prend en paramètre `template` la classe effectuant le calcul d'intersection. On peut ainsi facilement utiliser le même squelette d'algorithme en utilisant des algorithmes d'intersection différents. Cette classe d'intersection est également paramétrée par le type du triangle, cela permet de générer du code simple et double précision sans dupliquer le code.

L'intérêt de la méthode `possible()` est détaillé dans la prochaine section. La méthode `intersection` renvoie `true` si il y a une intersection entre le rayon et le triangle et alors la variable `dist` contient la distance entre l'origine et le point d'intersection. S'il n'y a pas d'intersection, `false` est renvoyé.

Le constructeur de la classe permet de précalculer certaines données. Ce constructeur étant appelé avant la double boucle sur les coordonnées du fragment, les données ne seront calculées qu'une fois par triangle pour une position donnée. Pour cet algorithme nous pouvons précalculer les coordonnées des arêtes du triangle : `edge1_` et `edge2_`.

```
template <typename triangle_type>
class intersect_general
{
public:
  typedef typename triangle_type::vertex_type vertex_type;
  typedef typename triangle_type::value_type value_type;

  __device__ __host__
  intersect_general(const triangle_type& t)
    : t_(t),
      edge1_(t.b - t.a),
      edge2_(t.c - t.a)
    {
    }

  __device__
  bool possible()
    {
      return true;
    }

  __device__ __inline__
```

```
  bool intersection(const vertex_type& O, cont vertex_type& D, float& dist)
    {
      // begin calculating determinant - also used to calculate u parameter
      vertex_type pvec = cross(D, edge2_);
      // if determinant is near zero, ray lies in plane of triangle
      value_type det = dot(edge1_, pvec);

      if (det > -EPSILON && det < EPSILON)
        return false;

      value_type inv_det = value_type(1.) / det;

      // calculate distance from vert0 to ray origin
      vertex_type tvec = O - t_.a;

      // calculate U parameter and test bounds
      value_type u = dot(tvec, pvec) * inv_det;
      if (u < 0.f || u > value_type(1.))
        return false;

      // prepare to test V parameter
      vertex_type qvec = cross(tvec, edge1_);

      // calculate V parameter and test bounds
      value_type v = dot(D, qvec) * inv_det;
      if (v < value_type(0.) || u + v > value_type(1.))
        return false;

      // calculate distance, ray intersects triangle
      dist = dot(edge2_, qvec) * inv_det;
      return true;
    }
private:
  const triangle_type& t_;
  const vertex_type edge1_;
  const vertex_type edge2_;
};
```

8.1.3 Complexité

Nous allons calculer le nombre d'opérations nécessaires dans le cas où le rayon intersecte le triangle (toutes les opérations doivent donc être effectuées). Nous devons d'abord décomposer les deux opérations algébriques utilisées :

- **Produit scalaire :** 3 multiplications, 2 additions.
- **Produit vectoriel :** 6 multiplications, 3 soustractions.

Le nombre d'opérations élémentaires à effectuer dans le pire cas est :

- 27 multiplications
- 1 division
- 9 additions
- 9 soustractions

8.2 Optimisation Z-buffer

Ce nouvel algorithme s'applique uniquement à l'algorithme Z-buffer et permet de réduire nettement le nombre d'opérations en tirant profit des hypothèses supplémentaires de cet algorithme par rapport à l'algorithme N-buffer.

8.2.1 Propagation de constantes

Dans le cas de l'algorithme Z-buffer que nous avons décrit, l'orientation des brins est toujours la même : parallèle à l'axe z et dirigé vers les z positifs. Le vecteur direction du rayon est donc toujours $(0, 0, 1)$. Concernant l'origine du rayon, nous pouvons considérer que sa composante en z est toujours 0. Nous pourrions affecter l'origine du rayon à la hauteur actuelle du brin, mais cela demanderait un accès mémoire supplémentaire au début pour récupérer la hauteur initiale. Si la suite de l'algorithme révèle qu'il n'y a pas d'intersection entre le triangle et le brin, l'accès mémoire aura été inutile. Les accès mémoires étant très souvent un facteur limitant, particulièrement sur GPU, nous avons choisi cette stratégie de calcul pour les limiter.

Pour un point (x, y) de la grille nous avons donc $O = (x, y, 0)$ et $D = (0, 0, 1)$. En injectant ces constantes dans l'algorithme précédent et en les propageant dans tous les calculs, nous pouvons éliminer de nombreuses opérations inutiles.

8.2.2 Back-face culling

L'article cité propose deux versions de l'algorithme d'intersection : une version générale pour n'importe quel triangle et une version avec élimination des intersections sur les faces cachées (**back-face culling**). Cette seconde version peut être utilisée dans une application de raytracing puisque les intersections avec les faces cachées ne seront pas visibles sur la scène finale. L'utilisation du **culling** diminue le nombre d'opérations nécessaires au calcul de l'intersection, mais l'orientation des faces (**clockwise** ou **anti-clockwise**) dépend de l'ordre de déclaration des sommets du triangle. Étant donné que les triangles peuvent subir une transformation quelconque, nous ne pouvons pas corriger l'orientation des triangles lors d'une étape de prétraitement global. La correction de l'orientation des sommets est appliquée dans le constructeur de notre algorithme d'intersection, c'est à dire avant la boucle sur tous les brins du fragment. Cette réorientation n'est appliquée qu'une fois, ce qui permet ensuite d'utiliser le code plus optimisé avec **back-face culling** pour chaque itération de la boucle sur le fragment.

8.2.3 Algorithme

En appliquant les deux optimisations citées précédemment, nous obtenons l'algorithme suivant. La méthode **possible()** est appelée immédiatement après l'appel au constructeur de l'algorithme, cette méthode renvoie `true` si une intersection est possible, c'est-à-dire si le plan du triangle n'est pas parallèle au brin. Si c'est le cas, il est inutile d'itérer sur le fragment.

```
template <typename triangle_type>
class intersect_zbuffer
{
public:
  typedef typename triangle_type::vertex_type vertex_type;
  typedef typename triangle_type::value_type value_type;
public:
  __device__
  intersect_zbuffer(const triangle_type& t)
    : t_(t)
    {
      det_ = (t_.c.x - t_.a.x) * (t_.a.y - t_.b.y) + (t_.c.y - t_.a.y) * (t_.b.x -
          t_.a.x);
      // Check face orientation.
      if (det_ < value_type(-EPSILON))
      {
        // Invert determinant and reorient triangle.
        det_ = -det_;
        edge1_ = t.b - t.a;
        edge2_ = t.c - t.a;
      }
      else
      {
        edge1_ = t.c - t.a;
        edge2_ = t.b - t.a;
      }
    }

  __device__
  bool possible()
  {
    return det_ > value_type(EPSILON);
  }

  __device__ __inline__
  bool intersection(const value_type x, const value_type y, value_type& dist)
    {
      // calculate distance from vert0 to ray origin
      vertex_type tvec = {x - t_.a.x, y - t_.a.y, -t_.a.z};

      // calculate U parameter t_.and test bounds
      value_type u = tvec.x * (-edge2_.y) + tvec.y * edge2_.x;
      if (u < value_type(0.) || u > det_)
        return false;

      // prepare to test V parameter
      vertex_type qvec = cross(tvec, edge1_);

      // calculate V parameter and test bounds
      value_type v = qvec.z;
      if (v < value_type(0.) || u + v > det_)
        return false;

      // calculate distance, ray intersects triangle
      dist = dot(edge2_, qvec) / det_;
      return true;
    }
private:
  const triangle_type& t_;
  vertex_type edge1_;
  vertex_type edge2_;
```

```
  value_type det_;
};
```

8.2.4 Complexité

- 11 multiplications
- 1 division
- 4 additions
- 7 soustractions

8.3 Rastérisation

On utilise la méthode d'intersection utilisée en rastérisation La rastérisation est le procédé permettant de transformer un ensemble de primitive géométriques définies vectoriellement (par exemple des triangles, des sphères) en une image discrète. Cette technique est utilisée par la carte graphique afin de transformer un ensemble de triangles en un ensemble de pixels pouvant être affichés à l'écran pour représenter la scène visualisée.

8.3.1 Présentation

En synthèse d'images, la primitive géométrique utilisée est le triangle. En utilisant la matrice de projection active (`GL_PROJECTION` pour OpenGL), les triangles sont projetés sur le plan proche (**near plane**) de la pyramide de vue (**viewing frustum**). Le fragment de chaque triangle (l'ensemble des pixels se situant sous la projection) est calculé, pour chaque pixel du fragment un test d'intersection est effectué. S'il y a intersection, alors la distance d'intersection est calculée par interpolation à partir des coordonnées de chaque sommet. Cette distance est ensuite comparée à celle située dans le **depth buffer** pour déterminer si ce point d'intersection est le plus proche trouvé jusqu'à maintenant. Ce calcul correspond à un test de visibilité du triangle depuis le pixel considéré. La carte graphique fait ce travail nativement de manière extrêmement efficace, mais le hardware qui effectue ce calcul n'est pas accessible via l'API CUDA (voir article Laine et Karras (2011)).

Cette méthode ressemble à notre algorithme de Z-buffer. Les pixels de l'algorithme original deviennent les brins de notre grille. Mais, comme notre surface de départ n'est pas forcément plane, nous n'avons pas de plan de projection évident. Comme pour l'algorithme d'intersection précédent, nous pouvons utiliser le plan $z = 0$ comme plan de projection. Étant donné que ce plan ne correspond pas à notre surface initiale, nous ne pouvons pas appliquer l'algorithme de rastérisation classique : nous devons supprimer la composante de perspective. En d'autres termes, notre matrice de transformation n'utilisera pas les coordonnées homogènes. Un article présentant une architecture de rastérisation sous CUDA est Zhang et Majdandzic (2010).

8.3.2 Algorithme

Le triangle étant projeté sur un plan, le test d'intersection se résume à un test d'inclusion d'un point dans un triangle dans un espace euclidien 2D. Un premier avantage de cette méthode est que nous n'utilisons donc pas les composantes en z des sommets avant d'avoir vérifié que le brin est bien à l'intérieur du triangle. Le nombre de transactions mémoires est donc diminué si le test d'intersection échoue.

Prenons un triangle dans l'espace euclidien 2D usuel. Ce triangle est défini par 3 sommets V_0, V_1, V_2 avec $V_i = (x_i, y_i)$. Pour chaque paire de sommets, nous pouvons définir une **fonction d'arête** permettant de déterminer de quel coté de l'arête se trouve un point (x, y) donné. Nous avons donc trois fonctions d'arête :

$$\begin{aligned} e_0(x,y) &= -(y_2 - y_1)(x - x_1) + (x_2 - x_1)(y - y_1) \\ e_1(x,y) &= -(y_0 - y_2)(x - x_2) + (x_0 - x_2)(y - y_2) \\ e_2(x,y) &= -(y_1 - y_0)(x - x_0) + (x_1 - x_0)(y - y_0) \end{aligned} \tag{8.9}$$

Pour un point 2D fixé, si les fonctions d'arête s'évaluent en un résultat positif, alors ce point est compris dans le triangle. Si un seul résultat est négatif, le point est en dehors du triangle. À partir des résultats de ces trois fonctions, nous pouvons passer aux coordonnées barycentriques avec seulement 2 divisions (pouvant être transformées en multiplication) et 2 soustractions :

$$\begin{aligned} u &= \frac{e_1(x,y)}{2A} \\ v &= \frac{e_2(x,y)}{2A} \\ w &= 1 - u - v \end{aligned} \tag{8.10}$$

A est l'aire signée du triangle, calculée par la formule suivante :

$$A = \frac{1}{2}((x_1 - x_0)(y_2 - y_0) - (x_2 - x_0)(y_1 - y_0)) \tag{8.11}$$

Si le point (x, y) est à l'intérieur du triangle, nous pouvons interpoler sa position en z à partir des coordonnées en z des trois sommets : (z_0, z_1, z_2). En utilisant les coordonnées barycentriques que nous venons de calculer, nous pouvons interpoler très simplement la profondeur du point d'intersection :

$$z(x, y) = wz_0 + uz_1 + vz_2 \tag{8.12}$$

Le code C++ obtenu est au final nettement plus concis et plus compréhensible que le code de l'algorithme précédent. Le test d'inclusion est maintenant rassemblé en un seul test utilisant trois appels à la fonction `signbit`. Cette fonction renvoie `true` si le nombre flottant à son bit de signe à 1, `false` sinon. Cette opération peut s'implémenter très simplement avec un masque et des opérations logiques, rendant le test d'inclusion très rapide.

```
template <typename triangle_type>
class intersect_raster
{
public:
  typedef typename triangle_type::vertex_type vertex_type;
  typedef typename triangle_type::value_type value_type;
public:
  __device__
  intersect_raster(const triangle_type& t)
    : t_(t)
    {
      det_ = (t_.c.x - t_.a.x) * (t_.a.y - t_.b.y) + (t_.c.y - t_.a.y) * (t_.b.x -
          t_.a.x);
      p0 = t_.a;
      // Check face orientation.
      if (det_ < value_type(-EPSILON))
      {
        det_ = -det_;
        p1 = t_.c;
        p2 = t_.b;
      }
      else
      {
        p1 = t_.b;
        p2 = t_.c;
      }
    }

  __device__
  bool possible()
  {
    return det_ > value_type(EPSILON);
  }

  __device__ __inline__
  bool intersection(const value_type x, const value_type y, value_type& dist)
    {
      // Edge functions.
      value_type e0 = (p1.y - p2.y) * (x - p1.x) + (p2.x - p1.x) * (y - p1.y);
      value_type e1 = (p2.y - p0.y) * (x - p2.x) + (p0.x - p2.x) * (y - p2.y);
      value_type e2 = (p0.y - p1.y) * (x - p0.x) + (p1.x - p0.x) * (y - p0.y);

      // Check inclusion in triangle.
      if (signbit(e0) || signbit(e1) || signbit(e2))
        return false;

      value_type inv_det = value_type(1.) / det_;
      // Barycentric coordinates.
      value_type u = e1 * inv_det;
      value_type v = e2 * inv_det;
      value_type w = value_type(1.) - u - v;

      // Depth interpolation.
      dist = (w * p0.z + u * p1.z) + v * p2.z;

      return true;
    }
private:
  const triangle_type& t_;
  vertex_type p0;
  vertex_type p1;
  vertex_type p2;
  value_type det_;
};
```

8.3.3 Complexité

- 11 multiplications
- 1 division
- 5 additions
- 14 soustractions

Cet algorithme utilise plus d'opérations et plus de mémoire que le précédent, mais il compense par son test d'inclusion extrêmement simple (par le passage en 2D). De plus, sur GPU, de nombreuses opérations peuvent être réunies pour former une opération FMA (se référer à la partie optimisation CUDA de ce rapport). Nous allons voir que sur CPU la différence de performance est notable.

8.4 Benchmark

8.4.1 GPU

Nous avons comparé les trois méthodes d'intersections sous CUDA sur la carte GeForce GTX 560 Ti sur 4 configurations différentes.

	général	Z-buffer	rastérisation
ebauche	2343 ms	773,6 ms	751,8 ms
finition	4027 ms	1385,3 ms	1352,8 ms
volcano	206,9 ms	99,5 ms	94,7 ms
random	106,0 ms	44,3 ms	39,3 ms

Nous pouvons observer, sans surprise, une différence importante de temps de calcul entre l'implémentation générale et la même implémentation optimisée pour le Z-buffer. La différence avec la troisième implémentation est ensuite minime. Cela s'explique par la différence du nombre de registres utilisés, voici, pour chaque algorithme, le nombre de registres utilisés par thread, dans le cas d'un usinage 3 axes :

- **général :** 50
- **Z-buffer :** 35
- **raster :** 38

Le premier algorithme utilise plus de registres car de nombreuses variables supplémentaires doivent être utilisées. Le troisième algorithme utilise plus de registres que le second mais compense par un test d'inclusion plus rapide.

8.4.2 CPU

	général	Z-buffer	rastérisation
ebauche	42892 ms	34625 ms	28756 ms
finition	51297 ms	35531 ms	31427 ms
volcano	8321 ms	6432 ms	5794 ms
random	2033 ms	1853 ms	1623 ms

La différence de temps de calcul entre la version Z-buffer et la version rastérisation est plus observable dans ce cas là. Les raisons possibles de cette différence sont tout d'abord une meilleure optimisation par le compilateur et l'utilisation de la fonction `signbit` pour tester l'inclusion dans le triangle.

8.5 Conclusion

Nous avons présenté le principe et l'implémentation C++ des algorithmes d'intersection ayant été utilisés successivement au cours du projet. Sur GPU les 2 dernières méthodes sont équivalentes, mais sur CPU la différence de performance est importante. De plus, la troisième version est plus compréhensible mathématiquement que la version optimisée Z-buffer qui est la version générale ayant subi une passe de propagation de constantes. Enfin, cette version est celle utilisée dans le cas de la rastérisation de triangle, nous restons ainsi dans la logique de notre approche par fragments.

Chapitre 9

Généricité et réutilisabilité

9.1 Problématique

Les sections précédentes ont présenté de nombreuses optimisations et variantes de l'algorithme de simulation choisi. Nous avons un algorithme de base commun : l'algorithme Z-buffer, mais plusieurs aspects de cet algorithme peuvent être modifiés :

- **Support d'exécution :** l'algorithme peut s'exécuter sur le CPU ou sur le GPU. Si une fonction doit être compilée pour une exécution GPU, alors elle doit être déclarée avec l'attribut `__device__`. Une exécution GPU doit diviser le calcul en threads puis en blocs pour bénéficier de l'architecture massivement parallèle de CUDA. Au contraire une exécution CPU peut utiliser une simple boucle `for`.
- **Stratégie de parallélisation :** nous avons présenté trois principales stratégies de parallélisation CUDA.
- **Algorithme d'intersection :** nous sommes partis d'un algorithme général d'intersection rayon-triangle que nous avons optimisé pour le cas du Z-buffer. Nous avons introduit trois algorithmes différents.
- **Précision :** les calculs peuvent s'effectuer en simple précision (avec des variables de type `float`) ou double précision (avec des variables de type `double`)
- **Atomicité :** l'opération à utiliser dépend à la fois du support d'exécution et de la précision utilisée. On peut aussi choisir de ne pas utiliser d'opérations atomiques (dans le cas d'une exécution CPU).
- **Type d'usinage :** le type d'usinage est lié à la trajectoire de la configuration de simulation utilisée. Il existe au total quatre stratégies d'usinage : 3 axes avec ou sans rotation broche et 5 axes avec ou sans rotation broche. Dans le cas d'un outil représenté par un maillage de triangles, chaque type d'usinage implique une transformation différente sur les triangles. Par exemple un usinage 3 axes nécessite uniquement une translation alors qu'un usinage 5 axes nécessite d'appliquer une matrice de transformation puis une translation.

Si nous devions implémenter un algorithme séparé pour chaque variation de l'algorithme Z-buffer, le produit cartésien des stratégies citées implique que le nombre d'algorithmes à implémenter est dantesque :

$$N = 2 \times 3 \times 3 \times 2 \times 2 \times 4 = 288 \tag{9.1}$$

Implémenter 288 algorithmes est irréalisable. Même en implémentant un sous-ensemble de M algorithmes parmi les 288 algorithmes, la majeure partie du code serait dupliquée M fois. Chaque implémentation ne présenterait qu'une faible variation par rapport aux autres. De plus chaque modification du squelette de l'algorithme doit être répercutée sur

tous les algorithmes implémentés. Cette méthode rendrait le projet non maintenable et a donc été rejetée immédiatement.

9.2 Solutions

En utilisant les fonctionnalités du C++, trois solutions peuvent être envisagées afin de ne pas avoir à implémenter l'ensemble des algorithmes.

9.2.1 Génération de code

À partir d'un squelette d'algorithme, du code est généré pour chaque variation de l'algorithme. La génération de code peut se faire de plusieurs manières : avec des macros C, avec un langage de script, etc. Cette méthode est difficile à prendre en oeuvre et difficile à déboguer.

9.2.2 Dispatch dynamique

Le patron de conception Stratégie peut s'appliquer pour certains points cités ci-dessus. Par exemple on peut créer des stratégies pour le type d'usinage, les algorithmes d'intersection et l'atomicité. Le désavantage de cette méthode est son coût élevé en performances : l'utilisation d'une hiérarchie de stratégies implique que le dispatch vers la bonne implémentation sera effectué dynamiquement (en utilisant des méthodes virtuelles). Cette pénalité de performances n'est pas acceptable pour l'implémentation du cœur de l'algorithme de simulation. L'augmentation du nombre d'opérations serait largement pénalisante puisque chaque stratégie pourrait être appelée des milliards de fois.

9.2.3 Classe de politique

Cette méthode est inspirée des classes de **politique** (**policy class**) présentées dans Alexandrescu (2001), elle est basée sur une utilisation intensive des fonctionnalités de généricité C++ à travers l'utilisation des **templates**. Cette méthode utilise des stratégies comme la méthode précédente, mais il n'y a pas de surcoût à l'exécution, car le dispatch est statique et non dynamique. L'utilisation des templates permet, pour chaque combinaison de stratégies, de générer le code optimisé correspondant dans une nouvelle fonction. Un autre avantage est l'extensibilité : par exemple un nouvel algorithme d'intersection peut être facilement rajouté et s'intégrer avec le reste. Les nouvelles classes doivent simplement respecter l'interface de la stratégie implémentée.

Cette méthode est la fusion des précédentes, mais sans pénalité de performances et sans duplication du code source grâce à l'utilisation de la puissance des templates C++. C'est donc cette méthode qui a été retenue pour implémenter l'algorithme Z-buffer et ses variations. Il y a toutefois une duplication du code assembleur pour chaque algorithme ce qui augmente la taille du binaire, mais ce n'est pas un facteur limitant dans notre application.

9.3 Classes de stratégie

Nous allons introduire, pour chaque aspect de l'algorithme que nous présenté, comment nous avons implémenté les différentes stratégies. Les rôles de ces stratégies n'étant pas tous orthogonaux, il est nécessaire que certaines stratégies collaborent entre elles pour trouver la bonne implémentation (par exemple précision et atomicité sont liées).

- **Support d'exécution :** un kernel CUDA doit être configuré lors de son appel contrairement à une fonction C. De plus, sur GPU nous devons prendre en compte l'organisation en blocs et en threads afin de retrouver la tâche à exécuter : un calcul sur les indices est requis. Avoir une seule fonction pour le CPU et pour le GPU n'est pas possible. À la place les fonctions CPU et GPU appellent la même fonction C++ mais en utilisant des stratégies différentes.
- **Stratégie de parallélisation :** nous n'avons pas appliqué de classe de stratégie ici au vu du point précédent. La parallélisation des tâches est effectuée dans la fonction CPU ou GPU appelante.
- **Algorithme d'intersection :** l'algorithme d'intersection est implémenté avec une classe de stratégie, ces implémentations sont présentées dans ce rapport.
- **Précision :** l'utilisation de variables de type `float` ou de type `double` peut être implémentée en utilisant des templates C++, sans utilisation de classe de stratégie.
- **Atomicité :** l'atomicité est implémentée par des stratégies dépendant du support d'exécution. L'opération atomique devant être utilisée dépend de la précision, par conséquent cette stratégie est également paramétrée par la précision utilisée.
- **Type d'usinage :** à chaque type d'usinage correspond une classe de stratégie. Ces classes peuvent être utilisées pour une implémentation CPU ou GPU puisqu'elles ne font pas appel à des instructions particulières.

La généricité aurait pu être poussée davantage par exemple pour la stratégie de parallélisation, mais le compilateur CUDA NVCC ne gère pas toujours correctement les codes C++ faisant une utilisation intensive des templates. Dans certains cas du code valide était rejeté.

9.4 Exemple

Pour illustrer ce que nous venons d'expliquer, nous allons présenter des extraits du code de l'application développée. Tout d'abord voici la signature du squelette de l'algorithme Z-buffer :

```
template
<
  typename T,
  typename position_type,
  typename surface_type,
  typename tool_type,
  template <class> class machining_policy,
  template <class> class intersect_policy,
  template <class> class min_policy
>
__device__
inline
void zbuffer(const position_type& pos,
       surface_type& surf,
       const tool_type& tool)
```

Le type `T` est la précision utilisée : `float` ou `double`. Nous avons également en paramètre le type de position (lié au type d'usinage), le type de surface et le type de l'outil (liés à la précision utilisée). Enfin nous avons trois politiques restantes : type d'usinage (`machining_policy`), algorithme d'intersection (`intersect_policy`), et l'opération atomique utilisée (`min_policy`).

La stratégie d'usinage est initialisée sur une position (un n-uplet) puis appliquée sur chaque triangle :

```
  machining_policy<triangle_type> machining(pos);
  ...
```

```
machining.transform(triangle);
```

La stratégie d'intersection est paramétrée par le type de triangle et est initialisée avant le début de la boucle sur le fragment :

```
intersect_policy<triangle_type> intersect(triangle);
if (!intersect.possible())
  continue;
```

La stratégie d'opération atomique est utilisée dans la boucle sur le fragment si il y a intersection entre le triangle et le brin :

```
if (intersect.intersection(gridposx, gridposy, dist))
  min_policy<T>::update(&surf[y * grid_size + x], dist);
```

Chapitre 10

Architecture

Dans ce chapitre est présentée l'architecture logicielle que nous avons choisie pour le projet SIMSURF.

10.1 Architecture initiale

Au début du projet, le projet était séparé en plusieurs exécutables, chacun effectuant une tâche simple en suivant la philosophie UNIX. Les principaux programmes implémentés étaient :

- **Simulation CPU :** l'implémentation CPU parallèle ou séquentielle de l'algorithme Z-buffer. Ce programme prenait en entrée une surface, un outil et une trajectoire et écrivait le résultat dans un fichier surface choisi.
- **Simulation GPU :** la première implémentation CUDA de l'algorithme (non optimisée). Le fonctionnement était identique au programme CPU.
- **Visualisation :** implémentation OpenGL de visualisation d'une surface. Les fonctionnalités de caméra étaient minimales et les performances faibles. Un seul mode d'affichage était disponible. Ce programme prenait en argument le fichier surface à visualiser.
- **Comparaison :** programme de comparaison (en erreur absolue) de deux fichiers surfaces passés en argument, ce programme écrivait sur son entrée standard la distance entre les deux surfaces.
- **Trajectoire :** génération de trajectoires 3 axes en suivant un modèle (diagonale, volcan) ou une courbe paramétrique (spirale d'Euler).
- **Vérification :** vérification de la cohérence des résultats entre les implémentations CPU et GPU. Pour chaque exemple, les deux programmes de simulations sont appelés puis le programme de comparaison est utilisé pour calculer la distance entre les fichiers surfaces obtenus. Cette fonctionnalité était implémentée grâce au système de tâches de l'outil **make**.

La modularité de cette stratégie est bonne, mais du code était dupliqué entre les différents programmes (par exemple les codes d'écriture/lecture de fichiers de trajectoire, d'outil ou de surface). L'objectif était de pouvoir commencer à expérimenter sur chaque partie séparément sans se soucier des modifications sur les autres parties (à condition que l'interface d'entrée et de sortie soit respectée). Cette architecture est intéressante : elle permet limiter la quantité de code des programmes utilisant des technologies difficiles à maîtriser, par exemple CUDA et OpenGL.

10.2 Architecture Modèle-Vue-Contrôleur

L'objectif du projet étant de développer un démonstrateur permettant un zoom en temps réel sur une partie de la surface, il était nécessaire de rassembler simulation et visualisation. Ainsi, une fois les premières expérimentations CUDA et OpenGL terminées, il a été décidé de rassembler les fonctionnalités citées précédemment dans un seul programme. Trois composants principaux devaient être implémentés :

- **Simulation :** application de l'algorithme Z-buffer sur la configuration d'entrée.
- **Visualisation :** affichage du résultat de la simulation en 3D avec OpenGL.
- **Interaction :** déplacement/orientation de la caméra, zoom sur une partie de la surface par un clic utilisateur.

Ces trois composants interagissent entre eux par exemple pour recalculer le résultat si la configuration a changé ou pour changer la vue si la caméra a bougé. Chaque modification étant initiée par la partie **interaction** du projet. Cette architecture peut être implémentée par le patron de conception (**design pattern**) Modèle-Vue-Contrôleur (MVC). Ce patron sépare la visualisation du résultat de son calcul et sépare également la visualisation de l'interaction utilisateur. Le rôle de chaque composant est bien défini :

- **Modèle :** calcul des données brutes en fonction de la configuration, cette partie correspond à la simulation.
- **Vue :** présentation des données du modèle pour l'affichage, réception des interactions utilisateur qui seront envoyées au Contrôleur. Cette partie correspond à la visualisation OpenGL.
- **Contrôleur :** traitement des interactions utilisateurs et mise à jour de la vue et/ou du modèle en fonction de l'action reçue. Cette partie correspond à l'interaction avec l'interface.

Le diagramme de ce patron est présenté sur la figure 10.1, les flèches pleines indiquent une indirection directe et les pointillés indiquent une association indirecte (une observation seule).

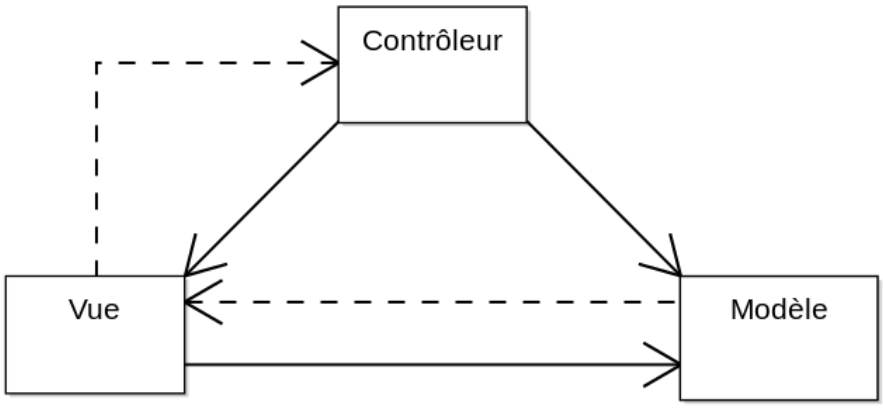


FIGURE 10.1 – Architecture Modèle-Vue-Contrôleur

Nous avons utilisé cette méthode pour réaliser avec succès la première implémentation du démonstrateur rassemblant visualisation et simulation.

10.3 Limites du MVC

Au fur et à mesure que des fonctionnalités de visualisation ou de simulation étaient rajoutées dans l'application, l'architecture précédente a montré des limites à trois niveaux :

- **Complexité ;** : les interactions entre les trois modules sont devenues complexes avec l'ajout de nombreuses fonctionnalités supplémentaires.
- **Réutilisabilité :** notre programme comportant trois modes : batch, comparaison et visualisation. Seul le troisième mode utilise le patron MVC mais ce patron influence l'architecture globale de la plupart des classes de l'application.
- **Modularité/Extensibilité :** rajouter des traitements sur les données (pré-traitement, post-traitement) est difficile puisqu'il faut s'intégrer dans plusieurs parties du MVC.

10.4 Architecture dataflow

10.4.1 Principe

Pour les raisons venant d'être énoncées, il a été décidé vers la fin du projet de changer l'architecture globale. Nous ne sommes pas repartis depuis le début puisque tous les algorithmes de calcul et de visualisation furent réutilisés, seule l'organisation des algorithmes entre eux a changé. Nous avons choisi une architecture basée sur le modèle de programmation par flots de données (**dataflow programming**). Cette architecture s'inspire de la philosophie UNIX (les données sont échangées par des pipelines) et donc de la première implémentation réalisée, mais au lieu de programmes exécutables nous assemblons des blocs de calculs. Les données à traiter sont modifiées successivement par ces blocs en suivant les connexions du graphe, d'où le nom de paradigme **dataflow**.

L'exécution du programme est modélisée sous la forme d'un graphe orienté représentant la chaîne d'opérations à appliquer sur les données. Les nœuds sont appelés **acteurs** et correspondent à un bloc effectuant un calcul précis. Les arcs du graphe connectent les acteurs entre eux et représentent le chemin emprunté par les données. Un nœud possède un **port** pour chaque connexion sortant auxquelles il est rattaché. Plusieurs acteurs connectés forment un **réseau**. Un réseau est également un acteur : on peut définir une hiérarchie de réseaux en utilisant cette fonctionnalité récursive (un réseau dont les acteurs sont aussi des réseaux).

10.4.2 Modularité et extensibilité

Un acteur est uniquement défini par son interface d'entrée et de sortie, c'est-à-dire par le type des connexions entrantes et des connexions sortantes. Par exemple l'acteur `SurfaceCompare` a deux ports d'entrée de type `Surface` et un port de sortie de type `Error` qui contient des statistiques sur la distance entre les deux surfaces. Seule l'interface étant importante, un acteur peut être remplacé par un autre acteur possédant la même interface de manière totalement transparente, et éventuellement dynamiquement (cela correspond au patron de conception **Stratégie**). De plus, la granularité de l'acteur peut être modifiée facilement. Un réseau étant également un acteur, si ce réseau possède la même interface d'entrées/sorties, on peut remplacer l'acteur par un réseau pour séparer plus finement les différentes étapes du calcul. Par exemple, la lecture d'un fichier surface peut être implémentée par un acteur ayant pour entrée un nom de fichier et comme sortie un objet `Surface`. On peut implémenter cette fonctionnalité avec un unique acteur ou en décomposant les différentes extensions dans un sous-réseau, voir diagramme 10.2.

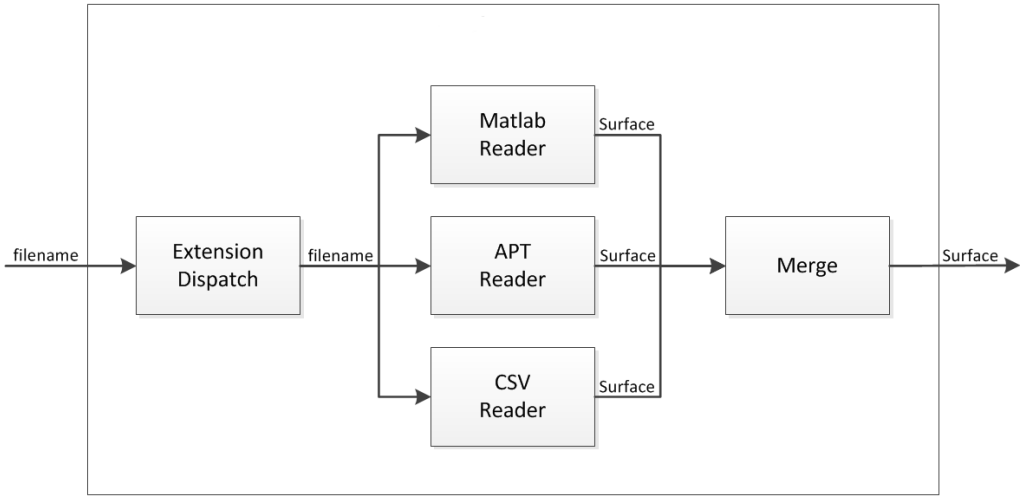


FIGURE 10.2 – Bloc de lecture d'une surface.

Autre avantage de cette méthode, mais qui pour le moment reste inexploité : le parallélisme naturel de l'architecture permet d'exécuter les acteurs en parallèle, seuls les accès aux connexions doivent être protégés par des verrous.

10.4.3 Exemples

En assemblant des acteurs d'entrée/sortie, des acteurs de prétraitement, un acteur de simulation et des acteurs de post-traitement, on peut très facilement créer un réseau pouvant exécuter une configuration en mode **batch** comme illustré sur le diagramme 10.3.

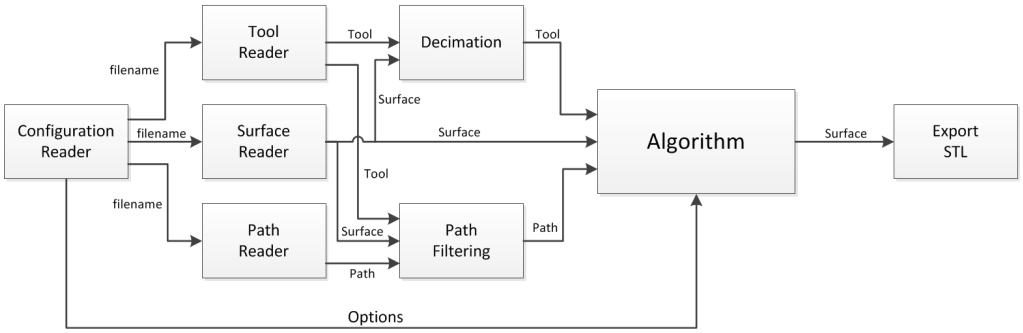


FIGURE 10.3 – Configuration batch.

En utilisant ce réseau comme un acteur, on peut assembler deux acteurs de type **batch** dans un nouveau réseau pour former un programme de comparaison de résultats, voir diagramme 10.4.

Développer un filtre et l'intégrer à la chaîne de traitement devient facile, il suffit de chaîner les filtres dans l'ordre dans lequel on souhaite les appliquer. On peut rajouter des traitements sur n'importe quelle connexion afin de pouvoir intégrer des acteurs de pré-traitement, post-traitement, **debug**, **log**, etc. Un exemple de chaîne de traitement est présenté sur le diagramme 10.5.

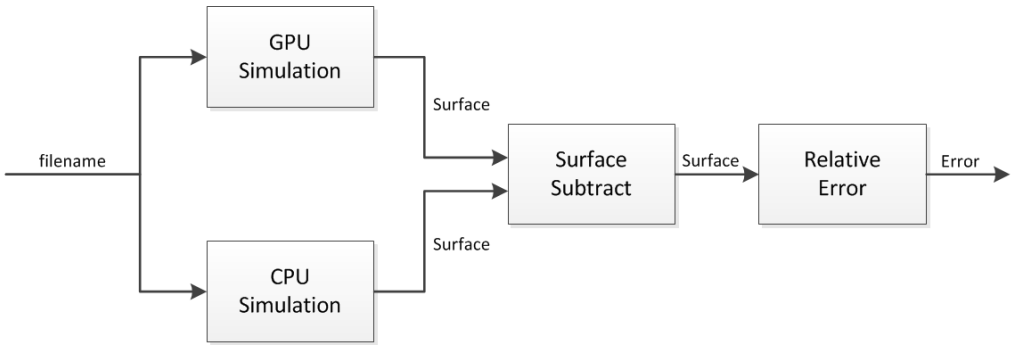


FIGURE 10.4 – Configuration de comparaison.

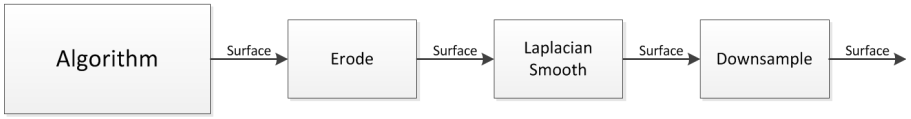


FIGURE 10.5 – Chaîne de traitement de postprocessing.

10.4.4 Implémentation

Un acteur réalisant un traitement doit dériver de la classe abstraite `Actor`. Les acteurs qui sont en réalité des réseaux doivent dériver de la classe abstraite `MultiActor`.

```
/*
  This is the base class for any computation block, it is called an actor.
  Children classes should override the two main methods: initialize and schedule.
*/
class Actor
{
public:
  /*
    Initialize the actor internally. For example construct helper
    objects and variables needed during processing.
    The initialize method is called on a top down fashion on the whole
    network. Actors are initialized recursively before firing any action
    of the network.
  */
  virtual void initialize()
    {
    }

  /*
    Try to fire the actor, return true if the actor was fired, false
    otherwise.  A typical action will generally consume data from input
    connections, process the data and produce modified data on the
    output connections.
    An actor can contain several actions, the choice of which action
    should be fired can depend on the availability of data from input connections,
    an internal Finite State Machine (FSM), a priority list, an UI state, etc.
  */
  virtual bool schedule()
    {
      return false;
    }
};
```

Un acteurs doit permettre un accès à ses connexions entrantes et sortantes. Les connexions entrantes sont de type `InputConnection` et les connexions sortantes sont de type `OutputConnection`. Ces deux types dérivent du type abstrait `Connection`. Une connexion est initialisée entre deux acteurs en utilisant la fonction `connect`. Voici un exemple de réseau, celui corres-

pondant à une exécution d'une configuration en mode batch :

```
class NetworkBatch: public MultiActor
{
public:
  // Name of the configuration file.
  InputConnection<const char*> config_input;
  // Name of the requested engine.
  InputConnection<const char*> engine_input;
public:
  // Result of simulation.
  OutputConnection<Surface*> output;
public:
  NetworkBatch()
    {
      add_actor(config_reader_);
      add_actor(factory_);
      add_actor(execute_);
    }

  void initialize_connections()
    {
      // Set local connections.
      config_reader_.input = config_input;
      factory_.input = engine_input;
      execute_.output = output;

      // Set inner connections.
      connect(config_reader_.path_output, execute_.path_input);
      connect(config_reader_.surf_output, execute_.surf_input);
      connect(config_reader_.tool_output, execute_.tool_input);

      connect(factory_.output, execute_.engine_input);
    }
private:
  // Read the files specified in a configuration file.
  ConfigReader config_reader_;
  // Output an Engine* from an engine name.
  EngineFactory factory_;
  // Actor executing an engine on a given configuration.
  EngineExecute execute_;
};
```

10.5 Représentation configuration

Nous allons expliquer brièvement comment sont représentés les éléments nécessaires à l'exécution de la simulation : l'outil, la surface et la trajectoire. Pour chacune des classes que nous allons présenter, il existe une classe équivalente CUDA. Les hiérarchies n'ont aucun lien entre elles pour contourner des problèmes rencontrés lors de la compilation.

10.5.1 Outil

Toutes les classes sont implémentées dans le fichier `tool.hh`. Tous les types d'outils dérivent de la classe abstraite `Tool`. Cette classe déclare une seule méthode (virtuelle pure) : `bounding_box` qui renvoie la boîte englobante 3D de l'outil. Les classes concrètes dérivant de `Tool` sont :

- `MeshTool` : outil sous forme d'un maillage de triangle, possède une interface similaire à un vecteur STL pour accéder aux triangles.
- `SphereTool` : primitive géométrique sphère définie par un rayon.
- `TorusTool` : primitive géométrique tore définie par un petit rayon et un grand rayon.
- `ConicTool` : primitive géométrique cône définie par une hauteur et un rayon de base.

10.5.2 Surface

Toutes les classes sont implémentées dans le fichier `surface.hh`. La classe abstraite `Surface` comporte de nombreux accesseurs afin d'obtenir les dimensions de la surface : hauteur, largeur, pas, domaine, centre... Un template de classe `SurfaceTemplate<T>` dérive de `Surface`. Le type `T` est la précision utilisée : `float` ou `double`. Nous obtenons les types `Surface32` et `Surface64`.

10.5.3 Trajectoire

Toutes les classes sont implémentées dans le fichier `path.hh`. Comme pour l'implémentation de la Surface nous avons un template de classe `PathTemplate<T>` dérivant de la classe abstraite `Path`. Le type `PathTemplate<T>` a une interface similaire à un vecteur STL pour permettre l'accès à une position par son indice. Le type `T` est le type d'une position : `PathTemplate<float3>` est une trajectoire 3 axes et `PathTemplate<float6>` est une trajectoire 5 axes.

Chapitre 11

Moteurs de simulation

Comme nous l'avons souligné précédemment, l'algorithme Z-buffer peut présenter de nombreuses variations. Nous avons présenté comment utiliser les fonctionnalités du C++ afin de personnaliser certains comportements du squelette d'algorithme initial. Nous allons montrer comment les différentes versions de l'algorithme sont intégrées dans le projet.

Chaque algorithme est intégré dans une classe C++ appelée **Engine**. Il s'agit d'une classe abstraite permettant de regrouper des fonctionnalités communes pour initialiser et exécuter les algorithmes. Voici l'interface de la classe **Engine** :

```
class Engine
{
public:
  Engine();

  virtual ~Engine();

  // Retrieve the name of the engine.
  virtual const char* name() = 0;

  // Return 'true' if the engine can execute the given configuration,
  virtual bool can_compute(Path* path, Surface* surf, Tool* tool) = 0;

  // Initializa the engine for the given configuration.
  void set_configuration(Path* path, Surface* surf, Tool* tool);

  // Fill the given execution timer.
  void execution_timer(EngineTimer& timer) const;

  // Launch execution and wait for its termination.
  virtual void launch() = 0;

  // Launch execution and return immediately.
  virtual void launch_async() = 0;

  // Wait for the end of the execution.
  virtual void sync() = 0;

  // Called before 'compute'.
  virtual void initialize() = 0;

  // Launch the wrapped algorithm.
  virtual void compute() = 0;

  // Called after 'compute'.
  virtual void finalize() = 0;

  // Return 'true' if the execution has ended.
  virtual bool has_ended() = 0;

  // Retrieve the resulting surface after the execution.
```

```
  virtual Surface* result() = 0;
protected:
  Path* path_;
  Surface* surf_;
  Tool* tool_;
  EngineTimer engine_timer_;
};
```

Toutes les méthodes déclarées sont abstraites, les méthodes de configuration de l'exécution sont implémentées dans les deux classes abstraites **CPUEngine** et **GPUEngine**. Sur CPU l'asynchronisme est géré par les **pthread** alors que sous GPU il est géré par l'API CUDA. Le tableau suivant montre comment ces opérations sont implémentées sur CPU et sur GPU :

	CPUEngine	GPUEngine
launch_async()	`pthread_create`	`kernel<<<...>>>`
sync()	`pthread_join`	`cudaStreamSynchronize`
has_ended()	`pthread_kill(t, 0) == ESRCH`	`cudaStreamQuery`

Les méthodes `initialize` et `finalize` sont également implémentées dans ces deux classes. Ces méthodes initialisent et démarrent des **timers** pour mesurer le temps d'exécution. Sur CPU le temps est mesuré avec `gettimeofday` (fonctionne pour une exécution parallèle) et sur GPU avec la fonction `cudaEventElapsedTime`.

Les méthodes `can_compute` et `compute` sont implémentées dans les classes feuilles de la hiérarchie **Engine**. Ce sont les seules classes concrètes de la hiérarchie. Chaque classe encapsule une des variantes de l'algorithme que nous avons décrit, par exemple voici une liste de quelques classes implémentées :

- **CPURasterEngine**
- **GPURasterEngine**
- **GPUInvertedEngine**
- **GPUFineEngine**
- **GPUTorusEngine**
- **GPUSphereEngine**
- **CPUSphereEngineSSE**

Un extrait de la hiérarchie Engine est présenté dans le diagramme UML 11.1.

La méthode `can_compute` teste le type dynamique de chaque paramètre, afin de vérifier si l'algorithme peut traiter cette configuration. Voici par exemple l'implémentation de `cpu_torus_engine`, cet algorithme ne s'applique que sur une configuration 3-axes avec un outil torique.

```
  return is_type<TorusTool>(tool) && is_type<Path3>(path);
```

11.1 Engine Execute

Le bloc **Engine Execute** est utilisé pour le mode visualisation. Lorsque ce bloc reçoit une nouvelle configuration, chaque moteur de simulation qu'il connaît est interrogé pour déterminer si il peut exécuter cette configuration. La liste des moteurs pouvant exécuter cette configuration sont ensuite proposés à l'utilisateur sous la forme d'une liste de sélection dans l'interface. L'utilisation de la hiérarchie **Engine** permet ainsi de factoriser les codes de configuration CPU et GPU et permet de filtrer les algorithmes ne s'appliquant pas à la configuration actuelle. L'ordre de déclaration de ces moteurs permet d'établir

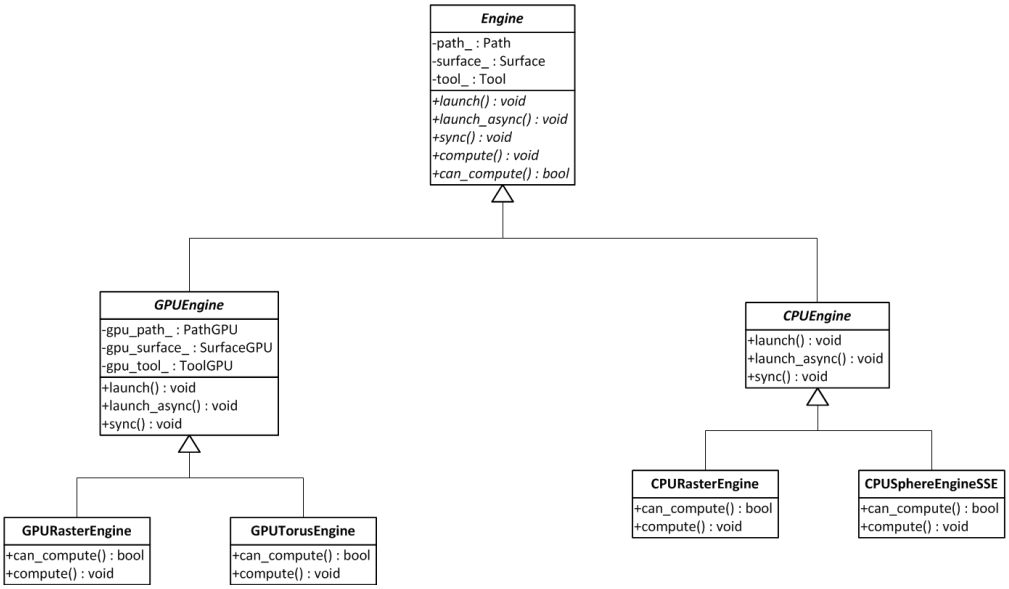


FIGURE 11.1 – Hiérarchie UML de Engine.

un ordre de préférence sur lequel devrait être utilisé, par exemple dans le cas où plusieurs algorithmes peuvent s'appliquer à une configuration, mais avec des performances différentes.

11.2 Ajout d'un algorithme

Un utilisateur souhaitant ajouter un algorithme dans le projet n'a pas besoin de comprendre son architecture dans tous ses détails. Pour ajouter un algorithme dans SIMSURF, deux cas de figure peuvent se présenter.

11.2.1 Extension d'un moteur existant

Un algorithme similaire existe déjà dans un moteur de simulation, mais nous souhaitons l'étendre pour traiter plus de configurations différentes. Par exemple actuellement l'algorithme de simulation utilisant une primitive géométrique tore ne peut traiter que des trajectoires de type 3 axes, ce moteur de simulation pourrait être étendu pour gérer les trajectoires 5 axes. Ce cas de figure demande de modifier un seul fichier : le fichier source du moteur. Si le nouvel algorithme utilise des templates C++, son implémentation pourra être placée dans le fichier `nbuffer.hh` afin d'éviter la duplication entre les moteurs. La méthode `can_compute` doit être modifiée pour refléter le changement de la liste des configurations acceptées. Dans notre cas nous devons remplacer le code initial :

```
return is_type<TorusTool>(tool) && is_type<Path3>(path) && is_type<Surface32>(
    surf);
```

Par le code :

```
return is_type<TorusTool>(tool) && (is_type<Path3>(path) || is_type<Path6>(path))
    && is_type<Surface32>(surf);
```

Concernant la méthode `compute`, nous devons utiliser la fonctionnalité `dynamic_cast` du C++ pour tester le type dynamique de la trajectoire et utiliser le bon algorithme. Le code initial :

```
void CPUTorusEngine::compute()
{
  TorusTool* tool = dynamic_cast<TorusTool*>(tool_);
  Surface32* surf = dynamic_cast<Surface32*>(surf_);

  Path3* path3 = dynamic_cast<Path3*>(path_);
  if (path3)
  {
    zbuffer_torus_3(path3, surf, tool);
    return;
  }
}
```

Est remplacé par le code :

```
void CPUTorusEngine::compute()
{
  TorusTool* tool = dynamic_cast<TorusTool*>(tool_);
  Surface32* surf = dynamic_cast<Surface32*>(surf_);

  Path3* path3 = dynamic_cast<Path3*>(path_);
  if (path3)
  {
    zbuffer_torus_3axis(path3, surf, tool);
    return;
  }
  Path6* path6 = dynamic_cast<Path6*>(path_);
  if (path6)
  {
    zbuffer_torus_5axis(path6, surf, tool);
    return;
  }
}
```

Aucun autre changement n'est nécessaire. Le moteur est maintenant enrichi pour traiter de nouvelles configurations. Il sera automatiquement proposé dans l'interface si la configuration utilisée est compatible.

11.2.2 Ajout d'un nouveau moteur

Un nouveau moteur peut être ajouté en suivant ces étapes :

- Création d'un couple de fichiers `.hh/.cc` pour un moteur CPU ou `.hh/.cu` pour un moteur GPU (le compilateur appelé dépend de l'extension du fichier).
- Écriture du fichier `.hh` : implémentation d'une classe dérivant de `CPUEngine` ou `GPUEngine` et attribution d'un nom.
- Écriture du fichier `.cc` ou `.cu` : implémentation de la méthode `can_compute` pour déterminant si une configuration est exécutable et de la méthode `compute` qui exécute l'algorithme de simulation.
- Ajout du nouveau moteur dans le constructeur de l'acteur listant les différents moteurs en mode visualisation (`EngineDispatch`).
- Ajout du nouveau moteur dans l'acteur `EngineFactory`, utilisée pour instancier un moteur en fonction du nom demandé.

Après ces étapes le nouvel algorithme implémenté peut être utilisé dans les trois modes : visualisation, comparaison et batch.

Chapitre 12

Optimisations

12.1 Représentation outil

Les outils utilisés pour l'usinage peuvent être d'une forme quelconque et même présenter des défauts à certains endroits. Pour modéliser ces défauts, les outils sont représentés par un maillage de triangles. Si l'on souhaite modéliser finement la géométrie de l'outil, plusieurs milliers de triangles sont nécessaires. Une telle précision n'est pas toujours nécessaire si l'on souhaite seulement obtenir l'allure générale de la surface (contexte macrogéométrique), au contraire cette précision sera bienvenue dans le cas d'un zoom sur une petite partie de la surface (contexte microgéométrique). Dans un contexte macrogéométrique, un nombre élevé de triangles augmente le temps de calcul sans forcément améliorer la précision des résultats. Pour remédier à ce problème et avoir des temps de calcul plus rapides, nous avons deux solutions :

- **Décimation du maillage outil :** le maillage outil est simplifié pour diminuer le nombre de triangles. Plusieurs algorithmes de simplification existent, c'est souvent la fusion d'une paire de sommets qui est réalisée (**edge collapse**). Si la simplification est de qualité, le nombre de triangles peut diminuer drastiquement tout en ayant seulement une faible perte de précision.
- **Utilisation d'une primitive géométrique :** plusieurs types d'outil peuvent s'apparenter à une primitive géométrique idéale : une sphère, un tore ou un cône. Si la modélisation des défauts de l'outil n'est pas l'objectif, cet algorithme obtiendra des résultats plus précis que la version maillée car nous utilisons l'expression analytique de l'objet géométrique.

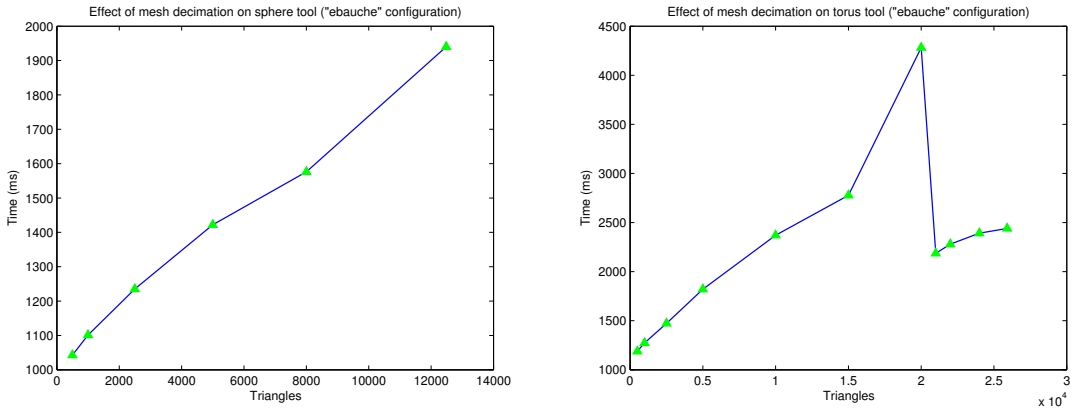
Nous allons maintenant étudier les deux stratégies séparément.

12.1.1 Simplification de maillage

Implémenter un algorithme de simplification de maillage est complexe si l'on veut obtenir de bons résultats (c'est-à-dire limiter la perte de précision) dans un temps acceptable. La simplification étant juste une simple technique pour diminuer les performances, son implémentation n'était pas prioritaire dans le projet. Nous avons décidé d'utiliser une implémentation disponible en licence libre sur internet. Une phase d'essai a éliminé de nombreuses implémentations trop lentes ou générant un maillage de mauvaise qualité, nous avons retenu l'implémentation de la bibliothèque **VCGLib**, cette bibliothèque est à la base de l'outil de manipulation et de visualisation de maillages **Meshlab**.

L'application implémente un acteur qui appelle un programme externe compilé préalablement. L'appel à ce programme est effectué avec la méthode classique **fork-exec** sous

Unix. L'impact de la simplification sur les performances dépend de la forme initiale de l'outil, dans le cas général le temps de calcul décroît linéairement avec le nombre de triangles. Cependant, dans certains cas particuliers, l'algorithme de simplification donne de mauvais résultats, le maillage devient pathologique et peu adapté à notre méthode calcul. Les deux cas de figure sont présentés sur le graphique 12.1, avec un outil sphérique le gain est linéaire, avec un outil torique le temps de calcul est soumis à un pic vers 20000 triangles.



(a) Sphere (b) Tore

FIGURE 12.1 – Performances avec simplification.

La géométrie du maillage simplifié introduisant le pic est comparée avec le maillage initial de l'outil dans la figure 12.2. La portion du maillage considérée correspond à la partie plane de l'outil torique (la partie la plus basse). Un tel maillage crée de nombreux fragments de grande taille (surtout pour les triangles en diagonale) pour peu d'intersections réelles. Le temps de calcul sera limité par le temps de calcul du fragment le plus long.

En conclusion, la simplification de maillage peut améliorer les performances de l'algorithme de simulation si la précision n'est pas prioritaire. L'amélioration des performances dépend de la qualité de l'algorithme de simulation.

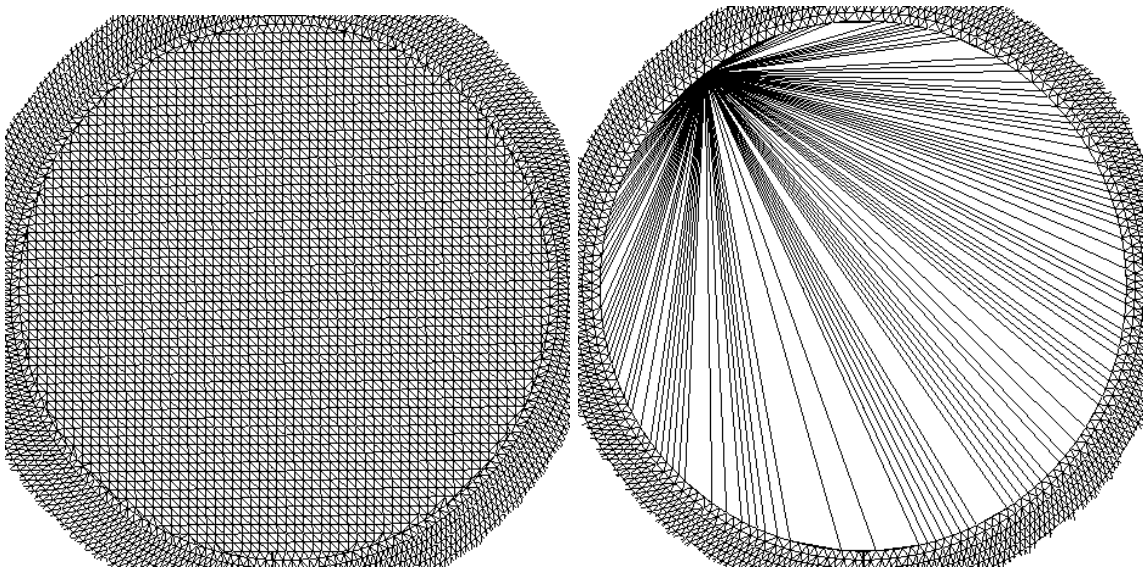
12.1.2 Primitive géométrique

Dans le fichier de configuration d'entrée, l'utilisateur peut demander l'utilisation d'une primitive géométrique au lieu d'un fichier de maillage. Par exemple `SPHERE 5.3` utilise une sphère de rayon 5.3, `CONE 1.3 3` utilise un cône de rayon 1.1 et de hauteur 3. Le principe de l'algorithme reste identique mais en appliquant la rastérisation sur ces nouvelles primitives géométriques au lieu d'un triangle. Nous avons donc un seul fragment à rastériser par position outil au lieu d'un fragment pour chaque triangle du maillage. En contrepartie le fragment est plus grand : la granularité de la rastérisation est augmentée.

Pour l'outil torique, sphérique et conique en usinage 3-axes, le fragment se calcule de la même façon puisque la projection de ces primitives sur la surface est un cercle, voir figure 12.3.

Le calcul de ce fragment est plus simple que pour le triangle puisque seul le rayon externe est nécessaire pour le calcul de la boîte englobante 2D.

Le calcul du test d'intersection et de distance est effectué en utilisant la représentation 2D de la primitive géométrique projetée. Par exemple pour la sphère, la distance d'inter-



(a) Initial (b) Simplifié

FIGURE 12.2 – Maillage pathologique après simplification.

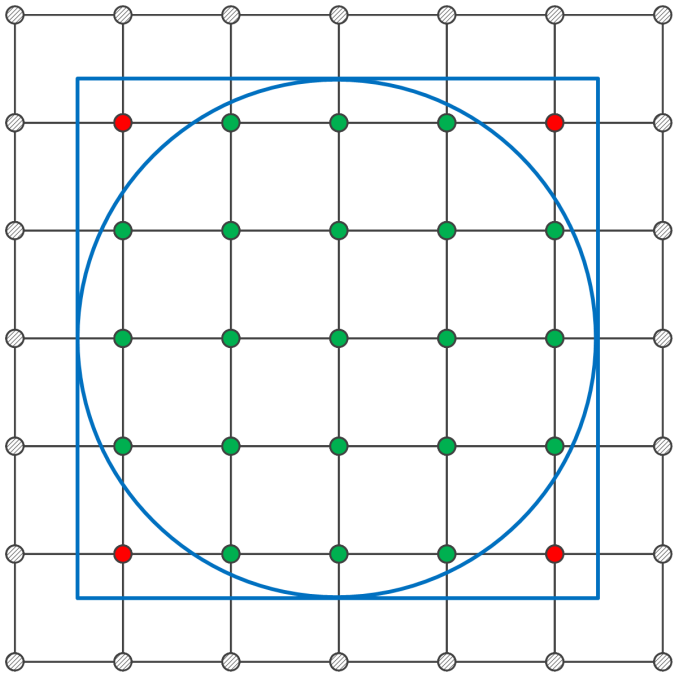


FIGURE 12.3 – Rastérisation d'une primitive sphérique.

section est calculée avec un simple calcul géométrique en connaissant la distance entre le brin et le centre de la sphère (la position outil). Ce procédé est illustré sur le schéma 12.4. Cette méthode peut être facilement étendue pour le les outils coniques et toriques.

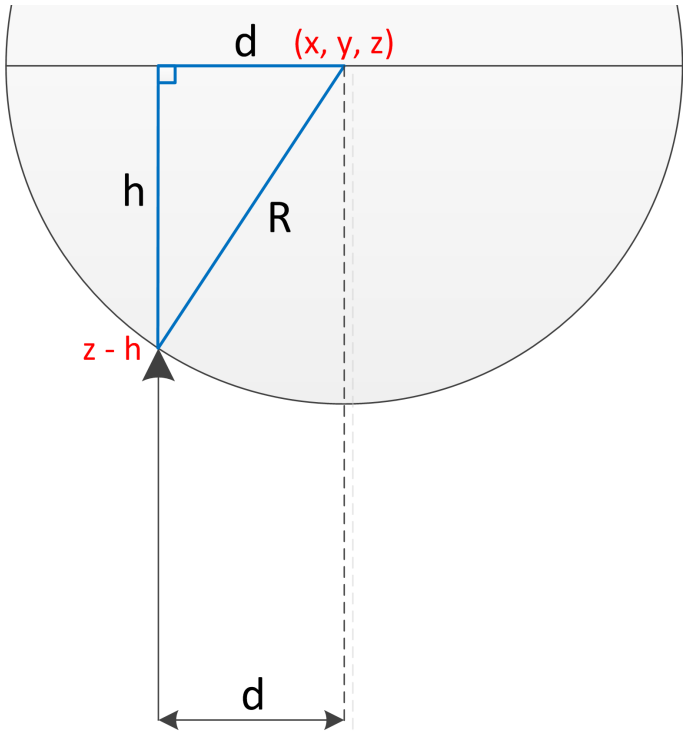
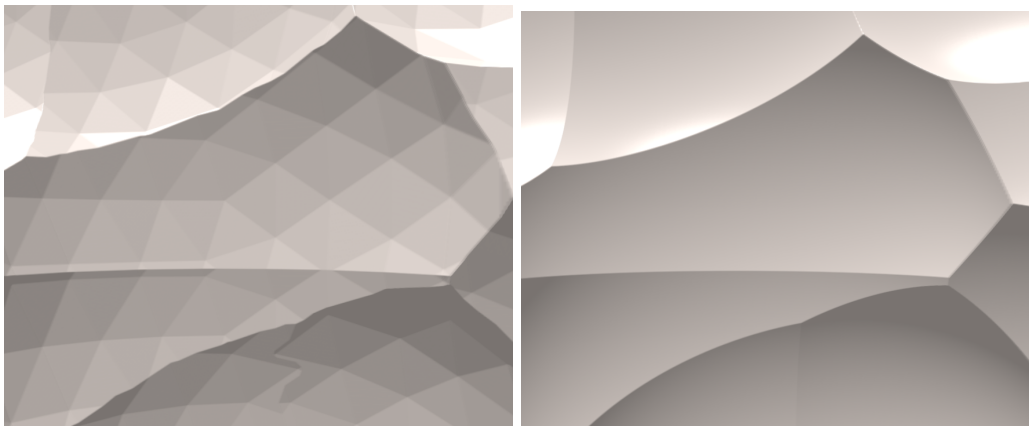


FIGURE 12.4 – Calcul de l'intersection avec une sphère.

Si l'on utilise un maillage pour représenter une sphère, les triangles seront visibles si il est trop grossier pour la résolution choisie. Par exemple la figure 12.5 illustre la différence entre une petite portion de surface simulée avec un maillage à gauche et la primitive géométrique sphère à droite.



(a) Maillage (b) Primitive

FIGURE 12.5 – Aspect de la surface après plusieurs zooms.

Le tableau suivant illustre la différence de temps de calcul entre les deux implémentations. Nous avons comparé les implémentations CPU et GPU utilisant la primitive

géométrique avec les implémentations utilisant le maillage de triangles. Nous avons realisé les tests sur les deux configurations présentées. Pour chacune des 5 configurations, la première ligne du tableau est l'implémentation avec maillage et la seconde ligne est l'implémentation en utilisant une primitive.

Path		Tool		CPU		GPU	
Nom	Taille	Nom	Taille	Xeon X5650	Phenom 955	Quadro 4000	GTX 560 Ti
spiral	200000	sphere	1984	1050	4634	247	105
spiral	200000	**primitive**	✖	34	76	11	9
volcano	200000	sphere	1984	1071	4584	199	85
volcano	200000	**primitive**	✖	44	82	13	11
downspiral	200000	cone	64	270	792	48	25
downspiral	200000	**primitive**	✖	34	92	29	18
ebauche	47837	tore	25904	7791	25160	1546	712
ebauche	47837	**primitive**	✖	2670	4327	665	333
finition	53667	sphere	12482	10542	30603	2361	1187
finition	53667	**primitive**	✖	3094	4228	2064	1055

Pour les trois premières configurations, la taille de l'outil est petite relativement à la taille de la surface (boîte englobante de taille 10 pour une surface de taille 1024). Pour les deux dernières configurations, le diamètre de l'outil est de 10 pour une surface de taille 50, les fragments sont par conséquent beaucoup plus larges, un nombre important de brins doivent être testés pour chaque position. Cette observation explique le faible gain sur GPU pour les deux dernières configurations en utilisant l'implémentation avec primitive : la granularité de chaque tâche devient trop importante comme nous l'avons expliqué précédemment, trop de brins sont testés par thread. La division d'une primitive géométrique en un maillage de triangles est au final un moyen naturel pour diminuer la granularité des tâches de la simulation.

Sur CPU, une implémentation de l'algorithme utilisant la primitive sphère a été réalisée en utilisant des instructions SSE. 4 brins consécutifs sont testés à la fois grâce aux instructions SIMD. Les résultats sont présentés dans le tableau suivant. Lorsque l'outil est de petite taille, les fragments sont trop petits pour pouvoir utiliser les instructions SSE suffisamment souvent. Au contraire si l'outil est grand, le gain de vectorisation est appréciable.

Path		Tool	CPU			
Nom	Taille	Nom	Xeon X5650	avec SSE	Phenom 955	avec SSE
spiral	200000	**primitive**	34	58	76	71
volcano	200000	**primitive**	44	60	82	75
finition	53667	**primitive**	3094	1207	4228	2114

La vectorisation SSE a été implémentée sur l'algorithme de rastérisation de triangles, mais le code généré était environ 2 fois plus lent que le code scalaire.

12.2 Configuration optimale CUDA

En plus de la stratégie choisie et de l'optimisation bas niveau du kernel il est nécessaire de choisir la configuration optimale de lancement afin d'atteindre les meilleures performances sur l'architecture CUDA ciblée.

Nous avons vu que des ensembles de threads sont rassemblés en blocs et que ces blocs forment une grille à dimensions. Les threads CUDA s'exécutent de façon massivement parallèle par groupes de warps et ces groupes de warps sont exécutés par ce que l'on appelle un **multiprocesseur**. Par exemple sur la carte Quadro 4000 un multiprocesseur peut héberger jusqu'à 48 warps en même temps soit $48 \times 32 = 1536$ threads. Pour remplir la capacité d'un multiprocesseur, l'ordonnanceur CUDA affecte plusieurs blocs de threads à un multiprocesseur. Le nombre de blocs pouvant être affectés au maximum à un multiprocesseur dépend de plusieurs facteurs. Pour obtenir les meilleures performances possible, le nombre de threads par blocs doit être choisi en conséquence sinon des warps resteront inutilisés sur chaque multiprocesseur.

12.2.1 Limitation par nombre de blocs

Un nombre limité de blocs pouvant être affectés à un multiprocesseur (8 pour un accélérateur de type 2.0). Ce facteur est rarement limitant, ce cas n'intervient que lorsque les threads utilisent très peu de registres.

12.2.2 Limitation par nombre de registres

Si les threads utilisent trop de registres, l'affectation est également limitée par la taille du banc de registres du multiprocesseur (32 Ko sur les deux GPUs utilisés).

12.2.3 Limitation par mémoire partagée

Les multiprocesseurs ont aussi une limite de quantité de mémoire partagée pouvant être allouée lors de l'exécution. Cette limite dépend de la configuration choisie avant le lancement du kernel. 64 Ko de mémoire sont disponibles sur le multiprocesseur et sont partagés entre le cache L1 et la mémoire partagée. Pour un accélérateur de type 2.0 il est nécessaire de favoriser l'un ou l'autre durant le partage (un accélérateur 3.0 permet d'effectuer un partage équitable), soit 48 Ko et 16 Ko.

Quelle que soit la configuration choisie, la somme de la mémoire partagée utilisée par tous les blocs d'un même multiprocesseur ne peut pas dépasser la configuration fixée.

12.2.4 Détermination du facteur limitant

Pour obtenir la meilleure occupation (occupancy) des multiprocesseurs il est donc nécessaire de prendre en compte chacun de ces paramètres. Un de ces facteurs sera limitant et donc le nombre optimal de threads par bloc en dépend. La feuille de calcul fournie par NVIDIA (**CUDA Occupancy Calculator**) permet de calculer la configuration optimale de lancement en fonction de la **compute capability** de la carte graphique, du nombre de registres utilisés et de la mémoire partagée utilisée. Le seul paramètre à fixer est le nombre de threads par bloc, le nombre de blocs nécessaire peut ensuite être déduit de ce paramètre, quant à l'organisation de la grille elle n'a pas d'impact sur l'ordonnancement des threads.

Nous avons vu que le nombre optimal de threads par bloc dépend des capacités de l'architecture cible. De plus, tout changement dans le kernel peut impacter le nombre de registres utilisés donc la configuration. Il est donc préférable de déterminer dynamiquement la configuration de lancement du kernel. On peut pour cela répliquer les calculs effectués par l'**Occupancy Calculator** de NVIDIA pour trouver la configuration optimale. Les formules suivantes viennent soit de la feuille de calcul soit de NVIDIA (2012).

Tout d'abord nous devons calculer le nombre de warps dans un bloc W_{block} :

$$W_{block} = \text{ceil}(\frac{T}{W_{size}}, 1) \tag{12.1}$$

Avec :
- T : le nombre de threads par bloc.
- W_{size} : la taille d'un warp, soit 32.
- $\text{ceil}(x, y)$ calcule x arrondi au multiple supérieur le plus proche de y.

On peut calculer le nombre maximum de blocs par multiprocesseur dû au nombre de warps par bloc B_W :

$$B_W = \max(\text{ceil}(\frac{W_{max}}{W_{block}}, 1), B_{max}) \tag{12.2}$$

Avec :
- W_{max} : le nombre maximal de warps par multiprocesseur.
- B_{max} : le nombre maximal de blocs par multiprocesseur.

Pour un accélérateur de version 2.0 ou supérieur, l'allocation de registres s'effectue à la granularité du warp, le nombre de registres alloués pour un bloc R_{block} est donc :

$$R_{block} = \text{ceil}(R_k \times W_{size}, G_R) \times W_{block} \tag{12.3}$$

Avec :
- R_k : le nombre de registres utilisé par une instance du kernel.
- G_R : la granularité de l'allocation de registres.

On peut ensuite calculer le nombre maximum de blocs par multiprocesseur imposé par le nombre de registres B_R :

$$B_R = \max(\text{ceil}(\frac{R_{max}}{R_{block}}, 1), B_{max}) \tag{12.4}$$

Avec :
- R_{max} : le nombre maximal de registres par multiprocesseur.

Le nombre effectif de blocs B_{lim} pouvant être attribué à un multiprocesseur est donné par le facteur limitant :

$$B_{lim} = \min(B_W, B_R) \tag{12.5}$$

12.2.5 Détermination de la configuration optimale

Dans notre cas nous voulons à déterminer le nombre optimal de threads par bloc en connaissant le nombre de registres utilisés par le kernel. On souhaite à maximiser l'occupation O de chaque multiprocesseur c'est-à-dire le nombre de warps actifs sur le nombre de warps maximum :

$$O = \frac{B_{lim} \times W_{block}}{W_{max}} \tag{12.6}$$

On cherche donc la valeur de T (le nombre de threads) maximisant la valeur de O, on appellera cette valeur T_{opt} :

$$T_{opt} = \underset{T}{\text{argmax}}\, O \tag{12.7}$$

12.3 Exemple

Considérons le cas d'un kernel utilisant 29 registres par thread et avec 256 threads par bloc :

- $T = 256$
- $R_k = 29$

La Quadro 4000 est de computing capability 2.0 et possède donc les caractéristiques architecturales suivantes :

- $W_{size} = 32$
- $W_{max} = 48$
- $B_{max} = 8$
- $G_R = 128$

Le nombre de warps par bloc est :

$$W_{block} = \text{ceil}(\frac{256}{32}, 1) = 8 \tag{12.8}$$

Le nombre maximum de blocs par multiprocesseur induit par la taille d'un bloc est :

$$B_W = \max(\text{ceil}(\frac{48}{8}, 1), 8) = 6 \tag{12.9}$$

Le nombre de registres par bloc est :

$$R_{block} = \text{ceil}(29 \times 32, 128) \times 8 = 8192 \tag{12.10}$$

Ainsi le nombre maximum de blocs par multiprocesseur induit par la quantité de registres est :

$$B_R = \max(\text{ceil}(\frac{32768}{8192}, 1), 8) = 4 \tag{12.11}$$

Le nombre de blocs maximum par multiprocesseur est alors :

$$B_{lim} = \min(B_W, B_R) = 4 \tag{12.12}$$

L'occupation d'un multiprocesseur est donc :

$$O = \frac{4 \times 8}{48} \approx 67\% \tag{12.13}$$

Chapitre 13

Évaluation des performances

13.1 Automatisation

Pour évaluer les performances de plusieurs algorithmes ou d'optimisations spécifiques, nous avions besoin d'un outil permettant d'automatiser les benchmarks de notre application. Dans cette optique un script Python a été développé. Ce script permet de lancer A algorithmes sur N différentes configurations puis d'agréger les résultats sous forme de tableau LaTeX. Le benchmark est configuré par l'intermédiaire d'un fichier au format JSON. Ce fichier contient les différentes configurations de simulation et la liste des algorithmes devant être utilisés. Le script applique chaque algorithme à chaque configuration et mesure le temps d'exécution. Afin d'obtenir des résultats plus fiables, chaque exécution est répétée N fois, la moyenne du temps d'exécution est alors calculée. La valeur de N est spécifiée dans le fichier JSON.

Afin de ne pas perdre de performances GPU à cause de l'affichage graphique, ce script est lancé en mode `TTY` après avoir arrêté le serveur X. Chaque configuration est traitée en lançant une instance du processus SIMSURF en mode `batch`.

13.1.1 Benchmark GPU

Avant exécution d'un kernel, le runtime CUDA (`CUDART`) doit être initialisé. Cette initialisation ne peut pas être réalisée manuellement et est effectuée lors du premier appel à une fonction CUDA, par exemple un appel à un kernel ou un appel à `cudaMalloc`. Ce comportement est problématique pour les benchmarks puisque le premier lancement sera plus long. Pour contourner ce problème, un kernel vide est lancé sur un seul thread au début du programme.

```
__global__
void warmup_kernel()
{
}
```

Le runtime CUDA est maintenant initialisé avant l'exécution de l'algorithme et le temps d'exécution des instances suivantes de l'algorithme devrait être plus stable.

13.1.2 Benchmark CPU

Afin de s'assurer que les données de la simulation ne sont pas présentes en cache au lancement de l'exécution, le cache CPU est vidé lors de l'initialisation. Vider le cache CPU est très simple : il suffit de réaliser une copie mémoire de très grande envergure :

```
void flush_cache()
{
  static unsigned length = 1 << 26;
  static char* input = new char[length];
  static char* output = new char[length];

  memcpy(output, input, length);
}
```

Avec cet ajout, le temps d'exécution des algorithmes sur CPU s'est stabilisé : la variance entre les différents lancements est devenue plus faible.

13.2 Résulats

Nous allons comparer les temps de calculs des différents moteurs de simulation dans leur état final (avec toutes les optimisations activées).

13.2.1 Configurations

Nous avons utilisé différents couples de trajectoire/outil. Les configurations ébauche et finition correspondent à 2 cas d'usinage réels issus de CATIA alors que les autres configurations ont été générées manuellement. Voici une brève description des configurations utilisées :

- **single :** une seule position au centre de la surface. Le diamètre de l'outil est à peine plus petit que la surface.
- **random :** des positions aléatoires sur toute la surface. L'outil est petit par rapport aux dimensions de la surface (100 fois plus petit).
- **spiral :** une spirale démarrant au milieu de la surface. L'outil est petit par rapport aux dimensions de la surface.
- **downspiral :** pareil que la configuration précédente mais la hauteur diminue au fil de la spirale.
- **volcano :** la surface résulat ressemble à un volcan aplati. L'usinage est realisé en effectuant un balayage complet de la surface. L'outil est petit par rapport aux dimensions de la surface.
- **ebauche :** phase d'ébauche lors de l'usinage d'une pièce. Le diamètre de l'outil utilisé (outil torique) est 5 fois plus petit que la surface totale.
- **finition :** la phase de finition appliquée après l'ébauche, il s'agit d'un usinage 5 axes. L'outil est sphérique et de même dimensions que l'outil torique précédent.

Pour certains configurations nous avons ajouté un test après un zoom important sur une portion de la surface. Pour les premières configuration le zoom est de 64 fois, pour la configuration d'ébauche le zoom est de 8 fois. Les gains de performances sont inversés par rapport à la configuration initiale, nous expliquons pourquoi dans la partie **Interprétation**.

Les moteurs de simulation utilisés sont :

- **CPU Raster** : implémentation multicore de l'algorithme Z-buffer avec l'algorithme de rastérisation pour calculer les intersections brins/triangle.
- **GPU Raster** : kernel CUDA équivalent à l'implémentation précédente. Un thread traite tous les triangles dans une position donnée.
- **GPU Inverted** : un thread CUDA traite un triangle dans toutes les positions.
- **GPU Fine** : un thread CUDA traite un triangle dans une position seulement.

13.2.2 Benchmark Configuration ENS

Les temps sont donnés en millisecondes (ms).

Path		Tool					
Nom	Taille	Nom	Taille	CPU Raster	GPU Raster	GPU Inverted	GPU Fine
single	1	sphere	1984	20	848	8	8
random	200000	sphere	320	366	69	884	124
zoom	135	sphere	320	168	1211	739	54
spiral	200000	sphere	1984	1061	247	370	515
zoom	191	sphere	1984	321	161	199	69
downspiral	200000	cone	64	280	48	3082	100
zoom	201	cone	64	499	2310	6851	142
volcano	200000	sphere	1984	1081	199	384	514
zoom	121	sphere	1984	148	1449	139	51
ebauche	47837	tore	25904	7891	1548	1555	4652
zoom	619	tore	25904	943	3473	241	294
finition	53667	sphere	12482	11542	2361	✖	2722

13.2.3 Benchmark Configuration Personnelle

Les temps sont donnés en millisecondes (ms).

Path		Tool					
Nom	Taille	Nom	Taille	CPU Raster	GPU Raster	GPU Inverted	GPU Fine
single	1	sphere	1984	30	498	7	7
random	200000	sphere	320	1260	48	507	56
zoom	135	sphere	320	264	757	439	28
spiral	200000	sphere	1984	4634	105	179	227
zoom	191	sphere	1984	557	997	117	33
downspiral	200000	cone	64	792	25	1786	57
zoom	201	cone	64	818	1432	3948	68
volcano	200000	sphere	1984	4584	85	180	226
zoom	121	sphere	1984	369	901	83	25
ebauche	47837	tore	25904	25160	712	698	2390
zoom	619	tore	25904	1857	2080	118	151
finition	53667	sphere	12482	30603	1187	✖	1117

13.2.4 Interprétations

Concernant le matériel utilisé : les temps de calcul obtenus confirment ce que nous avions déjà observé au début de ce rapport. Le processeur Intel Xeon est jusqu'à 3,5 fois plus rapide que le processeur AMD Phenom, cette différence est cohérente avec la différence de prix. La carte graphique GTX 560 est jusqu'à 2 fois plus rapide que la carte Quadro 4000 alors qu'elle est 3 fois moins chère. En excluant les cas de zoom, sur la première configuration l'implémentation GPU est en moyenne fois 5 plus rapide que l'implémentation CPU. Pour la deuxième configuration, le gain varie entre 20 et plus de 50 selon la configuration utilisée. Lors des premières implémentations, le facteur d'accélération était bien plus important entre GPU et CPU, mais les optimisations apportées amélioraient systématiquement davantage l'implémentation CPU par rapport à l'implémentation GPU. L'écart entre les deux versions s'est donc réduit au fil des optimisations.

Les tests de performance avec zoom démontrent que le kernel CUDA doit être choisi en fonction de la configuration, au contraire sur CPU ce n'est pas important étant donné que le nombre de threads reste très inférieur au nombre de positions. Sur GPU, étant donné que le nombre de positions est très inférieur au nombre de threads pouvant être exécutés simultanément sur tous les multiprocesseurs, la granularité du moteur de simulation rastérisation est beaucoup trop élevée : le parallélisme disponible n'est pas exploité. Le moteur utilisé doit alors exhiber une granularité beaucoup plus fine, par exemple le moteur **GPU Fine** (1 thread traite 1 triangle dans 1 position) doit être utilisé pour exploiter. Utiliser CUDA peut améliorer grandement les performances, mais si la granularité des tâches n'est pas correctement définie, le parallélisme ne sera pas exploité et l'implémentation pourra être plus lente que sur CPU.

Chapitre 14

Nombres flottants IEEE 754

Dans ce chapitre nous allons présenter comment est stocké un nombre à virgule flottante suivant la norme IEEE 754. Nous allons également étudier les implications de cette représentation comme la comparaison de nombres à virgule flottante, le concept d'Unit in the Last Place (ULP), les nombres dénormalisés et la stabilité numérique.

14.1 Représentation IEEE 754

14.1.1 Format général

Un nombre flottant est composé de trois éléments (voir figure **??**).
- Le signe s qui est toujours représenté sur un seul bit.
- L'exposant e.
- La mantisse m qui correspond à la partie fractionnaire du nombre.

FIGURE 14.1 – Représentation IEEE 754.

La valeur d'un nombre flottant normalisé est donnée par la formule :

$$(-1)^s \times 1.m \times 2^{e-b} \tag{14.1}$$

Le 1 implicite devant la mantisse ($1.m$) permet d'avoir une représentation unique pour chaque nombre. La valeur b est appelée le biais. Pour représenter des exposants négatifs, au lieu d'une représentation par complémentation à 2, il faut soustraire b à la valeur de l'exposant. Ce mécanisme rend la comparaison de nombres flottants plus facile comme nous allons le montrer.

14.1.2 Représentations spéciales

Certaines valeurs spéciales peuvent également être encodées, mais nous n'entrerons pas dans les détails, il s'agit de :
- $+0$ ou -0
- Les nombres dénormalisés pour représenter des valeurs très faibles.
- $+\infty$ et $-\infty$.

– **NaN** (Not a number), par exemple lorsqu'on calcule la racine carrée d'un nombre négatif.

14.1.3 Format simple précision (32-bit)

En C/C++ il s'agit du type `float`.

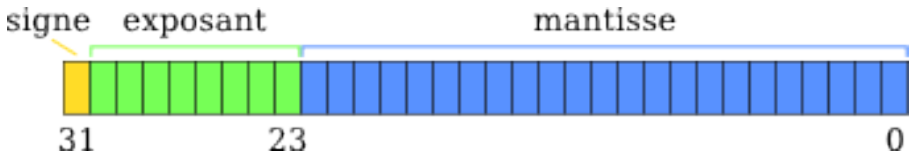


FIGURE 14.2 – Format simple précision.

14.1.4 Format double précision (64-bit)

En C/C++ il s'agit du type `double`.

FIGURE 14.3 – Format double précision.

14.2 Unit in the Last Place

14.2.1 Principe

La notion d'Unit in the Last Place (ULP) est un concept important en calcul scientifique. Cette notion est utilisée pour mesurer la précision des implémentations de fonctions mathématiques complexes, par exemple les fonctions transcendantes (exponentiel, logarithme, sinus, cosinus). Cette notion permet aussi de comparer deux résultats de simulation en fournissant une méthode de comparaison indépendante de la taille des valeurs numériques considérées.

La notion d'ULP est aussi liée à la représentation binaire des nombres flottants. Nous avons vu que deux nombres flottants proches ont des représentations binaires proches. La taille de la mantisse étant limitée, certains nombres ne peuvent pas être représentés en format simple ou double précision. Le nombre est alors arrondi afin d'être stocké dans une variable de type flottant. La conséquence est que l'ensemble des valeurs représentables prend des valeurs discrètes, certaines valeurs ne sont pas représentables. Ainsi, incrémenter de 1 la représentation binaire d'un nombre flottant permet d'obtenir le prochain nombre flottant représentable avec ce type de donnée. La différence entre les deux nombres obtenus est appelée ULP. De manière formelle, l'ULP est la valeur représentée par le bit de poids faible (least significant bit) du nombre flottant.

Soit un nombre x ayant pour exposant e en représentation normalisée. La valeur de l'ULP est alors :

$$\mathrm{ULP}(x) = \epsilon \times 2^e \tag{14.2}$$

ϵ est appelé l'epsilon machine (la borne supérieure de l'erreur relative après arrondi).

14.2.2 Exemples

Le tableau suivant montre sur différentes valeurs l'effet de l'arrondi pour stocker en format simple précision (opérateur rn) et la valeur de l'ULP correspondante.

v	$\mathrm{rn}(v)$	$\mathrm{ULP}(\mathrm{rn}(v))$	$\mathrm{rn}(v) + \mathrm{ULP}(\mathrm{rn}(v))$
0,3	0,300000011920929	$2,980232238769531 \times 10^{-8}$	0,3000000417232513
1,2	1,200000047683716	$1,192092895507812 \times 10^{-7}$	1,200000166893005
517,31	517,3099975585938	$6,103515625 \times 10^{-5}$	517,31005859375
1051,73	1051,72998046875	0,0001220703125	1051,730102539062
67311,11	67311,109375	0,0078125	67311,1171875

14.2.3 Précision des fonctions

La documentation de CUDA donne pour chaque opération sa précision en ULP. Les opérations arithmétiques classiques comme l'addition, la soustraction ou la multiplication ont une précision de $0,5$ ULP (à cause de l'arrondi) par rapport à la valeur mathématique réelle conformément au standard IEEE 754. Par contre, la précision du calcul des fonctions transcendantes n'est pas fixée par le standard. Le résultat peut donc différer de plusieurs ULP selon la valeur de l'argument.

14.3 Nombres dénormalisés

14.3.1 Principe

Nous avons vu la représentation des nombres flottants dits normalisés. Lorsque l'exposant e vaut 0 le nombre est au contraire dit dénormalisé. Les nombres dénormalisés permettent de représenter des valeurs très faibles et ainsi de combler l'espace entre 0 et le plus petit nombre normalisé. Sans les nombres dénormalisés, si le résultat d'une opération est plus faible que le plus petit nombre représentable, la valeur est alors 0 (**underflow** ou flush to zero). Les nombres dénormalisés ajoutent un nouvel intervalle de nombres pour éviter un underflow soudain à 0.

14.3.2 Performances

CPU

Les nombres dénormalisés sont souvent traités comme des exceptions par les instructions de calcul flottant (que ce soit sur FPU ou en mode SSE). Sur certaines architectures il n'existe pas de support hardware des nombres dénormalisés, lorsque l'unité de calcul flottant a pour opérande un nombre dénormalisé, une exception est levée, c'est alors le CPU qui calcule le résultat de l'opération. Dans ce cas-là, les temps de calcul peuvent être nettement ralentis. En mode SSE ou en mode FPU, il est possible de désactiver les nombres dénormalisés en activant 2 modes

- **Denormals Are Zero (DAZ)** : les opérandes dénormalisés sont traités comme la valeur 0.
- **Flush To Zero (FTZ)** : lorsque le résultat d'une opération ne peut pas être représenté par un nombre normalisé, le résultat est alors 0 (underflow).

L'utilisation de l'option de compilation `-ffast-math` désactive les nombres dénormalisés, si des nombres dénormalisés surviennent souvent dans le calcul, activer cette option peut améliorer les performances de façon significative.

GPU

Les nombres dénormalisés ne sont supportés que sur cartes graphiques de type 2.0 et supérieures, mais il n'y a pas de pénalité de temps d'exécution comme sur CPU. Les nombres dénormalisés ne posent pas de problème sur GPU.

14.4 Transformation des grandeurs physiques

Lorsque nous manipulons des variables de type simple ou double précision, la valeur stockée n'a pas forcément de signification physique. Les opérations sont effectuées sur des variables sans unité. On pourrait par exemple envisager de transformer les grandeurs physiques en entrée en les multipliant par une constante avant d'effectuer les calculs.

Avec le format IEEE754, multiplier les nombres par une puissance de 2 (positive ou négative) n'affecte pas la précision puisque la mantisse possède toujours autant de chiffres. La précision relative est inchangée. Ainsi, multiplier les valeurs numériques par une constante n'améliorera pas la précision des calculs. Cependant il peut toutefois être nécessaire d'effectuer ce changement d'échelle pour éviter que des résultats intermédiaires ne soient pas représentables en format normalisé. Afin d'obtenir une bonne stabilité numérique, le plus important est d'éviter de faire des additions entre nombres avec des exposants très différents.

14.5 Stabilité numérique

14.5.1 Catastrophic cancellation

En calcul flottant, lorsque des petits nombres sont calculés à partir de grands nombres on observe une importante perte de précision. Lorsqu'une soustraction est effectuée entre 2 nombres flottants très proches, le résultat obtenu a moins de précision que les deux opérandes. On parle de subtractive cancellation : la soustraction exhibe les erreurs d'arrondis survenu sur les grands nombres.

Exemple 1

Considérons le problème de recherche de racines d'une équation quadratique de type $ax^2 + bx + c = 0$ Les solutions sont : $x = \frac{-b \pm \sqrt{b^2-4ac}}{2a}$ Prenons le cas $a = 1$, $b = 200$, $c = -0.000015$:

$$x^2 + 200x - 0.000015 = 0 \tag{14.3}$$

Le discriminant a pour valeur :

$$\sqrt{b^2 - 4ac} = \sqrt{200^2 + 4 \times 1 \times 0.000015} = 200.00000015... \tag{14.4}$$

En arithmétique réelle, les racines sont :

$$\begin{array}{rcl} (\text{ -200 - } 200.00000015) \text{ / } 2 & = & \text{-200.000000075} \\ (\text{ -200 + } 200.00000015) \text{ / } 2 & = & \text{0.000000075} \end{array} \tag{14.5}$$

Avec une arithmétique ayant une mantisse décimale de taille 10 :

$$\begin{array}{rcr} (\text{-200 - 200.0000001}) \ / \ 2 & = & \text{-200.00000005} \\ (\text{-200 + 200.0000001}) \ / \ 2 & = & \text{0.00000005} \end{array} \tag{14.6}$$

La première racine possède 10 chiffres de corrects, mais la deuxième racine ne possède aucun chiffre de correct. Cet algorithme de calcul de racine n'est donc pas stable.

Exemple 2

Ce problème peut également survenir en double précision par exemple considérons la fonction suivante :

$$\mathrm{f}(x) = \frac{1 - \cos(x)}{x^2} \tag{14.7}$$

Pour $x = 1.1 \times 10^{-8}$ on obtient

$$\mathrm{f}(x) = 0.9175396897728 \tag{14.8}$$

Alors qu'en arithmétique réelle on obtient :

$$\mathrm{f}(x) = 0.49999999999999999495... \tag{14.9}$$

L'erreur relative est d'environ 83.5% !

Chapitre 15

Calcul flottant dans SIMSURF

15.1 Comparaison de nombres flottants positifs

15.1.1 Principe

Le décalage de l'exposant par un biais rend la comparaison de nombres flottants plus facile. Grâce à cette représentation, lorsque deux nombres flottants sont de signe positif ($s = 0$), leur comparaison équivaut à une comparaison lexicographique de leur représentation binaire. En d'autres termes, des nombres flottants proches ont des représentations binaires proches. De façon pratique, on peut réinterpréter la représentation binaire de chaque nombre comme des nombres entiers (de type `int`) puis comparer les deux nombres entiers pour obtenir le résultat de la comparaison.

15.1.2 Type Punning

En C/C++, cette méthode peut s'implémenter en utilisant la technique du **type punning**. Voici un exemple d'utilisation :

```
float a = 2.7183f;
float b = 3.1416f;

int ia = *(int*)&a;
int ib = *(int*)&b;
bool lt = ia < ib;
```

15.1.3 Application dans SIMSURF

Cette méthode est utilisée dans l'implémentation GPU de SIMSURF afin d'accélérer la mise à jour de la mémoire globale lorsqu'un brin est coupé. Nous utilisons une opération atomique pour mettre à jour la hauteur du brin tout en évitant les conflits d'accès. Nous avons besoin d'une opération calculant le minimum de deux valeurs de façon atomique.

La fonction `atomicMin` ne peut prendre que des opérandes de type nombre entier (`int`). Nous pourrions à la place utiliser une opération de type **Compare-And-Swap** (CAS). Mais comme nous l'avons vu précédemment, cette implémentation à base est plus lente qu'une implémentation à base de **atomicMin**.

Étant donné que la hauteur des brins est toujours positive, nous pouvons utiliser la technique du **type punning**. La fonction **atomicMin** de CUDA est alors utilisable puisque la comparaison entre nombres flottants positifs équivaut à une comparaison entre nombres entiers. Un benchmark des différentes opérations atomiques est présenté dans ce rapport.

15.2 Comparaison d'implémentations

L'ULP est un indicateur plus robuste que la différence pour comparer deux résultats de simulation. Une différence de 1 ULP indique que les deux nombres sont très proches, en effet il n'existe pas de nombre flottant représentable entre ces deux nombres. En calculant la différence entre les deux nombres, si l'exposant est élevé, on peut avoir une différence importante en valeur absolue. Les ULP sont ainsi utilisés pour comparer deux résultats de simulation effectués avec la même précision par exemple entre les implémentations CPU et GPU simple précision. Lors de l'exécution des tests de régression, une erreur moyenne de 2 ULP par brin est considérée comme une erreur et le test échoue. Sans utiliser les ULPs, il est difficile de donner une limite permettant de considérer que les deux implémentations divergent suffisamment.

15.3 Intersection brin-triangle

Lors du calcul d'intersection entre un brin et un triangle, la stabilité numérique est primordiale. Effectuer un test d'intersection entre un rayon et un triangle éloigné peut générer une erreur numérique importante.

Lorsqu'un thread travaille sur une position, chaque triangle du maillage outil doit être amené à sa bonne position : il faut appliquer selon le cas une rotation autour de l'axe outil, un changement d'axe et/ou une translation vers la position outil. L'algorithme d'intersection choisi utilise les coordonnées barycentriques pour vérifier si le point d'intersection avec le rayon est bien à l'intérieur du triangle. Il est pour cela nécessaire de calculer les coordonnées des deux arêtes (e_1, e_2) du triangle partant du premier sommet. Ces deux arêtes forment un repère dont les coordonnés sont appelées u et v. On peut soit calculer les coordonnées des arêtes avant ou après la translation. Comme nous l'avons vu, additionner un grand nombre (les coordonnées de la translation) et un petit nombre génère une erreur d'arrondi. Ainsi, pour une meilleure stabilité numérique il est préférable d'effectuer le calcul avant la translation.

15.3.1 Exemple

Considérons le triangle t_1 constitué des 3 sommets : $a = (0, 0, 2)$, $b = (0, 3.1, 5.1)$ et $c = (3.1, 0, 7)$. Considérons également le même triangle après une translation de vecteur (d, d, d) avec $d = 1024$ (valeur pouvant être représentée exactement en format simple précision). Nous appellerons ce triangle t_2. Observons l'impact sur les étapes de calcul lors du calcul de l'intersection de t_1 avec le brin $r_1 = (1.4, 1.4, 0)$ et $r_2 = r_1 + (d, d, d)$

	$e_1.y$	det	u	v	dist
t_1/r_1	3.0999**9**990	9.609**9**9965	3.4**1**000008	4.33999**67**	5.3580**6**465
t_2/r_2	3.0999**7**558	9.609**8**4897	3.4**0**989756	4.3400**4**163	d + 5.3580**3**223

Nous pouvons observer que les coordonnées u et v et la distance d'intersection sont différents entre les deux cas. Cette observation signifie que dans certains cas un point d'intersection peut être considéré à l'intérieur ou à l'extérieur du triangle selon si la translation a été appliquée auparavant. La hauteur du brin change également selon la stratégie utilisée.

15.3.2 Stratégie

Afin d'obtenir une meilleure stabilité numérique, nous devons éviter les problèmes de type catastrophic cancellation. L'application de la translation sur les sommets du triangle fait perdre de la précision sur les coordonnées. De plus les coordonnées x et y du brin sont très proches des coordonnées x et y des triangles par contre les coordonnées z sont très différentes. Ajouter et soustraire des valeurs avec des exposants très différents implique également une perte de précision.

Au lieu de calculer l'intersection dans le repère monde en effectuant la translation sur le triangle, on peut effectuer l'intersection dans le repère de l'objet en appliquant la translation inverse au brin (sauf sur la coordonnée z). De cette manière on ne perd pas de précision sur les coordonnées des triangles et les toutes les coordonnées des sommets et du brin sont du même ordre de grandeur. La première stratégie est présentée sur la figure 15.1, la version améliorée est ensuite présentée sur la figure 15.2.

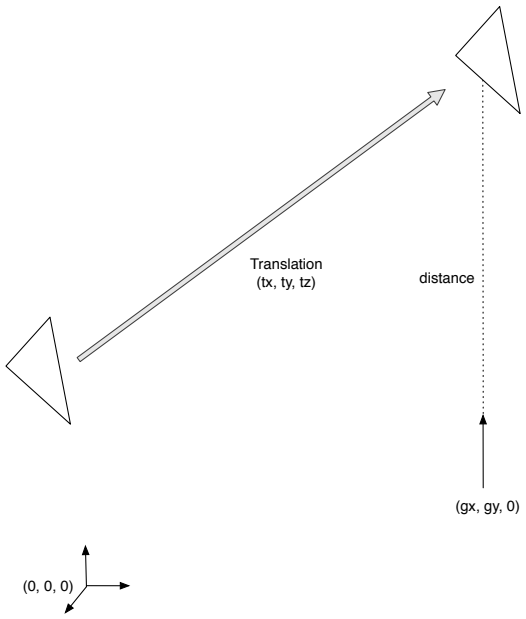


FIGURE 15.1 – Stratégie initiale.

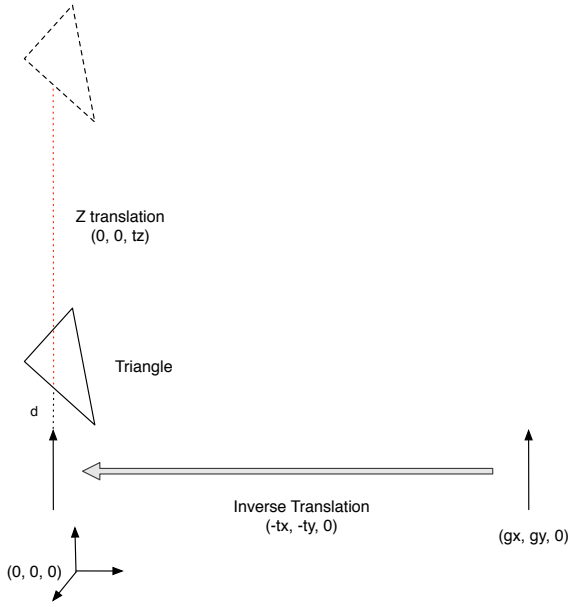


FIGURE 15.2 – Stratégie améliorée.

15.4 Z-buffer simple/double précision

Les implémentations CPU et GPU de la simulation peuvent s'effectuer en mode simple ou double précision (32 bits ou 64 bits). En utilisant les possibilités de généricité offertes par les template C++ il n'y a pas de duplication de code : la même fonction peut être utilisée en paramétrant le type de donnée par float ou double.

Nous allons étudier l'impact du choix du type de donnée sur le temps de calcul et sur la précision des résultats de la simulation. Nous allons mener cette étude sur le CPU et le GPU. Les benchmarks ont été réalisés sur la configuration personnelle.

15.4.1 Influence sur le temps de calcul

Le nombre de registres utilisés et l'occupation mémoire ont été réduits en utilisant une stratégie de calcul mixte : seuls les brins sont stockés en double précision, les triangles et les positions sont stockés en simple précision (une meilleure précision n'est pas nécessaire à ce niveau là). L'opération d'intersection est effectuée en 64-bit car une meilleure précision est seulement nécessaire à cet endroit là du code. Stocker toutes les données en double précision augmente le nombre de transactions mémoire nécessaires et augmente nettement le temps de calcul sans améliorer la précision finale.

Dans les sections suivantes nous allons comparer les implémentations 32-bit et 64-bit sur CPU et sur GPU.

CPU

Les différences de temps d'exécution sur CPU sont illustrées dans le tableau suivant. Les configurations utilisées sont expliquées dans le chapitre **Evaluation des performances**.

Le projet étant compilé et exécuté sur une architecture 64-bit, les opérations sur les nombres flottants sont effectuées en utilisant les instructions SSE scalaires (sans vectorisation). Les registres SSE sont de taille 128 bits, il n'y a donc pas de différence fondamentale

Path		Tool		Temps (ms)	
Nom	Taille	Nom	Taille	CPU 32-bit	CPU 64-bit
random	200000	sphere	320	1270	1359
spiral	200000	sphere	1984	4383	5015
downspiral	200000	cone	64	779	799
volcano	200000	sphere	1984	4676	5221
ebauche	47837	tore	25904	25508	28388
finition	53667	sphere	12482	31711	32308

sur de temps d'exécution entre une opération 32-bit et une opération 64-bit. La différence observée s'explique par la différence de taille du type de donnée : plus de transactions mémoires sont nécessaires et plus de **cache miss** sont générés.

Sur une architecture 32-bit, les calculs flottants sont effectués sur la FPU en mode précision étendue (80 bits). L'implémentation 32-bit serait donc plus précise qu'en SSE étant donné que les opérations intermédiaires sont effectuées sur 80 bits. Il n'y aurait donc pas non plus de différence fondamentale sur le temps d'exécution car 32-bit et 64-bit sont gérés en précision étendue.

GPU

Obtenir des performances élevées sur CUDA nécessite de limiter les accès à la mémoire globale, or en double précision ces accès seront doublés. De plus le nombre de registres utilisé dans un kernel à un impact très important sur le nombre de threads pouvant être lancés par bloc et par conséquent sur les performances. De plus, seulement certains cœurs CUDA peuvent exécuter des opérations flottantes 64 bits. L'implémentation CUDA 64-bit utilise 59 registres contre 33 pour l'implémentation 32-bit, le parallélisme potentiel est donc plus faible car la taille du banc de registre sur chaque multiprocesseur sera limitant.

Les résultats sont présentés dans le tableau suivant.

Path		Tool		Temps (ms)	
Nom	Taille	Nom	Taille	GPU 32-bit	GPU 64-bit
random	200000	sphere	320	48	106
spiral	200000	sphere	1984	105	549
downspiral	200000	cone	64	25	108
volcano	200000	sphere	1984	85	410
ebauche	47837	tore	25904	714	3272
finition	53667	sphere	12482	1392	4185

Sur CPU, la pénalité de performance induite par l'utilisation de double précision est faible. Il est donc acceptable d'utiliser cette précision. Cependant, de nombreux mécanismes interviennent et selon l'architecture et le compilateur il est possible que la différence de précision entre les deux implémentations soit faible (par exemple si la précision étendue est utilisée). Il est aussi possible que l'implémentation flottante soit 2 fois plus rapide si le compilateur arrive à vectoriser le calcul.

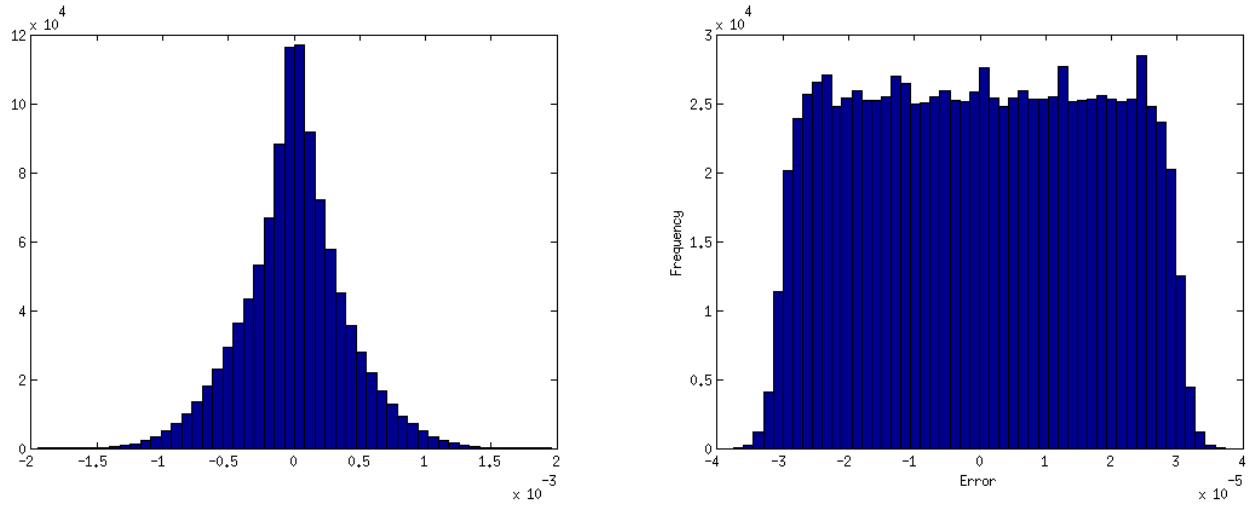
Sous CUDA utiliser la double précision est réellement handicapant au niveau des performances puisque tous les cœurs CUDA ne peuvent être utilisés.

15.4.2 Influence sur la précision

Nous faisons ici l'hypothèse que sur une même architecture, le résultat en double précision est nécessairement plus précis (au sens plus proche du résultat mathématique réel) que le résultat en simple précision. La différence entre les deux résultats représente donc l'erreur obtenue en utilisant de la simple précision. Les résultats sont convertis en représentation double précision avant comparaison.

Résultat

Les résultats CPU et GPU étant similaires, nous ne présentons qu'un seul graphique à la figure 15.3. La configuration utilisée est **downspiral** (outil conique). Nous présentons également à côté la version avec stabilité améliorée présentée dans la section précédente. L'erreur est absolue, sur une base d'un brin de hauteur 1000.



(a) Version initiale

(b) Version améliorée

FIGURE 15.3 – Erreur absolue par rapport à l'implémentation 64-bit.

15.5 Option `-use_fast_math`

CUDA propose un flag `-use_fast_math` permettant de remplacer certaines opérations sur nombres flottants par un équivalent plus rapide (utilisant moins d'instructions) mais au prix d'une précision plus faible. Nous allons étudier l'effet de l'activation de cette option sur le temps de calcul et sur la précision. `-use_fast_math` active les options `-ftz=true` (flush to zero, pas de nombres dénormalisés), `-prec-div=false` (division précise), `-prec-sqrt=false` (racine carrée précise).

15.5.1 Impact sur le temps de calcul

Sans ce flag on observe un ralentissement significatif : jusqu'à **40%**. Plus de registres sont utilisés (37 contre 33) donc moins de parallélisme théorique. Cette augmentation du nombre de registres s'explique probablement par la création de davantage de variables temporaires afin de rester en conformité avec le standard IEEE 754. Après enquête, la

différence de temps de calcul provient principalement de l'opération de division plus rapide et dans une moindre mesure la désactivation des nombres dénormalisés.

15.5.2 Impact sur la précision

Nous avons comparé le résultat obtenu sur notre configuration de test avec et sans l'utilisation du flag `-use_fast_math`. Les erreurs sont rassemblés dans l'histogramme 15.4.

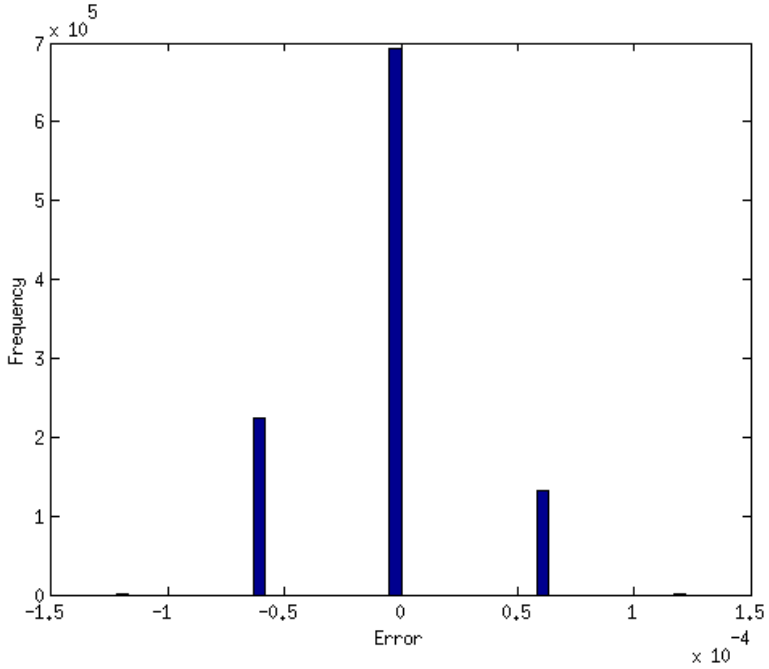


FIGURE 15.4 – Erreur absolue obtenue avec `-use_fast_math`.

La distribution à la fois discontinue et symétrique de l'histogramme peut surprendre, mais il faut se rappeler qu'ici nous comparons 2 résultats effectués en simple précision. Les valeurs observées : environ 6×10^{-5} et 1.2×10^{-4} correspondent exactement à 1 et 2 ULP pour cette hauteur (1 ULP : $2^{-14} = 6.1035156 \times 10^{-5}$). La documentation CUDA indique une erreur maximale de 2 ULP pour la fonction `__fdividef`. Notre observation est donc cohérente avec le résultat attendu.

15.5.3 Conclusion

Dans cet exemple l'utilisation du flag `-use_fast_math` introduit une erreur de seulement 1 ou 2 ULPs par rapport au résultat original. Au vu de l'amélioration significative du temps de calcul, l'utilisation de ce flag semble justifiée.

Chapitre 16

Visualisation

Nous allons maintenant expliquer pourquoi l'API OpenGL a été choisie pour la visualisation et comment l'interface utilisateur a été implémentée.

16.1 Choix d'une méthode de visualisation

Après un premier prototype du projet implémenté avec OpenGL, la question s'est posée de changer de stratégie pour la visualisation 3D de la surface résultat. Après analyse des différents logiciels et bibliothèques proposée sur le marché, plusieurs possibilités ont été rapidement isolées :

- Utilisation de l'API bas niveau OpenGL.
- Utilisation d'une bibliothèque haut-niveau dédiée à la visualisation scientifique. Par exemple **Open Scene Graph (OSG)** ou **Visualization Toolkit (VTK)**. On bénéficie alors de nombreux algorithmes de visualisation déjà implémentés.
- Intégration comme plugin dans un logiciel de visualisation 3D comme Meshlab. On peut utiliser toute l'interface du logiciel hôte ainsi que tous les algorithmes disponibles.

Après avoir évalué les avantages et les inconvénients de ces possibilités, nous avons conclu qu'utiliser l'API OpenGL était la meilleure option pour les besoins du projet SIMSURF. Le temps nécessaire pour apprendre VTK ou OSG a finalement semblé trop important, et une brève utilisation de ces bibliothèques a émis des doutes sur leur stabilité (certains exemples fournis ne marchaient pas). Concernant Meshlab, afficher une surface de taille 1024×1024 faisait chuter le nombre de FPS à 4-5, rendant cette possibilité inutilisable.

16.2 Utilisation de l'API OpenGL

16.2.1 Présentation

OpenGL est une spécification d'API pour le développement d'application 2D ou 3D. C'est une bibliothèque de bas niveau très proche de la carte graphique et permettant de personnaliser la plupart des étapes de traitement du **pipeline** graphique. Cette personnalisation des traitements peut soit s'effectuer par le changement d'un état interne (OpenGL est représenté comme étant une machine à états) ou par l'utilisation de shaders : des programmes s'exécutant sur la carte graphique et modifiant les sommets ou les pixels à afficher.

Une illustration simplifiée du pipeline de traitement graphique d'OpenGL est présentée sur le diagramme 16.1. Nous allons détailler les avantages et les inconvénients induits par

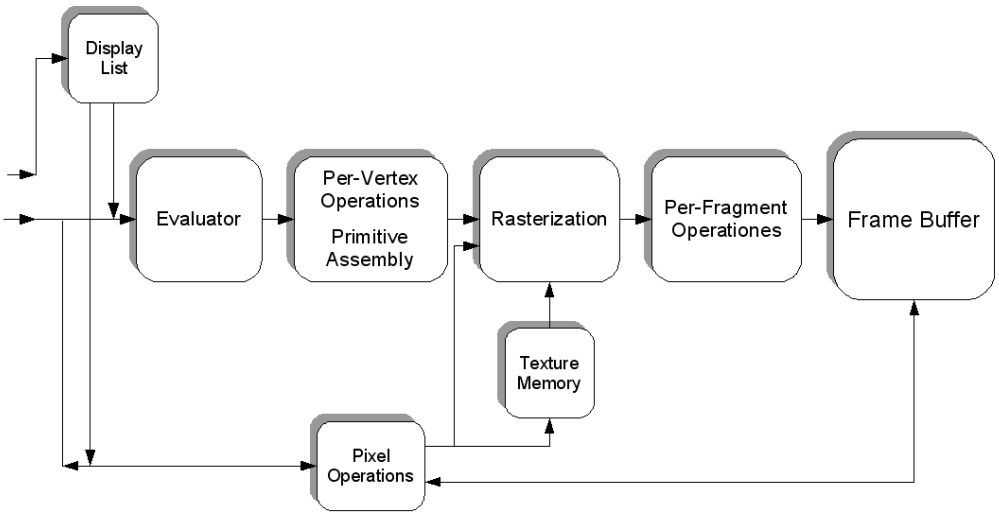


FIGURE 16.1 – Chaîne de traitement OpenGL.

le choix de cette méthode de visualisation.

16.2.2 Avantages

- Personnalisation avancée à tous les niveaux de la chaîne de traitement graphique en modifiant les états OpenGL ou en appliquant des shaders.
- Performances élevées car l'affichage d'un objet peut être optimisé en fonction de son type. Par exemple l'affichage d'une grille régulière peut être optimisé en utilisant une primitive OpenGL adaptée comme `GL_TRIANGLE_FAN` ou `GL_TRIANGLE_STRIP`. L'utilisation de ces primitives permet d'éviter des calculs redondants et diminue également l'occupation mémoire de l'objet.
- Cette solution est suffisante pour les besoins du projet SIMSURF puisque l'objectif est seulement de développer un démonstrateur. Utiliser une bibliothèque de haut-niveau serait plus avantageux sur un programme devant évoluer sur le long terme.

16.2.3 Désavantages

Nous avons vu qu'utiliser une bibliothèque de bas niveau permettait une très grande flexibilité et permet d'atteindre les meilleures performances possible en choisissant la méthode d'affichage adaptée à nos besoins. Cependant OpenGL peut s'apparenter à une sorte de langage assembleur de la programmation graphique et la flexibilité et la performance ont un coût. Nous avons relevé plusieurs désavantages pour cette approche :

- Portabilité plus complexe. Selon la carte graphique, toutes les fonctions de l'API OpenGL ne sont pas forcément disponibles. Nous avons par exemple rencontré un problème de portabilité sous Mac OS X avec une des premières versions du projet. La fonctionnalité de **primitive restart** n'était pas disponible sur la version de Mac OS X et donc la première implémentation a été réécrite. Obtenir une bonne portabilité en utilisant une bibliothèque de bas niveau nécessite une bonne expérience de la bibliothèque en question. Le problème de portabilité cité ici n'avait pas été anticipé car non mentionné par le livre dont l'implémentation est issue (Farber (2011)).
- De nombreux algorithmes doivent être réimplementés. Par exemple les algorithmes permettant le déplacement de la caméra et son orientation dans l'espace, ou encore

les algorithmes de **picking** permettant d'obtenir les coordonnées 3D de l'objet sous le curseur de la souris.
– L'intégration d'une interface est difficile. Il est possible avec OpenGL de tracer des primitives géométriques en 2D sur le plan de l'écran mais aucune fonctionnalité de plus haut niveau n'est disponible pour afficher des boutons et des menus ou gérer l'interaction avec l'utilisateur.

16.3 Fonctionnalités implémentées

OpenGL étant très bas niveau, toutes les fonctionnalités d'affichage doivent être implémentées. Nous avons aussi apporté une attention particulière à l'optimisation du code d'affichage afin de permettre de visualiser des surfaces de grande dimension (4096×4096 soit 33,5 millions de triangles)

16.3.1 Surface

La surface obtenue après la simulation est affichée en utilisant un maillage régulier de triangles, voir figure 16.2. Chaque carré de la grille est maillé par 2 triangles isométriques.

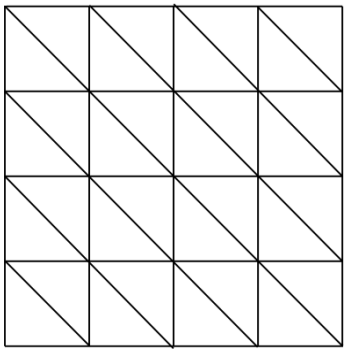


FIGURE 16.2 – Maillage d'une grille régulière.

Les coordonnées des sommets sont partagées afin de diminuer la consommation mémoire : un tableau contient tous les sommets de la grille et un autre tableau contient les indices consécutifs pour former tous les triangles de la grille. L'utilisation de la fonctionnalité `GL_TRIANGLE_STRIP` de OpenGL permet de déclarer moins d'indices que dans le cas ou tous les triangles sont distincts : nous avons seulement besoin de 2 sommets supplémentaires pour un nouveau triangle, le troisième sommet est implicitement le dernier sommet du dernier triangle déclaré. Pour tracer 2 triangles uniquement 4 indices sont nécessaires comme démontré sur la figure 16.3.

Pour gérer l'illumination de la surface par la source lumineuse, nous devons assigner à chaque point une normale. En chaque point, la normale est calculée en prenant la moyenne des normales de chaque triangle autour de ce point.

Pour diminuer la bande passante entre CPU et GPU, les sommets, les indices, les normales et les couleurs sont stockés sur la carte graphique en utilisant des **Vertex Buffer Object** (**VBO**).

Initialement l'affichage de la grille était préparé depuis un kernel CUDA en utilisant l'interaction OpenGL-CUDA et la fonctionnalité **primitive restart**. Cette fonctionnalité

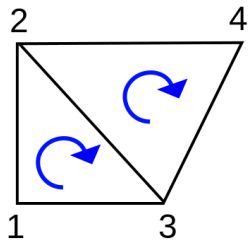


FIGURE 16.3 – Triangle Strip.

a été abandonnée car elle s'est avérée être moins performante que celle citée précédemment.

16.3.2 Outil et trajectoire

La trajectoire de l'outil est *rejouée* par-dessus le résultat de la simulation (l'enlèvement progressif de la matière n'est pas visible). Un outil représenté par une liste de triangles est affiché en dessinant chaque triangle successivement. Les autres outils sont affichés en utilisant les fonctions spéciales fournies par l'extension GLUT (par exemple `glutSolideSphere`). Ces appels à OpenGL sont compilés dans des **Display Lists** afin d'optimiser les performances.

16.3.3 Caméra

La caméra s'ajuste automatiquement aux dimensions de la surface visualisée en calculant sa boîte englobante. L'utilisateur peut alors déplacer la surface (**drag**) avec le clic gauche de la souris et se rapprocher/s'éloigner avec la molette de la souris. Le clic droit permet une rotation en mode **trackball** (l'objet est englobé dans une sphère 3D). Tous les calculs de rotation/translation de la caméra pour une surface quelconque ont été implémentés intégralement dans le projet. Le principe de la caméra trackball est présenté dans le schéma 16.4 venant de **?**. L'algorithme consiste à trouver la rotation permettant de passer du vecteur initial au vecteur actuel sous notre souris. On doit pour cela trouver l'axe et l'angle de cette rotation.

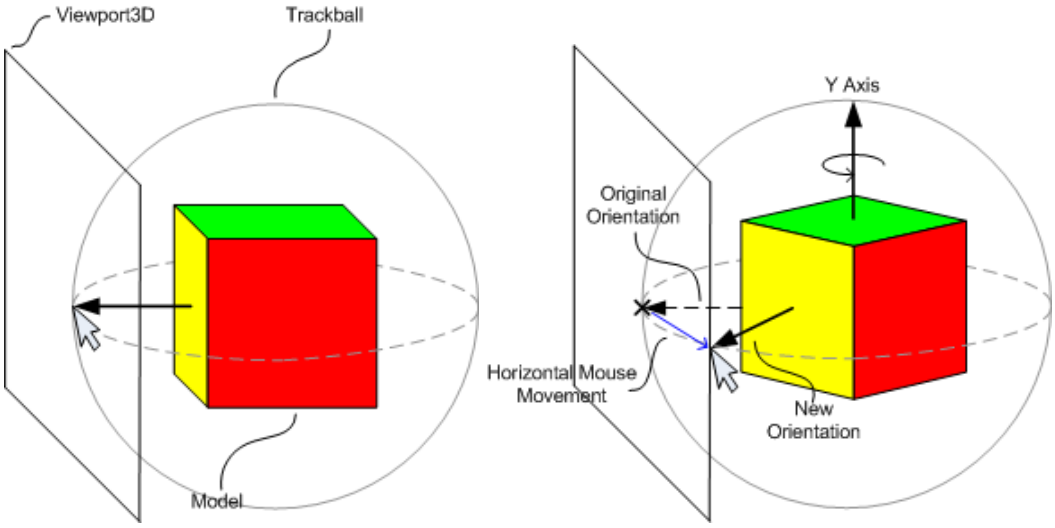
16.3.4 Zoom sur la surface

L'utilisateur peut sélectionner le mode zoom et utiliser le clic gauche ou le clic droit sur un endroit de la surface pour relancer une simulation ayant pour nouveau centre le point sélectionné. Un clic gauche divise l'espacement entre les brins par 2 (zoom-in) et un clic droit multiplie cet espacement par 2 (zoom-out). Le calcul du point d'intersection sur la surface s'effectue sans difficultés avec les fonctions `glReadPixels` et `gluUnProject`.

16.4 Interface Utilisateur

16.4.1 AntTweakBar

OpenGL ne fournit pas de fonctionnalités pour gérer l'affichage et les interactions avec une interface utilisateur. Utiliser des bibliothèques de haut-niveau aurait permis d'utiliser des fonctions déjà intégrées mais cette possibilité a été rejetée pour les raisons



(a) Position initiale (b) Déplacement horizontal

FIGURE 16.4 – Principe de la caméra trackball.

que nous avons citées. Au lieu de réimplémenter un système d'interface avec OpenGL, nous avons choisi de chercher une bibliothèque légère gérant l'affichage 2D d'une interface et l'interaction avec cette interface. Notre choix s'est rapidement porté sur la bibliothèque C/C++ AntTweakBar du fait de sa portabilité, de sa simplicité et des fonctionnalités proposées. AntTweakBar permet de manipuler facilement des variables de l'application comme des entiers, des flottants, des couleurs, des vecteurs ou même des quaternions (pour une rotation).

Voici un extrait de l'interface utilisateur du projet sur la figure 16.5, montrant les différentes fonctionnalités de manipulation de la bibliothèque AntTweakBar.

16.4.2 Intégration

Au moment de la transformation de l'architecture vers un modèle de programmation par flots de données, la question de l'intégration des actions utilisateurs s'est posée. Nous avons montré qu'il est facile d'ajouter de nouveaux filtres de traitement en chaînant des blocs. Cependant les filtres sont souvent paramétrés par une valeur (un seuil, une taille de voisinage...), afin de pouvoir modifier cette valeur depuis l'interface, deux possibilités se sont dégagées :

- Ajout d'appels à AntTweakBar dans les acteurs. Le désavantage est que le bloc ne peut plus être utilisé dans une configuration sans visualisation. Les variables modifiables depuis l'interface modifient directement les attributs de l'acteur.
- Utilisation de deux blocs : le bloc de filtre qui reçoit en entrée les données et les paramètres de filtrage ; et un bloc d'interface d'interface qui ajoute une variable à l'interface et envoie la nouvelle valeur vers le filtre quand elle est modifiée.

Finalement les deux possibilités sont utilisées, certains blocs (ceux liés à OpenGL) n'ont de sens que dans une configuration de visualisation, la première possibilité est alors utilisée pour ne pas surcharger l'interface de l'acteur de multiples paramètres. Concernant les blocs de traitement, c'est la seconde approche qui est privilégiée afin de pouvoir les réutiliser en mode batch ou comparaison.

FIGURE 16.5 – Extrait de l'interface utilisateur.

Si un acteur souhaite ajouter des variables ou des callbacks auprès de l'interface, il doit implémenter la méthode `register_interface`. Cette méthode est appelée récursivement sur les acteurs et les sous-réseaux en suivant un parcours profondeur : l'organisation de l'interface est ainsi automatique.

```
/*
  When an UI is used, a class can implement this optional method
  to register UI components. The 'bar' argument is the current TweakBar
  used by the network containing the current actor.
  An actor can ignore the argument and instantiate a new TweakBar instead.

  This method allow the class' internals to be modified by an UI
  action. For example a processing actor can be parametrized either by
  receiving the value of the parameter through an input connection or by
  a UI action (for example a value slider or a selection list).

  Best practice however would be to have all processing actors receive
  parameters from input connections and use generic actors to declare selection
  lists, sliders, buttons, etc... These generic UI actors should send values
  to the processing actor each time the parameter has changed.
*/
#ifdef USE_UI
virtual void register_interface(TwBar* bar);
#endif
```

16.4.3 Fonctionnalités interface

Nous allons présenter la liste des fonctionnalités implémentées actuellement dans le projet.

Configuration

- Choix du fichier de configuration : une surface initiale et une liste de couples trajectoire/outil à appliquer à la suite.
- Activation du mode zoom, le zoom-in est effectué avec le clic gauche, le zoom-out avec le clic droit.
- Stratégie de zoom : avec interpolation des hauteurs de la surface ou limitée (zoom limité par la taille de la surface initiale).
- Computing : variable booléenne permettant de savoir si un calcul est en cours.
- Statistiques sur le temps d'exécution, divisé en trois parties : initialisation, exécution et finalisation de l'algorithme (en millisecondes).
- Choix du moteur de simulation à utiliser, la liste est modifiée dynamiquement en fonction de la configuration.

Affichage

- Orientation de la lumière : modification des coordonnées du vecteur directeur des rayons.
- Primitive d'affichage : triangles remplis, fils de fer, lignes ou points.
- Colorisation de la surface : uniforme ou colormap.
- Choix de la colormap, les mêmes que GNU Octave : Jet, Cool, HSV...

Vue

- Rotation de la surface : modifiable par un quaternion.
- Activation du mode vidéo (la trajectoire est rejouée par-dessus la surface).
- Temps total (en secondes) de la vidéo.

Filtres

- Activation du bloc de filtrage Laplacien de la surface résultat.
- Activation du bloc de simplification de maillage et choix du nombre de triangles en sortie.

Statistiques

- Statistiques sur l'outil : boite englobante, taille mémoire, nombre de triangles, dimensions moyennes des triangles.
- Statistiques sur la trajectoire : type de trajectoire, taille mémoire, nombre de positions.
- Statistiques sur la surface : dimensions, distance entre les brins, coordonnées de l'origine, taille mémoire.
- Statistiques sur l'accélérateur utilisé : mémoire globale disponible, activation du watchdog, ECC, nombre de cœurs CUDA, etc.

16.5 Capture d'écran

Une vue globale de l'application comprenant l'interface, la surface et l'outil par dessus est présentée sur la figure 16.6.

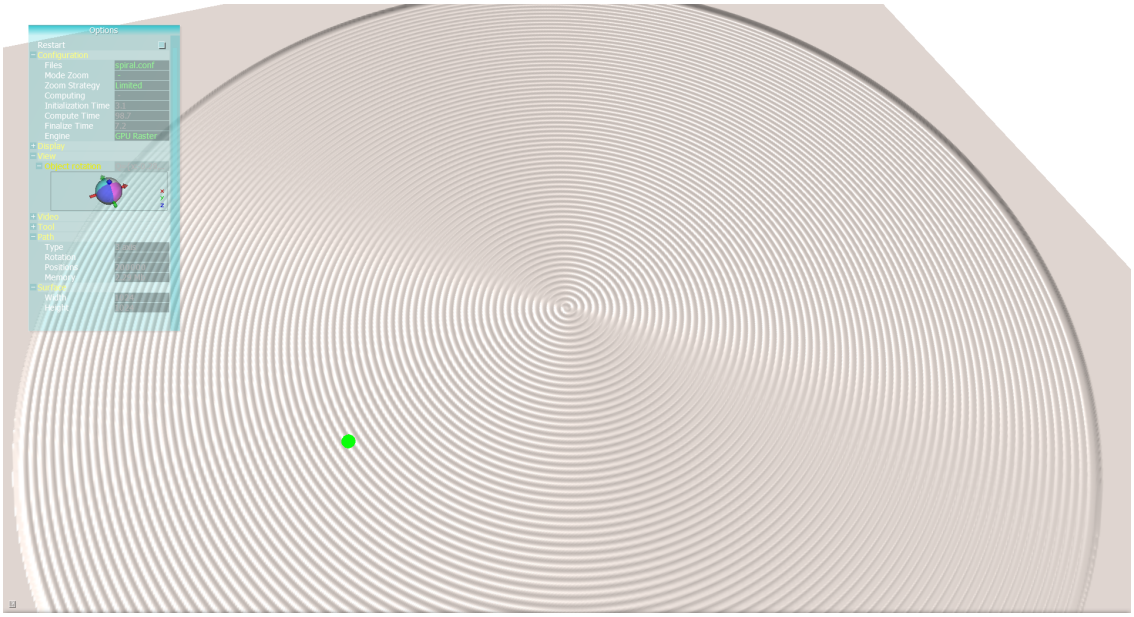


FIGURE 16.6 – Capture d'écran globale.

Chapitre 17

Conclusion

Réussir à exploiter les capacités de l'architecture massivement parallèle CUDA exige une très connaissance de son fonctionnement. Une bonne maîtrise granularité des tâches de l'algorithme, c'est-à-dire la quantité de travail qu'un thread CUDA doit effectuer, est primordiale afin de pouvoir utiliser simultanément tous les multiprocesseurs tout en permettant un équilibre efficace de la charge de travail. Si l'architecture CUDA est mal exploitée, l'implémentation peut au final être plus lente que l'implémentation CPU !

Une version optimisée de l'algorithme a été développée à la fois sur CPU et sur GPU. La version CPU multicœurs (sur 6 cœurs avec hyperthreading) est 8 fois plus rapide que la même version non parallèle. Cette implémentation finale est elle même 3 à 4 fois plus rapide que la première version CPU implémentée. L'implémentation CUDA est au minimum 5 fois plus rapide que l'implémentation multicœurs CPU sur la configuration ENS. Avec une carte graphique d'un prix équivalent au CPU professionnel utilisé, ce facteur pourrait être de 15 voir 20.

Une attention particulière a été apportée à l'architecture du projet et à la conception des algorithmes afin de limiter la duplication du code. L'architecture choisie, basée sur un modèle de programmation par flot de données, pourra, je l'espère, permettre aux utilisateurs ayant des connaissances de base en C++ d'ajouter des algorithmes dans la chaîne de traitement.

Le projet pourrait encore être augmenté de nombreuses fonctionnalités pour l'améliorer, certaines de ces fonctionnalités dépendent de l'évolution de l'architecture CUDA :

- Utilisation de l'API **Message Passing Interface** (MPI) pour paralléliser une exécution sur un cluster CPU ou GPU. Le centre de calcul du LMT pourrait être utilisé.
- Implémentation raytracing de la simulation. Une structure d'accélération doit alors être utilisée pour améliorer les performances. L'article Aila *et al.* (2012) montre des performances très intéressantes en utilisant CUDA.
- Utilisation des composants dédiés à la rastérisation sur la carte graphique. Ces composants ne sont actuellement pas programmables depuis CUDA mais cela pourrait améliorer les performances de notre algorithme d'un ou plusieurs ordres de grandeur.
- Refactorisation de certaines implémentations GPU : le compilateur NVCC n'est pas encore assez mature et certaines constructions utilisant des templates étaient rejetées.

Bibliographie

Aila, T., Laine, S. et Karras, T. (2012). Understanding the efficiency of ray traversal on GPUs – Kepler and Fermi addendum. NVIDIA Technical Report NVR-2012-02, NVIDIA Corporation.

Alexandrescu, A. (2001). *Modern C++ design : generic programming and design patterns applied*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Farber, R. (2011). *CUDA Application Design and Development*. Elsevier Science.

Haque, I. S. et Pande, V. S. (2009). Hard Data on Soft Errors : A Large-Scale Assessment of Real-World Error Rates in GPGPU. *CoRR*.

Jerard, R. B., Drysdale, R. L., Hauck, K., Schaudt, B. et Magewick, J. (1989). Methods for detecting errors in numerically controlled machining of sculptured surfaces. *IEEE Comput. Graph. Appl.*, 9(1):26–39.

Laine, S. et Karras, T. (2011). High-performance software rasterization on GPUs. *In Proceedings of High-Performance Graphics 2011*.

Möller, T. et Trumbore, B. (1997). Fast, minimum storage ray-triangle intersection. *J. Graph. Tools*, 2(1):21–28.

NVIDIA (2009). Fermi Compute Architecture Whitepaper. http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf.

NVIDIA (2012). CUDA C Programming Guide. http://developer.nvidia.com/cuda/cuda-downloads.

Zhang, W. et Majdandzic, I. (2010). Fast triangle rasterization using irregular Z-buffer on CUDA.