

ACADÉMIE DE MONTPELLIER
UNIVERSITÉ MONTPELLIER II
— SCIENCES ET TECHNIQUES DU LANGUEDOC —

MÉMOIRE DE STAGE DE MASTER

SPÉCIALITÉ : **Recherche en Informatique**
Mention : **Informatique, Mathématiques, Statistiques**

effectué au LP2A - DALI

—
sous la direction de DEFOUR DAVID

**Arithmétique flottante sur processeurs
graphiques**

par

DA GRAÇA Guillaume

Soutenu le 14 juin 2006

Table des matières

1	Introduction	1
1.1	Arithmétique flottante sur GPU	1
1.2	Motivations	1
1.3	Contributions	2
2	Contexte	3
2.1	La norme IEEE-754	3
2.1.1	Présentation de la norme IEEE-754	3
2.1.2	Représentation et précision	3
2.1.3	Nombres IEEE-754 et valeurs spéciales	4
2.1.4	Les exceptions	6
2.1.5	Les arrondis	6
2.1.6	Mesure de l'erreur	7
2.2	Les processeurs graphiques	7
2.2.1	Architecture et fonctionnement	7
2.2.2	Format flottant sur processeurs graphiques	13
2.3	La programmation sur processeurs graphiques	13
2.3.1	Brook	13
2.3.2	Les interfaces de programmations : OpenGL et DirectX	14
2.3.3	Cg	15
3	Test des opérateurs arithmétiques	17
3.1	État de l'art	17
3.2	Les algorithmes	18
3.2.1	Test de représentation des nombres flottants dans la mémoire et les registres temporels	18
3.2.2	Test de l'addition	20
3.2.3	Test de la multiplication	20
3.3	Les résultats	22
4	Opérateurs flottants 44 bits	25
4.1	État de l'art	25
4.2	Les algorithmes	26
4.3	Les résultats	30
4.3.1	Le temps d'exécution	30
4.3.2	La précision	32
5	Conclusion	33

A	Messages générés par Paranoïa pour la Nvidia 7800GTX	a
B	Temps d'exécution des opérateurs sur 44 bits	g
	Bibliographie	i
	Index	m

Remerciement

Je voudrais remercier David Defour mon directeur de stage de MASTER 2^{me} année qui a pris de son temps pour m'encadrer, me diriger et me conseiller pendant ces six mois de stage. Je remercie également Marc Daumas qui a partagé ses connaissances avec moi.

Je tiens également remercier tous les membres de l'équipe DALI qui m'ont accueillis, m'ont permis de m'impliquer dans différentes manifestations, tels que les Journées Nationales de l'Arithmétique des Ordinateurs (JNAO 2006) et avec lesquels j'ai passé d'agréables moments.

Je remercie enfin ma famille et tous mes amis qui m'ont soutenu et encouragé tout au long de ces 6 mois de stage.

Chapitre 1

Introduction

1.1 Arithmétique flottante sur GPU

L'objectif de notre recherche est d'exploiter les processeurs graphiques (en anglais : *Graphic Processing Unit* - GPU), et plus particulièrement de la puissance de calcul qu'ils possèdent. Il est nécessaire de mieux comprendre le fonctionnement de l'arithmétique flottante sur GPU pour maîtriser le comportement numérique des applications scientifiques exécutées sur GPU. Or les constructeurs ne nous donnent que très peu d'informations sur les opérateurs dont les GPU disposent, et il n'existe à ce jour pas de logiciel pouvant les tester. Il nous a donc fallu dans un premier temps créer une série de tests pour découvrir ce fonctionnement. De plus, les GPU actuels possèdent des types de données qui sont de faible précision. Donc, les applications dont la précision est primordiale ne conviendront pas à l'exécution sur GPU à cause de l'absence d'un format double précision, d'une non-uniformité du format à virgule flottante [9]. Il nous a donc fallu dans un deuxième temps émuler logiciellement des opérateurs haute précision, tout en tenant compte des tests effectués auparavant. Grâce à ces deux projets, nous voulons transférer des applications numériques appartenant à des domaines divers tels que la bio-informatique.

1.2 Motivations

Le GPU a un fonctionnement différent de celui d'un CPU. Un GPU est plus adapté pour le parallélisme de données alors que le CPU est plus adapté pour le parallélisme de tâches. Si l'on souhaite exécuter un processus sur un grand jeu de données, il est plus bénéfique d'utiliser un GPU, alors que si l'on souhaite exécuter plusieurs processus en même temps, le choix de l'exécution sur CPU est plus raisonnable. De ce fait, nous observons des différences entre les performances des GPU et des CPU :

1. les performances d'un GPU sont en croissance constante avec le temps, alors que celles d'un CPU ont tendance à stagner. Actuellement, on observe que les GPU ont une puissance de calcul sur les opérations flottantes environ 10 fois supérieure à celle des CPU.
2. la vitesse d'accès du GPU à sa mémoire est d'environ 6 fois supérieure à celle du CPU.

3. il n'y a pas de différence de prix entre un GPU et un CPU, alors que le GPU possède une performance plus importante que le CPU

1.3 Contributions

Lors de nos recherches, les travaux que nous avons effectués ont fait l'objet de deux publications :

- une publication à la conférence *SYMPA 2006*. Elle porte sur le comportement des opérateurs arithmétiques sur processeurs graphiques.
- une publication à la conférence *Real Number and Computer (RNC07)*. Elle porte sur l'émulation logicielle d'une arithmétique avec 44 bits de précision.

Chapitre 2

Contexte

2.1 La norme IEEE-754

La norme ANSI/IEEE-754 [5], datant de 1985, est une norme qui caractérise un système à virgule flottante. Elle définit le format de représentation, ou encore le comportement des opérateurs de base que sont l'addition, la multiplication, la division, ou encore la racine carrée. Elle a été créée dans le but d'homogénéiser les systèmes de représentation des nombres et de calcul pour augmenter entre autre la portabilité des programmes. Cette section présente les bases de la normes IEEE-754.

2.1.1 Présentation de la norme IEEE-754

La norme IEEE-754 définit entre autres les points suivants :

- la représentation des nombres en machine,
- la précision nécessaire de certaines opérations,
- les exceptions qui peuvent être rencontrées,
- les arrondis disponibles.

Ces derniers ont été créés dans le but de satisfaire ou faciliter de nombreuses propriétés telles que :

- conserver des propriétés mathématiques,
- faciliter la construction de preuves,
- rendre les programmes portables et déterministes d'une machine à une autre,
- rendre possible la gestion des exceptions,
- garantir l'unicité de la représentation (à l'exception du zéro signé).

2.1.2 Représentation et précision

La norme IEEE-754 est une norme de représentation pour les nombres à virgule flottante en base 2.

Un nombre flottant x est représenté dans une base de représentation étant fixé à 2 par trois champs : une mantisse m , un exposant e , et un signe s , de la façon suivante :

$$x = (-1)^s \cdot m \cdot 2^e$$

Le **signe** s permet de définir si le nombre est positif ou non en fonction de sa valeur.

L'**exposant** e est basé sur la représentation d'un entier biaisé . C'est-à-dire que si on possède N bits pour représenter notre exposant, on enlève $b = 2^{N-1} - 1$ à notre exposant. On dit alors que le biais est égal à b . L'exposant varie donc entre deux valeurs e_{min} et e_{max}

La **mantisse** m est un nombre à virgule fixe composé de n bits. Dans un soucis d'unicité, cette virgule se situe entre le premier et le second bit de la mantisse ($m = m_0, m_1 m_2 \dots m_{n-1}$). De ce fait, la mantisse aura pour valeur un nombre compris entre 0 et la valeur de la base (dans notre cas, la mantisse est inférieure à 2). De plus :

- Le premier bit m_0 est appelé **bit implicite** . Il est dit "implicite" car celui-ci n'est pas représenté, sauf pour les formats de représentations étendus. Il est entièrement déterminé par la valeur de l'exposant.
- Les autres bits représentent la **fraction** f ($f = m_1 m_2 \dots m_{n-1}$). C'est la représentation interne de la mantisse.

La norme impose au minimum deux formats : la simple précision codée sur 32 bits, et la double précision codée sur 64 bits. Elle propose aussi une précision étendue pour chaque format. Chaque champs est alors codé avec un certain nombre de bits résumé dans le tableau 2.1.2.

Précision	Nombres de bits	signe	mantisse	exposant	bit implicite
simple	32	1	23	8	oui
double	64	1	52	11	oui
simple étendu	≥ 43	1	≥ 32	≥ 11	non
double étendu	≥ 80	1	≥ 64	≥ 15	non

TAB. 2.1 – Taille des nombres d'après la norme IEEE-754

La correspondance entre la représentation binaire d'un flottant simple précision et sa valeur est représentée dans la figure 2.1

2.1.3 Nombres IEEE-754 et valeurs spéciales

Pour résoudre des problèmes tels que des dépassements de capacités ou des exceptions, la norme impose certaines représentations comme des infinis ou les *NaN* (Not a Number).

Le zéro signé La valeur *zéro* est représentée par une mantisse et un exposant nuls. Mais comme la valeur infini, le zéro est une quantité signée. La norme impose de plus que -0 et $+0$ soient égaux de manière à éviter tout comportement imprévisible.

Not a Number Cette forme spéciale désigne une exception qui peut être un résultat invalide comme $0/0$ par exemple. Cela ne stoppe pas pour autant le programme, et les calculs se poursuivent. Il existe deux types de *NaN* : Les *signaling NaN* (sNaN) et les *quiet NaN* (qNaN).

- Les *sNaN* signalent des exceptions lorsqu'ils sont fournis comme opérandes arithmétiques (par exemple, une variable non initialisée dans une opération). Ils ne sont jamais générés par une opération. On peut toutefois les créer manuellement en mémoire.
- Les *qNaN* représentent le résultat de certaines opérations invalides sur des infinis ou des NaN (par exemple $\sqrt{-1}$).

Ces deux dernières valeurs spéciales sont représentées par un exposant égal à 255, et une mantisse non nulle.

L'infini Cette forme spéciale signale un dépassement de capacité ou une opération telle que $1/0$. Comme le zéro signé, cette forme est une quantité signée. Elle est représentée par un exposant égal à 255, et une mantisse nulle.

2.1.4 Les exceptions

Les exceptions définies par la norme sont une information précieuse sur le comportement et le résultat d'une opération. Il existe 5 exceptions dites *collantes*, car une fois que l'une d'entre elles apparaît, il est impossible de s'en débarrasser sauf si l'utilisateur les supprime explicitement. De plus, l'utilisateur doit pouvoir lire ou écrire de telles exceptions. Ce sont :

- **overflow** ou dépassement vers l'infini. Elle apparaît lorsque le résultat d'une opération est plus grand que la valeur maximale représentable. Le résultat est alors $\pm\infty$.
- **underflow** ou dépassement vers 0. Elle apparaît lorsque le résultat est plus petit que la plus petite valeur représentable. Le résultat est alors soit 0, soit un nombre dénormalisé.
- **division par zéro**. Comme son nom l'indique, cette exception apparaît lorsque le diviseur est nul. Le résultat est alors $\pm\infty$.
- **résultat inexact**. Cette exception signifie que le résultat attendu de l'opération n'est pas représentable. Le résultat sera donc la valeur arrondie de l'opération vers un nombre représentable (ce qui peut conduire à d'autres exceptions).
- **opération invalide**. Elle apparaît lorsque le résultat d'une opération est égal à *NaN*. Le résultat est alors *NaN*.

2.1.5 Les arrondis

Le résultat r d'une opération arithmétique dont les opérandes sont des *nombre représentables* n'est pas forcément représentable. De ce fait, on doit arrondir le résultat. La norme IEEE-754 caractérise 4 modes d'arrondi :

- **arrondi vers $+\infty$** (noté $\triangle(r)$) : le résultat est le plus petit nombre représentable supérieur ou égal à r .
- **arrondi vers $-\infty$** (noté $\nabla(r)$) : le résultat est le plus petit nombre représentable inférieur ou égal à r .
- **arrondi vers 0** (noté $\mathcal{Z}(r)$) : le résultat est l'arrondi vers $+\infty$ de r si $r < 0$, sinon c'est l'arrondi vers $-\infty$ de r .
- **arrondi au plus près** (noté $\circ(r)$) : le résultat fourni est le nombre représentable le plus proche de r parmi les deux nombres consécutifs qui encadrent r (si r est au milieu de ces deux nombres, le choix se portera alors sur le nombre dont la mantisse est paire – voir [30]).

De plus, la norme requiert la propriété suivante pour ces opérations arithmétiques :

Propriété 1 (Arrondi correct) *Soient x, y des nombres représentables, \diamond le mode d'arrondi choisi et \bullet une des quatre opérations arithmétiques ($+$, $-$, \times , $/$). Le résultat obtenu lors du calcul de $(x \bullet y)$ doit être $\diamond (x \bullet y)$.*

L'arrondi correct est difficile à obtenir, aussi dans de nombreux cas, nous utiliserons la notion d'*arrondi fidèle* empruntée à Dekker [14], et complétée dans [13]. Cet arrondi a des conditions plus faibles :

Définition 1 (Arrondi fidèle) *L'arrondi fidèle d'un nombre réel correspond à l'un des deux nombres machine l'encadrant, et le nombre machine lui-même en cas d'égalité.*

2.1.6 Mesure de l'erreur

Le résultat des opérations sur des nombres représentables n'est pas forcément représentable, et il est donc arrondi. Cet arrondi provoque alors une erreur de calcul. Si l'on prend le résultat flottant arrondi r^* et le résultat réel r , on peut alors mesurer l'erreur commise en erreur absolue et en erreur relative.

Définition 2 (Erreur absolue) *L'erreur absolue E_a est l'écart entre la valeur réelle r et son approximation r^* :*

$$E_a = |r - r^*|$$

Définition 3 (Erreur relative) *L'erreur relative E_r est la différence entre la valeur réelle r et sa valeur arrondi r^* divisé par sa valeur exacte r :*

$$E_r = \frac{|r - r^*|}{|r|} = \frac{E_a}{|r|}$$

2.2 Les processeurs graphiques

2.2.1 Architecture et fonctionnement

La place de la carte graphique au sein de l'ordinateur

La carte graphique possède plusieurs unités : l'unité graphique (en anglais : *Graphics Processing Units*, ou GPU) et la mémoire graphique. Elle est un périphérique qui permet l'affichage d'images sur l'écran. Le GPU vient se connecter

via un bus à la carte mère, et plus particulièrement au *North Bridge*. Viennent également se connecter au *North Bridge* le processeur central (en anglais : *Central Processing Unit*, ou CPU) et la mémoire centrale (dit mémoire RAM). Le *North Bridge* est l'arbitre des communications entre ces différents périphériques auxquels il est connecté. Ainsi, la vitesse de communication entre le *North Bridge* et un des trois périphériques cités précédemment va varier. On peut voir sur la figure 2.3 les différentes vitesses de communication entre les différentes unités.

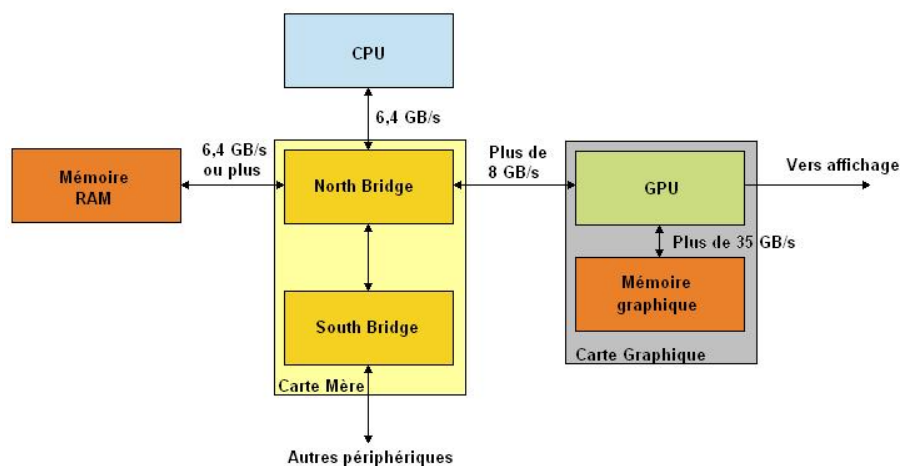


FIG. 2.3 – Vitesse de communication entre les différents éléments d'un ordinateur

On peut observer sur ce schéma que la vitesse d'accès à la mémoire graphique par le GPU est supérieure à la vitesse d'accès à la mémoire centrale par le CPU. On observe aussi que les données que l'on va traiter avec le GPU doivent se trouver dans la mémoire graphique. Il doit donc y avoir un envoi des données de la mémoire centrale vers la mémoire graphique par le CPU. Aussi si nous souhaitons utiliser le GPU pour effectuer des calculs, alors il y aura un gain uniquement si la quantité de calcul est suffisante pour couvrir le surcoût lié aux transferts de données entre le CPU et le GPU.

Modèle de fonctionnement des cartes graphiques : le pipeline graphique

Les GPU traitent des objets géométriques et des pixels. Les images sont créées en appliquant des transformations géométriques aux sommets et en découpant les objets en fragments ou pixels. Les calculs sont réalisés par différents étages de ce que l'on appelle le pipeline graphique, comme présenté dans la figure 2.4.

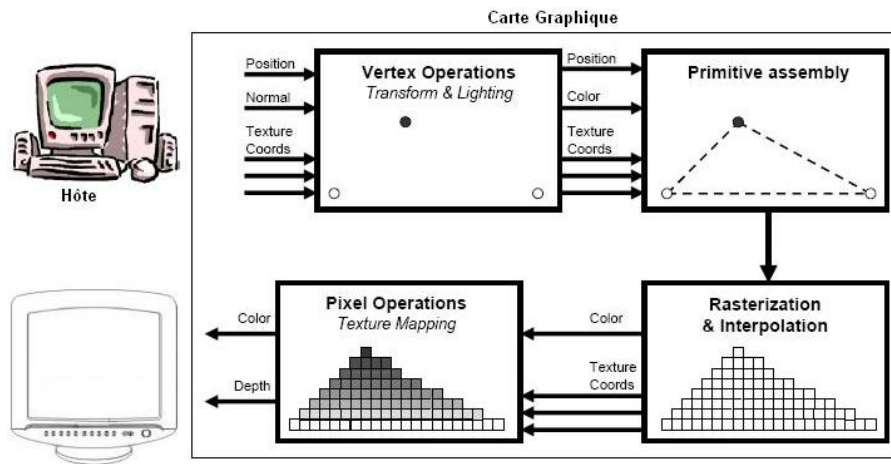


FIG. 2.4 – Vue d'ensemble d'un pipeline graphique

La machine hôte envoie des sommets pour positionner dans l'espace des objets géométriques primitifs (polygones, cercles, points, ...). Ces objets primitifs subissent des transformations (rotations, translations, ...) avant d'être assemblés pour créer un objet plus complexe. Ces opérations sont traitées dans l'unité de traitement des sommets (*vertex shader*).

Quand un objet a atteint sa position, sa forme et son éclairage finals, il est découpé en fragments ou pixels. Une interpolation est effectuée pour obtenir les propriétés de chaque pixel. Les pixels sont ensuite traités par l'unité de traitements des pixels (*pixel shader*) qui n'effectue aucune transformations géométriques, mais plutôt des tâches d'affichage comme par exemple appliquer une texture ou calculer la couleur d'un pixel. Cette dernière unité fait ses calculs depuis la mémoire sous la forme de *textures*, c'est-à-dire qu'elle fait des calculs sur des vecteurs matriciels et non pas linéaires comme sur CPU. Nous stockons des tableaux à deux dimensions dans de la texture car la structure optimale sur GPU est sous forme matricielle et non pas vectorielle. De plus amples informations sont données dans [28] et dans [19].

Le pipeline graphique effectivement implanté diffère légèrement de celui représenté dans la figure 2.4. Selon les cartes et les circuits, les constructeurs déplacent, partagent, dupliquent ou ajoutent certaines ressources. La figure 2.4 nous montrent les différentes étapes sur l'exemple d'un triangle. Les *vertex shaders* traitent 3 sommets alors que les *pixels shaders* traitent 86 pixels. Pour une figure donnée, le nombre de pixels est presque toujours plus important que le nombres de sommets et les architectures modernes contiennent plus d'unités de

traitements des pixels que d'unités de traitements des sommets. Le ratio actuel est par exemple de 24 pour 8 sur la carte graphique Nvidia 7800GTX ou 16 pour 8 sur la carte ATI RX1800XL.

Les shaders

Un *shader* est un programme pour paramétrer une partie du processus de rendu réalisé par une carte graphique. Il peut permettre de décrire des fonctions telles que l'absorption et la diffusion de la lumière, la texture à utiliser, le déplacement de primitives, ... Par la conception même du processus de rendu, les shaders sont les candidats idéaux pour une exécution en parallèle par les processeurs graphiques, permettent un traitement vectorisé qui soulage le CPU. Avec l'arrivée des nouvelles générations de GPU, les shaders sont devenus flexibles, efficaces et programmables.

Par abus de langage, on a appelé le *vertex shader* l'unité qui va traiter les sommets des primitives et le *pixel shader* l'unité qui va traiter les pixels.

Le Vertex Shader

La figure 2.5 tiré de [38] présente l'un des 8 *vertex shader* de la carte graphique Nvidia 7800GTX. Dans la classification de Flynn [17, 16], les 8 shaders fonctionnent en mode MIMD (*Multiple Instruction Multiple Data*) entre eux. Chaque *vertex shader* est capable d'initier à chaque cycle une opération MAD (*Multiply And Accumulate*) sur 4 données dans l'unité vectorielle et une opération '*special*' dans l'unité scalaire. Les opérations '*special*' implantées sont les fonctions exponentielles (exp, log), trigonométriques (sin, cos) et deux fonctions inverses ($1/x$ et $1/\sqrt{x}$). Avec l'arrivée de la version 3 du support matériel Direct3D, les *vertex shaders* sont capables d'accéder à la mémoire de texture grâce à une unité dédiée.

Direct3D [35] spécifie un certain nombre de prérequis sur les registres réservés dans les *vertex shaders* résumés dans le tableau 2.2.

Nom	registres d'entrées						registre de sortie
	v#	t#	c#	i#	b#	s#	o#
Type de registre	Input register	Temporary register	Constant float register	Constant Integer register	Constant boolean register	Sampler register	Output register
Compteur	16	12 (1)	256 (2)	16	16	4	12
Type de données	nombres flottants	nombres flottants	nombres flottants	entier	booléen	R	
Dimension	vecteur 4-D	vecteur 4-D	vecteur 4-D	vecteur 4-D	scalaire	1	
Permissions entrées/sorties	Lecture	Lecture	Définition	Définition	Définition	Utilisation	Écriture
Nombres de ports en lecture	1	3	1	1	1	1	N/A
Nombre de lecture par instruction	3	3	2	1	1	Non	N/A
Rel-Address	Non	Non	Non	a0 / aL	Non	Aucun	aL only
Défauts	Partial(0001)	Aucun	(0, 0, 0, 0)	(0, 0, 0, 0)	Faux	Oui	Aucun
DCL requis	Oui	Non	Non	Non	Non	Oui	Oui

TAB. 2.2 – Registres pour les vertex shaders

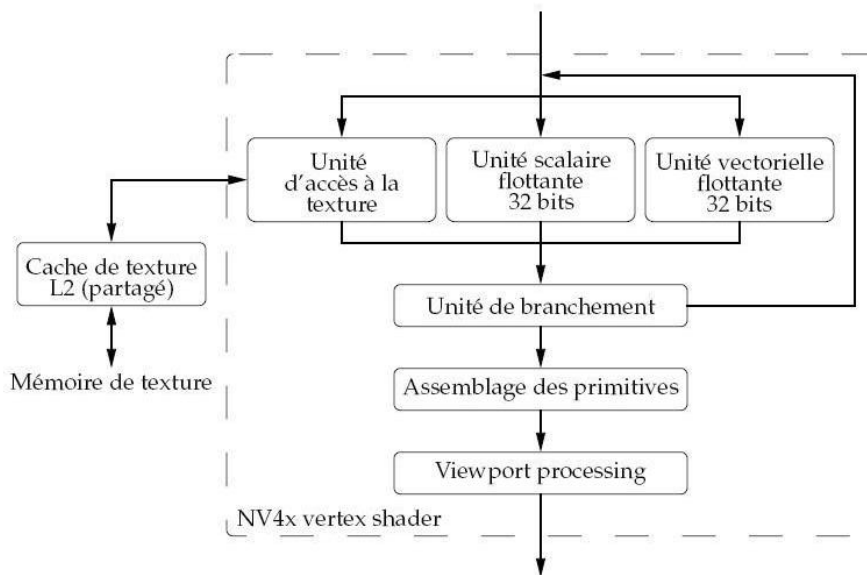


FIG. 2.5 – Architecture d'un vertex shader

On trouve dans ce type de *shader* différents registres qui peuvent être en lecture et/ou en écriture. Chacun de ces registres ne possède qu'un certain nombre de ports en lecture, c'est-à-dire qu'il n'y a qu'un certain nombre d'instructions pouvant accéder à un même registre. De même, le nombre de lecture par instruction signifie que lorsqu'une instruction accède à un registre, elle ne peut lire qu'un nombre limité de données. On retrouve des registres d'entrées, des registres temporaires pour stocker des valeurs que nous allons réutiliser. Les registres constants sont des registres inaltérables spéciaux qui stockent une constante. Les tentatives d'écrire à un registre constant sont illégales ou ignorées. En plus de ces registres, il existe un registre d'adresses, un registre pour les boucles, et un registre pour prédiquer les instructions conditionnelles..

Le Pixel Shader

La figure 2.6 décrit l'un des 24 *pixel shaders* de la Nvidia 7800GTX. Les 24 shaders fonctionnent en mode SIMD (*Single Instruction Multiple Data*) entre eux. La première unité exécute selon le programme 4 MAD ou un accès à la texture soit en lecture, soit en écriture, via l'unité de traitement des textures. Le résultat est envoyé à la deuxième unité flottante qui exécute 4 MAD. Dans le cas de la Nvidia 7800GTX, chaque pixel shader dispose d'une mémoire dédiée appelée cache de texture de niveau 1 et d'une unité capable de premiers traitements sur les textures.

Direct3D [34] spécifie un certain nombre de prérequis sur les registres réservés dans les *pixel shaders* résumés dans le tableau 2.3.

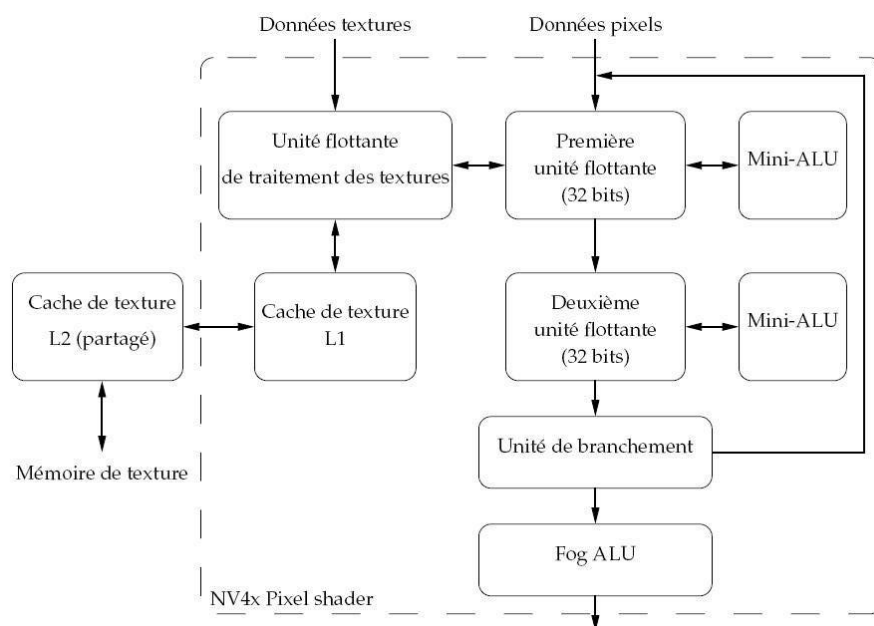


FIG. 2.6 – Architecture d'un pixel shader

Nom	registres d'entrées						registre de sortie
	v#	r#	c#	i#	b#	s#	oC#
Type de registre	Input Register	Temporary Register	Constant Float Register	Constant Integer Register	Constant Boolean Register	Sampler Register	Output color Register
Nombres	10	32	224	16	16	16	Voir multielement
Lecture/Écriture	Lecture	Lecture Écriture	Lecture	Lecture	Lecture	Lecture	Écriture
Nombres de ports en lecture		3	1	1	1	1	0
Nombre de lecture par instruction	Illimité	Illimité	Illimité	1	1	1	0
Dimension	4	4	4	4	1	4	4
RelAddr	aL only	N	N	N	N	N	N
Défauts	Partial(0001)	Aucun	0000	0000	FAUX	Dépend	Aucun
DCL requis	Oui	Non	Non	Non	Non	Oui	Non

TAB. 2.3 – Registres pour les pixel shaders

On trouve dans le *pixel shader* différents registres qui peuvent être en lecture et/ou en écriture. Chacun de ces registres ne possède qu'un certain nombre de ports en lecture, c'est-à-dire qu'il n'y a qu'un certain nombre d'instructions pouvant accéder à un même registre. De même, le nombre de lecture par instruction signifie que lorsqu'une instruction accède à un registre, elle ne peut lire qu'un nombre limité de données. On retrouve des registres d'entrées, des registres temporaires pour stocker des valeurs que nous allons réutiliser. Les registres constants lors de l'exécution du *fragment program* sont utilisés pour stocker les données nécessaires au *pixel shader*. En plus de ces registres, nous avons 4 registres d'entrées : un registre qui compte l'itération courante, un registre pour faire des comparaisons sur les vecteurs, un registre pour les coordonnées du pixel et un registre pour la position pixel sur la face avant ou arrière; et 1 registre de sortie qui conserve la profondeur du pixel en cours

2.2.2 Format flottant sur processeurs graphiques

Les GPU travaillent sur différents formats de représentation des nombres. Le tableau 2.4 reprend les formats implantés sur les cartes graphiques que nous avons étudiées. Le format sur 32 bits est lié à la version 3.0 de Direct3D. Ce format s'inspire du format simple précision de la norme ANSI-IEEE et ISO-IEC, mais les constructeurs ne garantissent pas la compatibilité de leur unité avec ces normes. La documentation disponible sur les cartes graphiques ne permet pas de déterminer en quoi ces implantations diffèrent de la norme.

Référence	Nombre de bits				Valeurs non numériques
	Total	Signe	Exposant	Fraction	
Nvidia	16	1	5	10	NaN, Inf
	32	1	8	23	
ATI	16	1	5	10	Absentes
	24	1	7	16	
	32	1	8	23	Non documentées
ANSI-ISO	32	1	8	23	NaN, Inf
	64	1	11	52	

TAB. 2.4 – Format de représentation des nombres à virgule flottante sur GPU et CPU

2.3 La programmation sur processeurs graphiques

2.3.1 Brook

Brook [1] est une extension de l'ANSI C et est conçu pour incorporer les idées de calcul de données en parallèle et puissance arithmétique. Le modèle général de calcul, désigné comme un flot, nous donnent deux avantages principaux comparés aux langages conventionnels traditionnels :

- *Parallélisme de données* : Permet aux programmeurs de définir des ensembles de données devant subir des traitements identiques.
- *Intensité arithmétique* : Encourage les programmeurs à spécifier les opérations sur les données qui vont minimiser les communications avec l'environnement global et maximiser les calculs locaux.

Brook est un langage de haut-niveau qui fonctionne comme une sur-couche d'une des deux API disponibles : OpenGL ou DirectX, ou du CPU. Voici un exemple 2.3.1 du mode de fonctionnement de programme écrit en Brook.

Exemple de programme Brook

```
1  #include <stdio.h>
2
3  kernel void sum(float a<>, float b<>, out float c<>){
4    c = a + b;
5  }
6
7  int main(int argc, char *argv[]) {
8    float data_a[100], data_b[100];
```

```

9   float a<100>, b<100>;
10  int i, j;
11
12  for ( i = 0; i < 10; i++ ) {
13    for ( j = 0; j < 10; j++ ) {
14      data_a[10 * i + j] = 0.125f * (float) (10 * i + j);
15      data_b[10 * i + j] = (float) 0;
16    }
17  }
18  streamRead(data_a, a);
19  streamRead(data_b, b);
20
21  sum(a, b, b);
22
23  streamWrite(b, data_b);
24
25  return 0;
26 }

```

Dans cet exemple, nous commençons par une phase de création du noyau de calcul (lignes 3 – 5) que l'on exécutera au choix sur le GPU ou sur le CPU. La fonction "main" (ligne 7 – 26) s'exécutera sur le CPU. Brook nous permet d'utiliser notre GPU comme un co-processeur. Dans cette fonction, nous déclarons 2 types de variables : l'un qui représente les vecteurs initialisé sur le CPU ligne 8). L'autre type correspond aux données que nous allons utiliser dans le GPU (ligne 9). Nous allons affecter des valeurs à ces variables grâce à la fonction "streamRead" (ligne 18 – 19). Nous effectuons nos calculs en faisant appel à notre fonction avec nos variables en paramètres (ligne 21), et nous récupérons nos données grâce à la fonction "streamWrite" (ligne 23).

Cet exemple met en évidence la simplicité d'utilisation d'un tel langage, mais aussi les inconvénients liés à son utilisation :

- suppressions de certains mots-clés (*static, goto, asm, ...*),
- restrictions sur les pointeurs (pas de pointeurs de pointeurs ou de pointeurs de fonctions, seuls * et -> sont autorisés),
- pas de récursion possible,
- pas de lecture ou écriture dans la mémoire globale,
- pas d'appels aux fonction de bibliothèques (*printf, malloc, ...*),
- pas contrôle précis des instructions (par exemple, on ne sait pas où et quand sont stockées, transférées et utilisées les données).
- ...

2.3.2 Les interfaces de programmations : OpenGL et DirectX

Une interface de programmation (en anglais : *Application Program Interface* ou API) a pour objet de faciliter le travail d'un programmeur en lui fournissant les outils de base nécessaires à tout travail à l'aide d'un langage donné. Elle constitue une interface servant de fondement à un travail de programmation

plus poussé. Il existe deux API pour la programmation sur cartes graphiques : OpenGL et DirectX.

OpenGL

OpenGL [3] (en anglais : *Open Graphics Library*) défini au départ par SGI, est une spécification qui définit une API multi plate-forme pour la conception d'applications générant des images 3D (mais également 2D). L'interface regroupe environ 250 fonctions différentes qui peuvent être utilisées pour afficher des scènes tridimensionnelles complexes à partir de simples primitives. Du fait de son ouverture, de sa souplesse d'utilisation et de sa disponibilité sur toutes les plate-formes, elle est utilisée par la majorité des applications scientifiques, industrielles ou artistiques 3D et certaines applications 2D vectorielles. Enfin, cette API se veut libre, et donc accessible par tous.

DirectX

Microsoft DirectX [2] est une suite d'API multimédia intégrée au système d'exploitation Windows permettant d'exploiter les capacités matérielles d'un ordinateur. DirectX fournit un ensemble de bibliothèques de fonctions essentiellement dédiées aux traitements audio/vidéo (carte vidéo, carte son, etc.) et aux périphériques d'entrée/sortie (joystick, carte réseau, souris, etc.).

L'avantage des fonctions de DirectX pour les programmeurs est que celles-ci utilisent (si possible) un algorithme alternatif (confié au processeur) quand le matériel installé ne gère pas ce type de traitement. Il fonctionne comme une surcouche de Windows, évitant théoriquement aux programmeurs de devoir s'occuper des différences matérielles qui existent entre les différents PC.

Enfin, DirectX est la propriété de la société Microsoft. Ce produit n'étant pas libre, les sources ne sont pas rendues publiques, contrairement à la bibliothèque OpenGL, concurrente de DirectX3D

2.3.3 Cg

Cg est un langage de programmation des shaders développé par Nvidia [36]. Il permet au moyen de l'une des deux API de développer son ou ses propres programmes à exécuter dans le vertex shader et le pixel shader. C'est un langage qui se veut générique car il suffit de choisir un profil adapté à l'API que l'on souhaite utiliser et à sa carte graphique. De façon similaire à *Brook*, l'exécution d'un programme dans les shaders nécessite plusieurs étapes :

1. Initialisation des bibliothèques et définition des états
2. Création d'une *texture 2D* (= tableau)
3. Création d'un *programme Cg* (= noyau de calcul)
4. Envoi des *données* dans la **texture** et du *programme* dans les **shaders**
5. Exécution de la méthode *Ping Pong* [20] pour réaliser les calculs
6. Copie des données de la texture vers un tableau

La première étape nous permet d'initialiser entre autres l'API que nous allons utiliser, et de définir les états du pipeline graphique. Cette première étape correspond à définir la zone de résultat et à faire en sorte que cette zone ne s'affiche pas lorsque l'on réalise du calcul scientifique sur *GPU*. La seconde étape consiste à générer une ou plusieurs textures qui vont contenir les données pour effectuer nos calculs. La troisième étape consiste à écrire les programmes qui vont être exécuter dans les shaders. Ensuite, nous envoyons les données et les programmes sur le GPU afin d'effectuer les calculs. La méthode *Ping Pong* (étape 5) est l'une des nombreuses astuces à utiliser lorsque l'on souhaite faire du calcul scientifique sur GPU. Elle résoud le problème des lectures/écritures dans un même espace mémoire, car la mémoire est soit en lecture, soit en écriture. Enfin, dans la dernière étape, nous récupérons nos données calculées depuis une texture dans le GPU vers un tableau.

Après avoir vu les étapes nécessaires au fonctionnement d'un programme Cg, comparons le coeur d'un programme Cg et l'équivalent C++. Nous avons choisi pour cela le programme SAXPY.

SAXPY en C++

```

1 void saxpy(float x[], float y[], float a, uint n){
2     uint i;
3     for(i = 0; i < n; i++){
4         y[i] += x[i] * a;
5     }
6 }
```

SAXPY en Cg

```

1 float4 saxpy( in float2 coords : TEXCOORD0,
2     uniform sampleRECT textureY ,
3     uniform sampleRECT textureX ,
4     uniform float alpha ) : COLOR    {
5     float4 y = texRECT(textureY, coords);
6     float4 x = texRECT(textureX, coords);
7     return y + alpha * x;
8 }
```

On peut observer de grosses différences entre un programme qui fonctionne sur CPU et un programme qui fonctionne sur GPU. Sur CPU, nous faisons directement en itérant sur les valeurs du tableau nos calculs, puis nous affectons la valeur du résultat dans l'adresse mémoire correspondant à $y[i]$ (ligne 2 – 5). Sur GPU, nous avons besoin d'aller chercher nos valeurs dans les textures à la coordonnée passée en paramètre (ligne 5 et 6), puis nous faisons nos calculs sans itération (ligne 7) et nous renvoyons notre résultat dans le paramètre couleur (ligne 4). D'autres exemples se trouvent dans [21] et [24].

Chapitre 3

Test des opérateurs arithmétiques

3.1 État de l’art

Avant la norme IEEE-754, les logiciels de test de l’arithmétique flottante sur CPU avaient pour but de découvrir les caractéristiques des fonctionnalités implantées [10, 22, 44]. Après 1985, année d’adoption de la norme, des logiciels sont apparus pour examiner la bonne implémentation de celle-ci avec une batterie de tests bien sélectionnés [29, 37, 49], et d’autres pour tester les fonctions élémentaires, spéciales ou complexes [12, 11]. Enfin, des logiciels comme UCBTest [4] tentent d’examiner la conformité des fonctionnalités normalisées, la qualité des autres fonctionnalités usuelles, et le bon fonctionnement de la chaîne de compilation.

Nous savons que les GPU ne respectent pas la norme IEEE-754, contrastant avec la majorité des CPU qui l’ont adopté. Pour adapter sur GPU un programme destiné à une exécution CPU, nous avons besoin de vérifier quelques hypothèses.

Il existe déjà un logiciel qui teste des propriétés de l’arithmétique sur GPU. Cet outil est un adaptation d’un sous-groupe de tests du logiciel *Paranoïa* [29], créé à la base pour CPU. Les résultats obtenus sur une Nvidia NV35, et une ATI R300 sont reportés dans le tableau 3.1

Opération	Arrondi exacte	Troncature	R300	NV35
Addition	$[-0.5, 0.5]$	$(-1, 0]$	$[-1.0, 0.0]$	$[-1.0, 0.0]$
Soustraction	$[-0.5, 0.5]$	$(-1, 1)$	$[-1.0, 1.0]$	$[-0.75, 0.75]$
Multiplication	$[-0.5, 0.5]$	$(-1, 0]$	$[-0.989, 0.125]$	$[-0.782, 0.625]$
Division	$[-0.5, 0.5]$	$(-1, 0]$	$[-2.869, 0.094]$	$[-1.199, 1.375]$

TAB. 3.1 – Erreur des opérations flottants exprimée en ulp obtenu lors de l’exécution de l’adaptation de *paranoïa*

Vous pourrez trouver en annexe (Annexe A) les résultats obtenus lors de l'exécution de paranoïa sur un Nvidia 7800GTX. Nous pouvons tirer plusieurs conclusions de telles données :

- L'addition et la soustraction sont tronquées sur les deux GPU.
- La soustraction semble bénéficier d'un bit de garde sur Nvidia, mais pas sur ATI.
- La multiplication possède un arrondi fidèle sur les deux GPU [43].
- L'erreur de division semble révéler que celle-ci est construite par la multiplication du dividende et de l'inverse arrondi du diviseur [8].

Nous n'avons pu obtenir le code source de cette adaptation, et de nombreuses questions restent sans réponses :

- Quels calculs sont effectués ?
- Est-ce que les calculs sont effectués dans le *vertex shader*, le *pixel shader* ou les deux ?
- Quels opérateurs sont testés ?
- ...

Enfin, nous avons essayé de faire fonctionner la version 32 bits de paranoïa sur une carte graphique ATI 32 bits (ATI RX1800XL), et le programme n'a pas fonctionné.

Ainsi, le développement de notre propre programme permettant les tests des opérateurs arithmétiques par l'implémentation d'algorithmes décrits par la suite semblait s'imposer.

3.2 Les algorithmes

Nous avons défini et implanté des algorithmes en ciblant le format 32 bits pour mieux comprendre le stockage des nombres flottants dans le registre et en mémoire, le fonctionnement de l'addition, et le fonctionnement de la multiplication. Dans un premier temps, ce programme a été écrit en OpenGL, avec l'utilisation des *Frame Buffer Object* pour le transfert et le stockage de données dans les textures. Ce dernier fonctionnait correctement sur des cartes de type Nvidia 7800GTX avec le driver *ForceWare 81.98*, mais ce n'était pas le cas pour des cartes de type ATI RX1800XL avec le driver *Catalyst 6.3*. Ainsi, une seconde version du programme a été écrite sous DirectX.

3.2.1 Test de représentation des nombres flottants dans la mémoire et les registres temporels

TEST 1 Le GPU est souvent utilisé comme co-processeur. Les données sont générées dans le CPU pour pouvoir être transférées dans la mémoire texture avant d'être traitées par le GPU. Il est intéressant de savoir si des conversions interviennent lors du transfert des données du CPU vers GPU. Nous avons envoyé des nombres dénormalisés, des NaN et des nombres infinis du CPU vers le GPU, pour ensuite les récupérer sur le CPU.

Sur certaines architectures (telles que les processeurs IA-32), les registres internes stockent les nombres avec une précision supérieure à celle des données en mémoire [27], et avec une dynamique plus grande sur l'exposant [33]. Nous avons voulu savoir si ce comportement existe aussi sur le GPU.

TEST 2 Tout d'abord, nous avons voulu tester si le nombre de bits d'exposant était supérieur dans les registres temporaires que le format proposé. Pour cela, nous avons utilisé des vecteurs pour calculer :

$$(\text{MAX_FLOAT} + \text{MAX_FLOAT}) - \text{MAX_FLOAT}$$

Avec MAX_FLOAT le plus grand nombre représentable sur GPU d'après le format des nombres flottants de celui-ci.

TEST 3 Nous avons voulu vérifier également le nombre de bits de mantisse en utilisant des vecteurs pour calculer :

$$1,5 + 2^{-i}$$

Nous pouvons voir dans le tableau suivant les résultats attendus, et nous voyons qu'à l'itération n, nous avons un égalité entre 1,5 et $1,5 + 2^{-i}$. Nous venons donc de trouver le nombre de bits de mantisse.

i	1,5	$1,5 + 2^{-i}$
1	1.10...0	10.000...0
2	1.10...0	1.110...0
3	1.10...0	1.101...0
⋮	⋮	⋮
n - 1	1.10...0	1.10...1
n	1.10...0	1.10...0

TEST 4 Nous avons voulu savoir si les différents MAD, où $\text{MAD}(x, y, z)$ effectue l'opération $x \times y + z$, conservent dans l'accumulateur plus de bits de mantisse du produit $x \times y$ que la précision de travail pour les utiliser quand les chiffres de poids forts du produit sont compensés par le troisième opérande z. C'est le cas des architectures normalisées récentes qui implémentent un FMA en arithmétique à virgule flottante [18, 33]. Nous avons généré des vecteurs de tests aléatoires et comparé le reste de la multiplication sur GPU avec la valeur exacte calculée sur CPU :

$$\text{MAD}(x, y, -x \otimes y)$$

où $x \otimes y$ est la multiplication arrondie au plus près au format simple précision sur CPU.

3.2.2 Test de l'addition

TEST 5 Lors de l'utilisation de l'adaptation de Paranoïa, nous avons remarqué que la soustraction sur les processeurs Nvidia semblait posséder 1 bit de garde. En l'absence d'informations supplémentaires, nous avons cherché à clarifier le comportement de la soustraction au niveau des *vertex* et des *pixel shaders* de l'ATI et de Nvidia. Les *pixel shaders* des deux architectures testées possèdent deux MAD cascades (voir page 12). Nous avons déterminé le comportement du premier additionneur en calculant :

$$1,5 - 2^{-i}$$

pour i variant de 1 à 64.

Dans le tableau suivant, nous pouvons observer les valeurs binaires des résultats attendues pour chaque itération, avec n correspondant au nombre de bits de mantisse que possède le format de représentation des nombres flottants.

i	1,5	$1,5 - 2^{-i}$
1	1.10...0	1.000...0
2	1.10...0	1.010...0
3	1.10...0	1.011...0
⋮	⋮	⋮
n	1.10...0	1.01...1
$n + 1$	1.10...0	1.01...1
⋮	⋮	⋮
$n + k - 1$	1.10...0	1.01...1
$n + k$	1.10...0	1.10...0

TEST 6 De façon similaire, nous avons testé le deuxième additionneur des *pixel shaders* en calculant l'opération suivante :

$$(1,5 - 2^{-i}) - 1,5$$

pour i variant de 1 à 64.

3.2.3 Test de la multiplication

TEST 7 Les résultats de paranoïa laissent à penser que la multiplication est tronquée à la fois sur ATI et sur Nvidia. Nous avons voulu tester la façon dont cette troncature était effectuée à l'intérieur des *pixel shaders*. Pour cela, nous avons souhaité déterminer si tous les produits partiels étaient générés comme présenté sur la figure 3.1 [45], ou si pour une question de réduction de coût, les bits de poids faible de certains produits partiels étaient ignorés. Dans ce cas, une constante est ajoutée à la somme des produits partiels pour réduire le biais statistique introduit par la troncature d'une quantité toujours positive.

i	1,5	$1,5 - 2^{-i}$	$(1,5 - 2^{-i}) - 1,5$
1	1.10...0	1.000...0	- 0.100...0
2	1.10...0	1.010...0	- 0.010...0
3	1.10...0	1.011...0	- 0.001...0
⋮	⋮	⋮	⋮
n	1.10...0	1.01...1	- 0.00...01
$n + 1$	1.10...0	1.01...1	- 0.00...01
⋮	⋮	⋮	⋮
$n + k - 1$	1.10...0	1.01...1	- 0.00...01
$n + k$	1.10...0	1.10...0	0

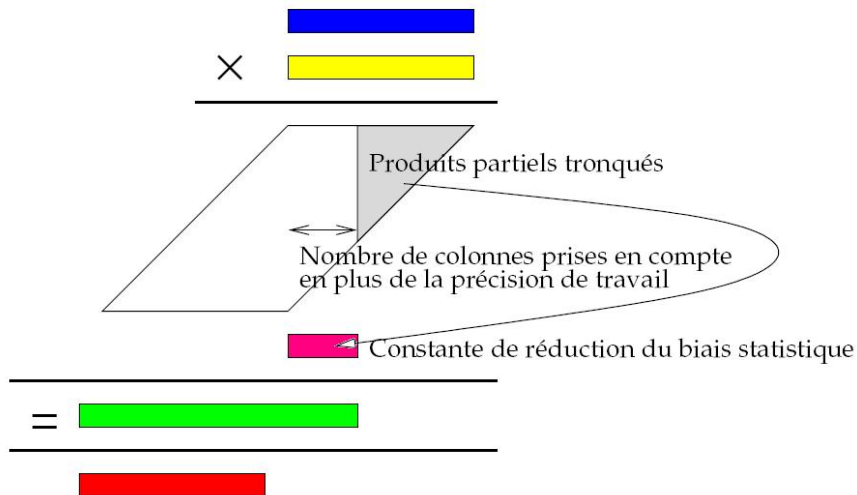


FIG. 3.1 – Produits partiels tronqués pour accélérer le multiplieur et réduire sa taille

Nous avons multiplié deux vecteurs de 24 nombres : un premier vecteur composé uniquement de la valeur $2^{23} + 3$, et un deuxième vecteur défini par la récurrence suivante pour $i > 0$:

$$y_1 = 1, y_{2i} = y_{2i-1} + 1, y_{2+1} = 4y_{2i-1} + 2$$

Cette suite peut aussi s'interpréter à l'aide d'opérateurs binaire ou logique ($()$) et décalage vers la gauche (\ll) en :

$$y_1 = 1, y_{2i} = y_{2i-1} | 01_2, y_{2+1} = (y_{2i-1} \ll 2) | 01_2$$

TEST 8 Ce deuxième vecteur alterne une valeur qui crée une chaîne de 1 sans propagation dans les bits de poids faible pour vérifier la troncature, avec une valeur générant une propagation de retenue à partir de la colonne la plus à droite jusqu'au premier bit du résultat.

Un deuxième test composé de deux vecteurs de 2^{23} nombres nous donne plus de précision. Il calcule le produit :

$$(2^{23} + 1) \times (2i + 1) \text{ pour } 0 < i < 2^{23}$$

Ce test détermine la valeur de la constante de réduction du biais statistique.

TEST 9 Enfin, un dernier test sera effectué, avec des vecteurs de tests de deux milliards de valeurs aléatoires sur ATI et Nvidia. Ce test va vérifier que les valeurs des opérations $A \times B$, $(-A) \times (-B)$, $-(A \times (-B))$ sont égales, ainsi que celles des opérations $(-A) \times B$ et $A \times (-B)$.

3.3 Les résultats

Tous ces tests¹ ont été implémenté en OpenGL et DirectX et exécutés sur une Nvidia 7800GTX et sur une ATI RX1800XL.

Test de représentation des nombres flottants dans la mémoire et les registres temporels

Les résultats obtenus par le *TEST 1* montrent que lors d'un transfert sans aucune opération, les nombres dénormalisés sont remplacés par 0, et que la valeur $\pm\infty$ n'est en aucun cas modifiée aussi bien sur la carte Nvidia et la carte ATI. Pour ce qui est de NaN, ils ne sont pas gérés de la même façon sur les deux processeurs :

- La carte ATI transforme les sNaN (*signaling NaN*) en qNaN (*quiet NaN*),
- La carte Nvidia ne fait aucune transformation sur cette valeur non numérique.

Enfin, l'exécution des *TEST 2, 3 et 4* montre qu'aucun des registres temporaires des *pixels* et des *vertex shaders* des cartes ATI et Nvidia n'utilise une plage d'exposant étendue pour éviter les débordements, ou un nombre de bits de mantisse supérieur pour augmenter la précision des calculs. Par ailleurs, aucun des MAD ne conservent le produit sur une précision étendue au delà des 24 bits de précision des nombres flottants simple précision.

Test sur l'addition

Nous avons remarqué que le résultat obtenu lors du *TEST 5* sur le premier additionneur était égal à 1,5 dès que $i \geq 26$ sur ATI et Nvidia dans les *vertex* et *pixel shaders*. Ce résultat laisse à penser que la première soustraction s'effectue avec un additionneur sur 26 bits. En ce qui concerne le deuxième additionneur

¹Ces résultats ont fait l'objet d'une publication [32] à la conférence SYMPA 2006.

du *pixel shader* (*TEST 6*), le résultat retourné était égal à 0 lorsque $i \geq 25$ sur ATI, et $i \geq 26$ sur Nvidia. Les deux additionneurs cascades des *pixel shaders* de la Nvidia se comportent donc de façon similaire. En revanche, nous pouvons penser que le cas où une seule addition est lancée dans les *pixel shaders* d'ATI, alors elle est effectuée par le deuxième additionneur qui dispose de 26 bits de précision, alors que lorsque deux additions sont lancées, les deux additionneurs sont utilisés et le premier ne dispose que de 25 bits de précision.

L'utilisation d'un seul bit de garde permet de ne pas commettre d'erreur d'arrondi lorsqu'on calcule la différence de deux nombres proches mais dont les exposants diffèrent d'une unité. En revanche, la présence d'un additionneur sur 26 bits dans une arithmétique tronquée peut faire que la propriété suivante n'est plus vérifiée [39, 40] :

$$e(x) - e(y) > t \longrightarrow x \oplus y = x$$

où $x \oplus y$ est le résultat de l'addition sur GPU, $e(.)$ la valeur de l'exposant de la variable considérée et t le nombre de bits de mantisse du format de travail. Cette propriété est nécessaire pour que la différence :

$$x + y - (x \oplus y)$$

soit toujours représentable en machine, sauf si il y a dépassement de capacité. Une recherche plus poussée devra être faite pour comprendre pourquoi les additionneurs des GPU testés disposent de deux bits de garde et non d'un seul.

Enfin, nous avons décidé de lancer les mêmes tests que précédemment mais en utilisant des flottants sur 16 bits au lieu de 32. Les résultats montrent que les additions sont effectués à précision maximale (26 bits de précision) pour ensuite être arrondie dans le format de destination souhaité (11 bits).

Test de la multiplication

Les résultats des tests de la multiplication (*TEST 7 et 8*) nous indiquent que le multiplieur des *pixel shaders* traite les produits partiels jusqu'à la 9^{me} colonne pour ATI, et jusqu'à la 6^{me} colonne pour Nvidia. Ces résultats sont confirmés par des tests aléatoires qui montrent qu'un biais est ajouté pour compenser les produits partiels manquants.

Enfin, selon toute probabilité, la notation signe-valeur absolue est utilisée et le multiplieur calcule de façon séparée le signe du résultat et sa valeur absolue (*TEST 9*).

Chapitre 4

Opérateurs flottants 44 bits

4.1 État de l'art

Beaucoup de processeurs obéissent à la norme IEEE-754 pour l'arithmétique flottante, qui définit le format simple et double précision. Mais pour des applications dans le domaine de la cryptologie par exemple, la précision fournie par le matériel n'est pas suffisante. Pour remédier à un tel problème, beaucoup de chercheurs ont essayé de développer un format multiprécision (il s'agit en fait d'un format de représentation avec une précision plus importante que celle proposé par le matériel, géré par le logiciel).

Sur CPU, il existe plusieurs bibliothèques multi-précision basées sur différents formats :

Bibliothèques basées sur une représentation entière Dans cette catégorie de bibliothèques, un nombre multiprécision est représenté par un tableau d'entier, représentable en machine (32 ou 64 bits). C'est le cas de la bibliothèque GMP [25], sur laquelle sont basées beaucoup d'autres bibliothèques (comme MPFR [41]). Ces bibliothèques permettent à l'utilisateur de choisir dynamiquement la précision qu'il lui est nécessaire pendant l'exécution du programme. D'autres, au contraire, choisissent de définir la précision au moment de la compilation.

Bibliothèques basées sur une représentation flottante Actuellement, la tendance veut que les CPU voient leurs opérateurs flottants optimisés. De ce fait, quelques bibliothèques comme MPFUN [6] exploitent ces opérateurs en utilisant un tableau de nombres flottants. Les autres utilisent une somme non évaluée de plusieurs nombres flottants double précision (les "double-double" de Briggs [7], les "quad-double" de Bailey [26], et les "expansions des nombres flottants" de Priest [42]). Cette représentation est basée sur les caractéristiques de la norme IEEE-754, en utilisant des algorithmes simples comme opérateurs. Malheureusement, ce format ne permet d'atteindre qu'une précision à coût raisonnable (quelques nombres flottants seulement) car la complexité des algorithmes augmentent de manière quadratique avec la précision.

Sur GPU, la précision disponible est limitée : par exemple, les couleurs sont représentées par des nombres de 8 bits. Avant l'apparition des "*shader 3.0*" qui nécessita l'ajout des nombres flottants 32 bits, les chercheurs et développeurs ont voulu remédier à ce problème de précision en développant des solutions logicielles, en extension à la précision matérielle.

Par exemple, Strzodska [48] proposa une représentation de nombres à virgule fixe d'une précision de 16 bits à partir d'opérations sur 8 bits. Dans ses recherches, il représente un nombre sur 16 bits sous la forme de la somme de 2 nombres de 8 bits. Selon l'auteur, les opérations dans ce format de représentation ne sont que 50% plus lente que les opérateurs d'origine.

4.2 Les algorithmes

Nous cherchons à atteindre la double précision. Pour cela, nous nous inspirons des travaux sur le format *double-double* décrit dans [7]. Pour nos algorithmes, nous avons choisi de représenter des nombres multi-précision comme des sommes non-évaluées de 2 nombres flottants représentables en matériel 4.1.



FIG. 4.1 – Exemple de représentation d'un nombre multiprécision

Dans cet exemple, nous avons voulu représenter un nombre multiprécision X par la somme non évalué de 2 nombres à virgules flottants A et B tels que :

- $X = A \oplus B$,
- $|B| \leq 2^{-24}|A|$.

Dans le coeur du GPU, le pipeline graphique possède de nombreuses unités de calculs. Ces unités ne sont pas conçues pour réaliser des tests et comparaisons de manière efficace. Alors, lorsque cela est possible, nous essayerons de remplacer ces tests par du calcul.

Nous présentons quelques propriétés et algorithmes de l'arithmétique flottante IEEE utilisée dans notre format. D'après les tests précédents, nous savons que l'addition et la multiplication sont tronqués, et que Nous possédons une soustraction sur 26 bits en interne. Nous considérons que nous utilisons des nombres flottants simple précision. Nous utiliserons $+$, $-$, \times et $/$ pour désigner les opérations arithmétiques usuelles, et \oplus , \ominus , \otimes et \oslash pour les opérations arithmétiques machines possédant une erreur d'arrondi.

Lemme 1 (Sterbenz [47, 23]) *si a et b sont des nombres flottants IEEE, et que $y/2 \leq x \leq 2Y$ alors $x \ominus y$ est calculé exactement sans erreur d'arrondi.*

Théorème 1 (Add12(Priest [42])) *Soient a et b des nombres flottants. L'algorithme suivant produit deux nombres flottants s et e de tel sorte que $a \oplus b = s + e$ exactement, où s représente une approximation de $a \oplus b$, et e représente l'erreur d'arrondi lors du calcul de s et que $|e| \leq 2^{-24}|s|$*

```

ADD12(a, b )
1  if (|b| ≥ |a|)
2      tmp = a;
3      a = b;
4      b = tmp;
5  endif
6  s = a ⊕ b;
7  d = s ⊖ a;
8  g = s ⊖ d;
9  h = g ⊖ a;
10 f = b ⊖ h;
11 e = f ⊖ d;
12 if e (⊕ d ≠ f)
13     s = a;
14     e = b;
15 endif
16 return (s, e )

```

Preuve La preuve de cet algorithme est dans la thèse de Priest ([42], page 14 – 17) lorsque l'on se trouve dans une arithmétique avec arrondi fidèle, ce qui est notre cas.

Théorème 2 (Split(Dekker [14])) *Soit a un nombre flottant sur (p) -bits, où $p \geq 3$. Choisissons un nombre de séparation s de telle sorte que $p/2 \leq s \leq p-1$. Alors l'algorithme suivant va produire une valeur a_{hi} sur $(p - s)$ -bits, et une valeur sans chevauchement a_{lo} sur (s) -bits tel que $|a_{hi}| \geq |a_{lo}|$, et $a = a_{hi} + a_{lo}$*

```

SPLIT(a )
1 c = ( $2^s \oplus 1$ )  $\otimes$  a;
2 abig = c  $\ominus$  a;
3 ahi = c  $\ominus$  abig;
4 alo = a  $\ominus$  ahi;
5 return (ahi , alo)

```

Preuve Cette preuve est une adaptation de la preuve de [46] avec les conditions observées sur GPU. La ligne 1 est équivalente au calcul de $2^s a \oplus a$, car multiplier par une puissance de 2 change son exposant. L'addition est sujette à un arrondi, aussi nous avons $c = 2^s a + a + \text{err}(2^s a \oplus a)$. La ligne 2 est sujette à un arrondi, d'où $a_{big} = 2^s a + \text{err}(2^s a \oplus a) + \text{err}(c \ominus a)$. A la fois $|\text{err}(c \ominus a)|$ et $|\text{err}(2^s a \oplus a)|$ sont inférieur à $ulp(c)$, donc l'exposant de a_{big} peut seulement être plus grand que celui de $2^s a$ si chaque bit significatif de a est non nul excepté les deux derniers bits. En vérifiant manuellement le comportement de *SPLIT* dans ces 4 cas, celui-ci peut vérifier que l'exposant de a_{big} n'est jamais plus grand que celui de $2^s a$. Alors $|\text{err}(c \ominus a)| \leq ulp(2^s a)$, et le terme d'erreur $\text{err}(c \ominus a)$ est représentable sur s bits. Grâce au lemme de Sterbenz la ligne 3 et 4 sont calculées exactement. On peut déduire que $a_{hi} = a - \text{err}(c \ominus a)$ et $a_{lo} = \text{err}(c \ominus a)$; la dernière expression exprimable sur s bits. Soit a_{hi} possède le même exposant que a , soit un exposant une fois plus grand, et dans ces deux cas, a_{hi} est représentable sur $p - s$ bits.

Théorème 3 (Mul12(Dekker [14])) *Soient a et b des nombres flottants normalisés. L'algorithme suivant produit deux nombres flottants normalisés p et e de tel sorte que $a \otimes b = p + e$, où p représente une approximation de $a \otimes b$, et e représente l'erreur d'arrondi lors du calcul de p et que $|e| \leq 2^{-24}|p|$*

```

Mul12(a , b )
1 p = a  $\otimes$  b;
2 (ah , al) = SPLIT (a);
3 (bh , bl) = SPLIT (b);
4 err1 = a  $\ominus$  (ah  $\otimes$  bh);
5 err2 = err1  $\ominus$  (al  $\otimes$  bh);
6 err3 = err2  $\ominus$  (ah  $\otimes$  bl);
7 e = (al  $\otimes$  bl)  $\ominus$  err3;
8 return (p , e);

```

Preuve La ligne 1 calcule $p = ab + \text{err}(a \otimes b)$ avec $\text{err}(a \otimes b)$ l'erreur de calcul de la multiplication. Comme toutes les autres multiplication et soustractions sont exactes et alors on calcule $e = -\text{err}(a \otimes b)$

Théorème 4 (Add22(Bailey [26])) *Soit $ah + al$ et $bh + bl$ les arguments dans notre format de l'algorithme suivant :*

```

ADD22(ah, al, bh, bl)
1 (t1, err1) = ADD12(ah, bh);
2 (err2, err3) = ADD12(al, bl);
3 (t2, err4) = ADD12(err1, err2);
4 (t3, err5) = ADD12(err3, err4);
5 (err6, u3) = ADD12(t2, t3);
6 (s, err7) = ADD12(t1, err6);
7 (e, err8) = ADD12(u3, err7);
8 return( s, e)

```

avec $(ah + al) + (bh + bl) = s + e$, avec que $|e| \leq 2^{-24}|s|$

Preuve Par définition de ADD12, nous avons :

$$\begin{aligned}
-t1 \oplus err1 &= ah + bh, & |err1| &\leq 2^{-24}(ah + bh) \\
-err2 \oplus err3 &= al + bl, & |err3| &\leq 2^{-24}(al + bl) \\
-t2 \oplus err4 &= err1 + err2, & |err4| &\leq 2^{-24}(err1 + err2) \\
& & &\leq 2^{-48}(ah + bh) + 2^{-24}(al + bl) \\
-t3 \oplus err5 &= err3 + err4, & |err5| &\leq 2^{-24}(err3 + err4) \\
& & &\leq 2^{-72}(ah + bh) + 2^{-48}(al + bl) + 2^{-48}(al + bl) \\
& & &\leq 2^{-72}(ah + bh) + 2^{-47}(al + bl) \\
& & &\leq (ah + bh)(2^{-72} + 2^{-71}) \\
-err6 \oplus u3 &= t2 + t3, & |u3| &\leq 2^{-24}((al + bl) + 2^{-24}(ah + bh) + 2^{-24}(al + bl) \\
& & &\quad + 2^{-24}(al + bl) + 2^{-48}(ah + bh)) \\
& & &\leq (al + bl)(2^{-46} + 2^{-24}) + (ah + bh)(2^{-48} + 2^{-72})
\end{aligned}$$

$$\begin{aligned}
-s \oplus err7 &= t1 + err6, & |err7| &\leq 2^{-24}((al + bl)(1 + 2^{-24}) + (ah + bh)(1 + 2^{-24} + 2^{-48})) \\
-e \oplus err8 &= u3 + err7, & |err8| &\leq (al + bl)(2^{-47} + 2^{-69}) + (ah + bh)(2^{-48} + 2^{-69})
\end{aligned}$$

$$\begin{aligned}
\text{Or } s \oplus e &= s + u3 + err7 - err8 \\
&= t1 + err6 + u3 - err8 \\
&= t1 + t2 + t3 - err8 \\
&= t1 + t2 + t3 + err4 - err8 - err5 \\
&= ah + bh + al + bl - err8 - err5 \\
&= ah + bh + al + bl + \varepsilon \\
&\quad \text{avec } \varepsilon = -err8 - err5
\end{aligned}$$

$$\begin{aligned}
\text{De plus, } |err5| + |err8| &\leq (ah + bh)(2^{-72} + 2^{-71}) + (al + bl)(2^{-47} + 2^{-69}) \\
&\quad + (ah + bh)(2^{-48} + 2^{-69}) \\
&\leq (ah + bh)(2^{-48} + 2^{-69} + 2^{-72} + 2^{-71}) + (al + bl)(2^{-47} + 2^{-69})
\end{aligned}$$

$$\begin{aligned}
\text{Par définition du ADD12} & \quad |al| \leq 2^{-24}|ah| \\
& \quad |bl| \leq 2^{-24}|bh|
\end{aligned}$$

$$\begin{aligned}
\text{Donc } |err5| + |err8| &\leq (ah + bh)(2^{-48} + 2^{-69} + 2^{-72} + 2^{-71}) + (al + bl)(2^{-47} + 2^{-69}) \\
&\leq 2^{-47}(ah + bh)
\end{aligned}$$

l'opérateur ADD22 possède donc en théorie une précision de 47 bits.

Théorème 5 (Mul22) Soit $ah + al$ et $bh + bl$ les arguments dans notre format de l'algorithme suivant :

```

Mul22( $ah$ ,  $al$ ,  $bh$ ,  $bl$ );
1  $t1$ ,  $t2$  = Mul12( $ah$ ,  $bh$ );
2  $t3$  = (( $ah \otimes bl$ )  $\oplus$  ( $al \otimes bh$ ))  $\oplus$   $t2$ ;
3 ( $p$ ,  $e$ ) = Add12( $t1$ ,  $t3$ );
4 return ( $p$ ,  $e$ );

```

avec $(ah + al) * (bh + bl) = p + e$, avec que $|e| \leq 2^{-24}|p|$

Preuve La preuve détaillé est proposé par Lauter dans [31] pour des cas particuliers du format double-double sur une architecture implantant la norme IEEE. La preuve de ce théorème avec les caractéristiques d'un GPU (simple précision, arrondi précis et bit de garde) est très similaire.

4.3 Les résultats

Nous avons implémenté avec le langage *Brook* les algorithmes cités précédemment. *Brook* est un langage de haut niveau développé pour le GPGPU. Ce langage nous a permis de tester¹ nos algorithmes dans des systèmes, pilotes et cartes graphiques variés avec peu de modifications.

Pendant nos tests, nous avons pu observer que lorsque nous utilisons la version *DirectX* générée par *Brook*, des optimisations interdites sur les opérations flottantes sont effectuées, ce qui n'est pas le cas avec la version *OpenGL*. Par exemple, la séquence d'opérations pour le calcul de l'erreur d'arrondi $r = ((a \oplus b) \ominus a)$ est remplacé par $r = b$. Pour résoudre ce problème, nous avons du modifié manuellement les *fragments programs* généré par *Brook*.

4.3.1 Le temps d'exécution

Il est très difficile de comparer les temps d'exécutions des opérateurs exécutés sur GPU, et sur ceux exécutés sur CPU. En effet, alors que les données sont déjà en mémoire pour le CPU, le GPU doit réceptionner les données de la mémoire globale vers sa mémoire locale avant de pouvoir effectuer les calculs. Pour avoir des comparaisons les plus fiables possibles, nous avons comparé les algorithmes proposés avec des opérateurs simple précision (addition, multiplication, multiplication et addition – MAD) sur CPU et GPU. Dans un soucis de clarté, nous avons normalisé les résultats obtenus par rapport au temps d'exécution de 4096 additions. Pour chaque version, nous avons effectué des tests avec différentes tailles de jeux de données. Les tests réalisés sur une carte graphique Nvidia 7800GTX avec une fréquence à 430 Mhz et 256 Mo de mémoire locale,

¹Ces résultats ont été soumis et acceptés dans [15] pour la conférence RNC07 (*Real Numbers and Computer*)

et un Pentium IV HT avec une fréquence à 3,2 Ghz ont été reporté respectivement dans les graphes 4.2 et 4.3. Vous trouverez dans les annexes les valeurs normalisées dans des tableaux (annexe B).

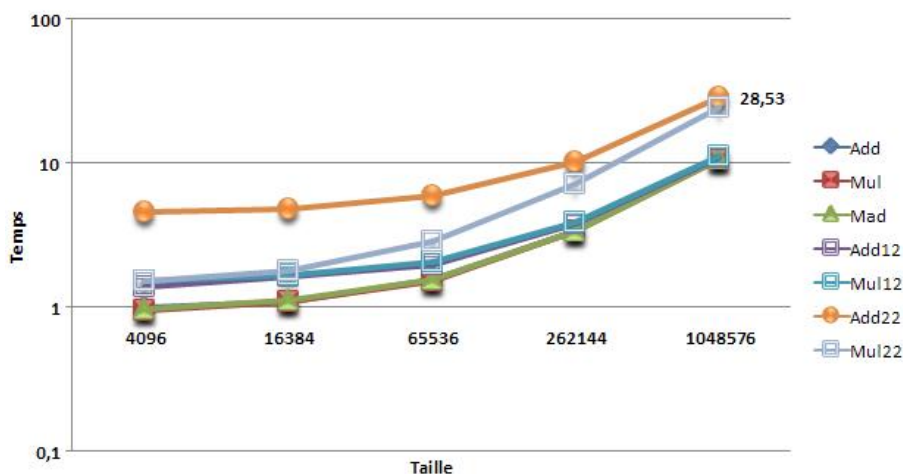


FIG. 4.2 – Comparaisons des temps d'exécution des opérateurs exécutés sur GPU

La différence entre l'exécution de 4096 additions sur GPU et l'exécution de 4096 additions sur CPU possède un facteur d'environ 100. Pour donner une idée, l'exécution de 4096 additions sur GPU et de l'échange de données entre le GPU et le CPU correspond à 100 fois l'exécution de 4096 additions sur CPU. Cette différence vient principalement de l'utilisation du bus du système pour la réception et l'envoi des données à traiter. Donc le GPU sera plus rapide que le CPU si le nombre d'opérations effectuées sur un même jeu de données relativement grand est très important.

La différence de temps entre un petit et un grand jeu de données est plus grande sur CPU que sur GPU. Cette différence est de 29 sur GPU et de 30000 sur CPU. Cela montrent que le GPU est plus efficace pour exécuter la même opération sur un grand jeu de données. Le temps d'exécution de ADD22 sur CPU est plus important que tous les autres opérateurs.

Nous observons que les temps d'exécution de l'addition, la multiplication, la multiplication et addition, le ADD12 et le MUL12 sont les mêmes sur le GPU. Le ADD22 et le MUL22 coûtent approximativement deux fois plus cher que les opérations basiques. Ce point prouve que les drivers des GPU arrive à pipeliner les instructions dépendantes de façon à minimiser le temps d'exécution

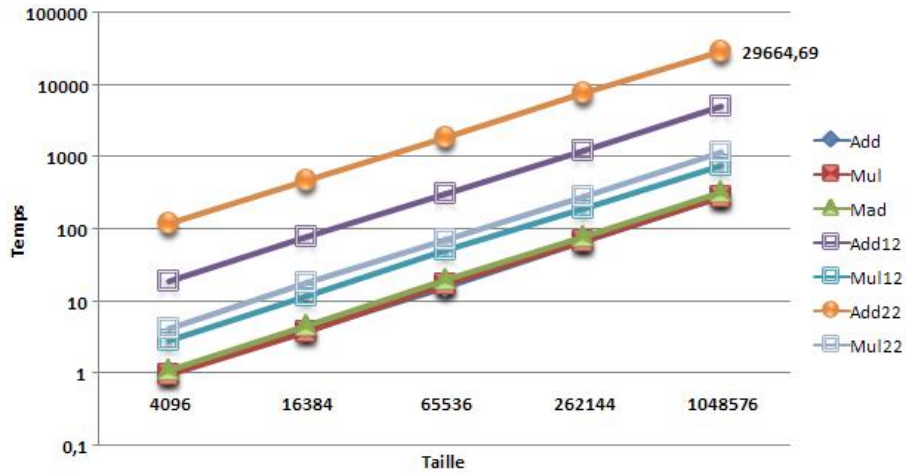


FIG. 4.3 – Comparaisons des temps d'exécution des opérateurs exécutés sur CPU

et maximiser l'utilisation des unités flottantes des GPU. Cela signifie aussi que le coût de ces opérations pourrait être plus élevé lors de leurs utilisations dans un vrai programme.

4.3.2 La précision

Nous avons exécuté nos algorithmes avec des vecteurs de tests aléatoires et mesuré l'erreur maximale à l'aide de la bibliothèque MPFR [41]. Pour ces tests, nous avons exclu les nombres dénormalisés qui ne sont pas supportés par le matériel ciblé. Le tableau 4.1 contient le nombre de bits de précision par opérateur

Opération	Erreur théorique	Erreur mesurée
Add12	(exact)	(exact)
Mul12	(exact)	(exact)
Add22	-47.0	-47.0
Mul22	-44.0	-45.0

TAB. 4.1 – Nombres de bits corrects par opération sur une carte Nvidia 7800GTX avec le nouveau format proposé

Chapitre 5

Conclusion

Ce stage de Master II recherche en informatique s'est articulé autour des méthodes, outils et preuves pour atteindre une bibliothèque haute précision sur processeurs graphiques. Nous avons dû chercher à comprendre par nous-même les caractéristiques de ces processeurs dernières générations pour optimiser les opérateurs de notre bibliothèque.

Contributions

Meilleure compréhension des GPU

Nous avons vu des algorithmes adaptés aux caractéristiques des opérateurs flottants des processeurs graphiques. Grâce à ces algorithmes, nous avons pu tester les propriétés des additionneurs et multiplieurs des *vertex* et *pixel shaders* de la Nvidia 7800GTX et de l'ATI RX1800XL. Nous avons, entre autre, montré que les registres temporaires stockent les nombres flottants uniquement sur 32 bits. Nous avons également déterminé le comportement des multiplieurs qui utilisent un biais pour compenser la troncature. Et enfin nous avons montré que les GPU disposent d'additionneurs sur 26 bits en interne ce qui a posé de nombreux problèmes dans les premières versions des opérateurs float-float développés.

Développement d'une bibliothèque avec des opérateurs sur 44 bits

Dans nos travaux, nous avons décrit un cadre général pour l'implémentation d'une émulation logicielle de nombres flottants avec une précision de 44 bits. Cette implémentation basée sur le langage *Brook* permet de manière simple et efficace de faire l'addition, la multiplication et le stockage de nombres flottants. L'intervalle des nombres représentables est le même que celui des nombres simple précision. Ces opérateurs hautes précisions nécessitent 2 fois plus de mémoire de texture. Cependant, ceux-ci se sont avérés rester assez rapide pour être utiliser comme des outils précis dans des algorithmes temps réel.

Perspectives

Poursuite des tests de l'arithmétique flottante sur processeurs graphiques

Il reste encore de nombreux tests complémentaires à écrire pour caractériser précisément le comportement des opérateurs déjà testés ainsi que ceux que nous n'avons pas testés (fonctions trigonométriques, logarithmiques et inverses).

Compléter et améliorer la bibliothèque haute précision

Des opérateurs restants doivent être implantés à notre bibliothèque haute précision, tels que les opérateurs inverse et racine carrée. De plus, grâce aux tests effectués ou à effectuer, il faut améliorer la qualité des algorithmes de manière à supprimer des opérations inutiles ou qui coûtent chers (tels que les branchements conditionnels).

Création d'une bibliothèque multi-précision

Après l'implémentation de notre bibliothèque haute précision, il nous faudra implémenter une bibliothèque multi-précision, de manière à effectuer des calculs avec la précision la plus grande possible, choisie ou non de façon arbitraire.

Calcul scientifique : transfert d'applications

Grâce à nos tests et à notre bibliothèque, il est intéressant de s'occuper du transfert d'applications du domaine scientifique vers les processeurs graphiques de manière à leur faire profiter de la puissance de calcul de ceux-ci, sans perte de précision. Transfert d'applications de la bio-informatique (BLAST) ou de la physique (rayonnement) par exemple.

Annexe A

Messages générés par Paranoïa pour la Nvidia 7800GTX

Microsoft Windows XP [version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

```
C:\Documents and Settings\Guillaume>cd "Mes documents\perpignan\recherche\GPGPU"  
C:\Documents and Settings\Guillaume\Mes documents\perpignan\recherche\GPGPU>Paranoia23.exe  
Created a 128x128 RenderTexture with BPP(32, 32, 32, 32)  
Created a 128x128 RenderTexture with BPP(32, 32, 32, 32)  
Created a 128x128 RenderTexture with BPP(32, 32, 32, 32)  
Created a 128x128 RenderTexture with BPP(32, 32, 32, 32)  
Created a 128x128 RenderTexture with BPP(32, 32, 32, 32)  
Created a 128x128 RenderTexture with BPP(32, 32, 32, 32)  
Created a 256x256 RenderTexture with BPP(32, 32, 32, 32)
```

```
Verifying U1 U2 F9 and B9  
VERIFY U1, U2, F9, B9, FAILURE CAN ALSO MEAN INCOMPLETE CARRY PROPOGATION  
U1 U2 F9 B9 Verified
```

FUZZY TEST

There are two pairs of conditions checked. One of the two must be true.
The first of the pair is $X \neq 1$. The second is $(X - 1/2) - 1/2 == 0$
No fuzziness detected in comparison

MULTIPLICATION GUARD BIT TESTS

This first checks that $1*x$ and $x*1$ behave the same
Then it checks if $(1+U2)*2$ and $2*(1+U2)$ behave the same
Multiplication guard bit tests: passed passed passed passed
MULTIPLICATION: Seems to have guard bit

MULT ACCURACY TEST

Checks multiplication accuracy per line 1980
MULTIPLICATION: Accuracy tests passed

DIVISION GUARD BIT TESTS

The following three division guard bit tests are from line 2000
DIVISION: $1/(1-U2) - (1 + U2) == 0$ test: failed
DIVISION: $1/3 == 3/9$ test: passed
DIVISION: $3/9 == 9/27$ test: passed
These are tests of $X/1 == 1$ from line 2040 and $1/(1+U2) < 1$ from 2070
DIVISION: $F9/1 == F9$ test: passed
DIVISION: $(1+U2)/1 == (1+U2)$ test: passed
DIVISION: $1/(1+U2) < 1$ test passed
DIVISION: FAILURE: Division lacks guard digit

ADDITION/SUBTRACTION GUARD BIT TESTS

Subtraction guard bit tests: passed passed passed
SUBTRACTION: Seems to have guard bit

SUBTRACTION COMPARISON TESTS

Tests for consistency in comparison and subtraction.
Specifically either $(1-U1) == 0$ or $(1-U1) - 1 < 0$.
Subtraction comparison tests (at least one of the following must pass): failed passed
SUBTRACTION: Comparison and Subtraction consistent (2170)

MULTIPLICATION ROUNDING TESTS

Tests on line 2390
 $(1.5-U2)(1+U2) = 1.5 + 0.5*U2 - U2^2 \rightarrow 1.5$ test: failed
 $(1.5+2*U2)(1-U2) = 1.5 + 0.5*U2 - U2^2 \rightarrow 1.5$ test: passed
 $(1.5-2*U2)(1+U2) = 1.5 - 0.5*U2 - 2*U2^2 \rightarrow 1.5 - U2$ test: failed
 $(1.5+U2)(1-U2) = 1.5 - 0.5*U2 - U2^2 \leq 1.5 - U2$ test: passed
Multiplication rounds up when it should round down (first four tests)
Tests based on calculations on line 2400
 $(1.5+U2)(1+U2) = 1.5 + 2.5*U2 + U2^2 \rightarrow$ up to $1.5+3*U2$ so EXACT
 $(1.5-2*U2)(1-U2) = 1.5 - 3.5*U2 + U2^2 \rightarrow$ Rounded to $1.5-4*U2$, so CHOPPED
 $(1.5 + 2*U2)*(1+U2) = 1.5 + 3.5*U2 + U2^2 \rightarrow$ up to $1.5+4*U2$ so EXACT
 $(1.5-U2)*(1-U2) = 1.5 - 2.5*U2 + U2^2 \rightarrow$ Rounded to $1.5-3*U2$, so CHOPPED
 $(1+2*U2)(1-U2) = 1 + U2 - 2*U2^2 \rightarrow$ Rounded to $1.0+1*U2$, so NEITHER
 $(1+U2)(1-U2) = 1-U2^2 \rightarrow$ up to 1 so EXACT
MULTIPLICATION: Is neither chopped nor correctly rounded

DIVISION ROUNDING TESTS

Tests on line 2480
DIVISION rounding: $(1.5+u2+u2)/(1+u2) - 1.5 \leq 0$ test: failed
 $(1.5+u2+u2)/(1+u2) = 1.500000119$ sign = 0 biased exponent = 127 mantisa = 400001
DIVISION rounding: $((1.5-U2-U2)-1.5)/(1-U2) - (1.5-U2-U2) \leq 0$ test: failed
DIVISION rounding: $((1.5+U2+U2)+U2)/(1+U2) \leq 1.5+U2$ test: failed
DIVISION rounding: $(1.5-U2-U2)/(1-U2) \leq 1.5-U2$ test: passed
DIVISION failed first 4 tests, rounding up where it should round down,
and is therefore neither clamped nor correctly rounded

Tests based on calculations on line 2490
 DIVISION rounding (X on 2490): $1.5/(1+U2) - (1.5-U2)$ test: $X==0$ so EXACT
 DIVISION rounding (Y on 2490): $(1.5-U2)/(1-U2)$ test: $Y==0$ so EXACT
 DIVISION rounding (Z on 2490): $(1.5+U2)/(1+U2) - 1.5$ test: $Z==0$ so EXACT
 DIVISION rounding (T on 2490): $1.5/(1-U2) - (1.5+U2+U2)$ test: $T==0$ so EXACT
 DIVISION rounding: $(1+U2+U2)/(1+U2) - (1+U2) == 0$ test: $Y2 < 0$ so CHOPPED
 DIVISION rounding: $(F9-U1)/F9 - 0.5 == F9-0.5$ test:
 Neither exact ($Y1==F9-0.5$) nor chopped ($Y1 < F9-0.5$)
 DIVISION: Is neither chopped nor correctly rounded

SUBTRACTION ROUNDING TESTS

Tests on line 2620: If both are true, then Paranoia presumes chopping
 $1-U1*U1$ -> to 1, (false)
 $1+U2*(0.5-U2) = 1+0.5*U2 - U2^2$ -> 1 (true)
 Tests on line 2650. Failure means not correctly rounded.
 $1+(0.5+U2)*U2 = 1+0.5*U2 + U2^2$ -> 1 (not correct)
 $1+(0.5-U2)*U2 = 1 + 0.5*U2 - U2^2$ -> 1 (correct)
 Tests on line 2670. Failure means not correctly rounded.
 $1-(0.5+U2)*U1 = 1 - 0.5*U1 - U2*U1$ -> F9 (correct)
 $1 - (0.5-U2)*U1 = 1 - 0.5*U1 + U2*U1$ -> 1 (correct)
 $S = (X+Y) + (Y-X) == 0$ test (line 2720): passed
 SUBTRACTION: Is neither chopped nor correctly rounded

SUBTRACTION ROUNDING TESTS - LESS MACs

This is the same set of subtraction rounding tests, but some effort has been put into getting rid of some multiply - accumulates. This is done by computing operands on the CPU, so if there is a serious problem with GPU add/subtract, at least the operands won't get messed up.
 Tests on line 2620: If both are true, then Paranoia presumes chopping
 $1-U1*U1$ -> to 1, (false)
 $1+U2*(0.5-U2) = 1+0.5*U2 - U2^2$ -> $1+1*U2$ (false)
 Tests on line 2650. Failure means not correctly rounded.
 $1+(0.5+U2)*U2 = 1+0.5*U2 + U2^2$ -> $1+U2$ (correct)
 $1+(0.5-U2)*U2 = 1 + 0.5*U2 - U2^2$ -> 1 (correct)
 Tests on line 2670. Failure means not correctly rounded.
 $1-(0.5+U2)*U1 = 1 - 0.5*U1 - U2*U1$ -> F9 (correct)
 $1 - (0.5-U2)*U1 = 1 - 0.5*U1 + U2*U1$ -> $1-1*U1$, (not correct)
 $S = (X+Y) + (Y-X) == 0$ test (line 2720): passed
 SUBTRACTION(less MACs): Is neither chopped nor correctly rounded

TESTING $X*Y == Y*X$

MULTIPLICATION: 10 pairs commuted

----BEGINNING NON-PARANOIA TESTS----

EXTRA MULTIPLICATION TESTS

This setup presumes chopping
 $(1+2*U2)(1-2*U2) = 1+4092*U1$
 $(1+4*U2)(1-4*U2) = 1+2046*U2$

$(1+2*U2)(1+U2) = 1+4100*U1$
 $(1+3*U2)(1+2*U2) = 1+0*U1$

EXTRA ADDITION/SUBTRACTION TESTS

This setup presumes chopping

$1-U1 < 1$

$1-F9*U1 = F9$

$1-0.5*U1 = 1$

$1-F9*0.5*U1 = 1$

SUBTRACTION: Appears to have guard bit, no round bit, and rounds operands up at least

----BEGINNING ERROR MEASUREMENT TESTS----

Testing Multiply Combinations

Exhaustive Sets progress:

.....

Error bound for mult= [-0.78125,0.625] ULPs

Testing Division Combinations

Exhaustive Sets progress:

.....

Error bound for div= [-1.19902,1.37442] ULPs

Testing Subtraction Combinations

Exhaustive Sets progress:

.....

Error bound for sub= [-0.75,0.75] ULPs

Testing Addition Combinations

Exhaustive Sets progress:

.....

Error bound for add= [-1,0] ULPs

SUMMARY

Profile = fp40

Readback seems to work

U1 U2 F9 B9 Verified

----BEGINNING PARANOIA TESTS----

No fuzziness detected in comparison

MULTIPLICATION: Seems to have guard bit

MULTIPLICATION: Accuracy tests passed

DIVISION: FAILURE: Division lacks guard digit

SUBTRACTION: Seems to have guard bit

SUBTRACTION: Comparison and Subtraction consistent (2170)

MULTIPLICATION: Is neither chopped nor correctly rounded

DIVISION: Is neither chopped nor correctly rounded

SUBTRACTION: Is neither chopped nor correctly rounded

SUBTRACTION: $(X-Y) + (Y-X) == 0$ test passed

SUBTRACTION(less MACs): Is neither chopped nor correctly rounded

SUBTRACTION(less MACs): $(X-Y) + (Y-X) == 0$ test passed

MULTIPLICATION: 10 pairs commuted

----BEGINNING NON-PARANOIA TESTS----

SUBTRACTION: Appears to have guard bit, no round bit, and rounds operands up at least sometimes

----BEGINNING ERROR MEASUREMENT TESTS----

Error bound for mult= [-0.78125,0.625] ULPs

Error bound for div= [-1.19902,1.37442] ULPs

Error bound for sub= [-0.75,0.75] ULPs

Error bound for add= [-1,0] ULPs

Annexe B

Temps d'exécution des opérateurs sur 44 bits

Taille	Add	Mul	Mad	Add12	Mul12	Add22	Mul22
4096	1	0,96	0,98	1,4	1,51	4,56	1,52
16384	1,1	1,1	1,12	1,64	1,66	4,8	1,79
65536	1,55	1,54	1,57	1,98	2,09	5,99	2,87
262144	3,41	3,37	3,39	3,81	3,95	10,34	7,17
1048576	10,75	10,67	10,7	11,06	11,2	28,53	24,33

TAB. B.1 – Comparaisons des temps d'exécution des opérateurs exécutés sur GPU

Taille	Add	Mul	Mad	Add12	Mul12	Add22	Mul22
4096	1	0,96	1,14	19,36	2,92	118,08	4,28
16384	3,99	3,87	4,66	77,53	12,13	477,01	18,08
65536	15,66	16,99	19,66	309,7	50,62	1843,58	71,16
262144	68,66	69,12	80,69	1242,99	194,69	7737,58	285,53
1048576	285,62	279,17	324,62	4986,21	780,67	29664,69	1143,66

TAB. B.2 – Comparaisons des temps d'exécution des opérateurs exécutés sur CPU

Bibliographie

- [1] Brook. In <http://graphics.stanford.edu/projects/brookgpu/>.
- [2] DirectX. In <http://www.microsoft.com/windows/directx>.
- [3] Opengl. In <http://www.opengl.org/>.
- [4] UCBTest. In <http://www.netlib.org>.
- [5] IEEE standart for binary floating-point arithmetic. In *ANSI/IEEE Standard, Std 754-1985, New York*, 1985.
- [6] David H. Bailey. A Fortran-90 based multiprecision system. *ACM Transactions on Mathematical Software*, 21(4) :379–387, 1995.
- [7] Henry Briggs. The doubledouble library. In <http://members.lycos.co.uk/keithmbriggs/doubledouble.html>, 1998.
- [8] Nicolas Brisebarre, Jean-Michel Muller, and Saurabh Kumar Raina. Accelerating correctly rounded floating point division when the divisor is known in advance. *IEEE Transactions on Computers*, 53(8) :1069–1072, 2004.
- [9] Cem Cebenoyan. Floating point specials on the gpu. Technical report, Nvidia, 2005.
- [10] William J. Cody. MACHAR : a subroutine to dynamically determine machine parameters. *ACM Transactions on Mathematical Software*, 14(4) :303–311, 1988.
- [11] William J. Cody. Algorithm 714 : CELEFUNT : a portable test package for complex elementary functions. *ACM Transactions on Mathematical Software*, 19(1) :1–21, 1993.
- [12] William J. Cody. Algorithm 715 : SPECFUN – a portable Fortran package of special function routines and test drivers. *ACM Transactions on Mathematical Software*, 19(1) :22–30, 1993.
- [13] Marc Daumas and David W. Matula. Validated roundings of dot products by sticky accumulation. *IEEE Transactions on Computers*, 46(5) :623–629, 1997.
- [14] Theodorus J. Dekker. A floating point technique for extending the available precision. *Numerische Mathematik*, 18(3) :224–242, 1971.
- [15] Guillaume Da Graça et David Defour. Implementation of float-float operators on graphics hardware. Mai 2006.
- [16] Michael Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12) :1901–1909, 1966.
- [17] Michael Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, 21(9) :948–960, 1972.

- [18] Freescale Semiconductor. *DSP56800 : Family Manual*, 2005.
- [19] Nico Galoppo, Naga K. Govindaraju, Michael Henson, and Dinesh Manocha. LU-GPU : efficient algorithms for solving dense linear systems on graphics hardware. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, page 3, Seattle, Washington, 2005.
- [20] Dominik Göttsche. Playing ping pong with render-to-texture.
- [21] Dominik Göttsche. GPGPU - basic math tutorial. In <http://www.mathematik.uni-dortmund.de/goeddeke/gppu/tutorial.html>.
- [22] W. Morven Gentleman and Scott B. Marovitch. More on algorithms that reveal properties of floating point arithmetic units. *Communications of the ACM*, 17(5), 1974.
- [23] David Goldberg. What every computer scientist should know about floating point arithmetic. *ACM Computing Surveys*, 23(1) :5–47, 1991.
- [24] GPGPU. Tutorial 0 : "hello gppu". In <http://www.gppu.org/developer/>.
- [25] Tobjörn Granlund. *The GNU multiple precision arithmetic library*, 2004. Version 4.1.3.
- [26] Yozo Hida, Xiaoye S. Li, and David H. Bailey. Algorithms for quad-double precision floating point arithmetic. In Neil Burgess and Luigi Ciminiera, editors, *Proceedings of the 15th Symposium on Computer Arithmetic*, pages 155–162, Vail, Colorado, 2001.
- [27] Intel. *Pentium II Processor : Developer's Manual*, 1997.
- [28] Naga Govindaraju Mark Harris Jens Krüger Aaron E. Lefohn John D. Owens, David Luebke and Timonthy J. Purcell. A survey of general-purpose computation on graphics hardware. *Eurographics 2005*, 2005.
- [29] Richard Karpinski. PARANOIA : a floating-point benchmark. *Byte*, 10(2) :223–235, 1985.
- [30] Donald E. Knuth. *The Art of Computer Programming : Seminumerical Algorithms*. Addison-Wesley, 1997. Third edition.
- [31] Christoph Quirin Lauter. Basic building blocks for triple-double intermediate format. Technical report, LIP, Ecole nationale Supérieure de Lyon, Septembre 2005.
- [32] Guillaume Da Graça et David Defour Marc Daumas. Caractéristiques arithmétiques des processeurs graphiques. 2006.
- [33] Peter Markstein. *IA-64 and elementary functions : speed and precision*. Prentice Hall, 2000.
- [34] Microsoft msdn. Registers - ps_3_0. In http://msdn.microsoft.com/archive/default.asp?url=/archive/en-us/directx9_c/directx/graphics/reference/shaders/pixelshader3_0/pixelshader3_0.asp.
- [35] Microsoft msdn. Registers - vs_3_0. In http://msdn.microsoft.com/archive/default.asp?url=/archive/en-us/directx9_c/directx/graphics/reference/shaders/vertexshader3_0/vertexshader3_0.asp.
- [36] Nvidia. Cg. In http://developer.nvidia.com/page/cg_main.html.
- [37] Michael Parks. Number theoretic test generation for directed rounding. In Israel Koren and Peter Kornerup, editors, *Proceedings of the 14th Symposium on Computer Arithmetic*, pages 241–248, Adelaide, Australia, 1999.

- [38] Matt Pharr. *GPUGems 2 : Programming Techniques for High-Performance Graphics and General-purpose Computation*. 2005.
- [39] Michèle Pichat. *Contributions à l'étude des erreurs d'arrondi en arithmétique à virgule flottante*. PhD thesis, Université Scientifique et Médicale de Grenoble, Grenoble, France, 1976.
- [40] Michèle Pichat and Jean Vignes. *Ingénierie du contrôle de la précision des calculs sur ordinateur*. Editions Technip, 1993.
- [41] Polka. *The multiple precision floating point reliable library*, 1999. Version 1.0.
- [42] Douglas M. Priest. Algorithms for arbitrary precision floating point arithmetic. In Peter Kornerup and David Matula, editors, *Proceedings of the 10th Symposium on Computer Arithmetic*, pages 132–144, Grenoble, France, 1991.
- [43] Douglas M. Priest. *On properties of floating point arithmetics : numerical stability and the cost of accurate computations*. PhD thesis, University of California at Berkeley, Berkeley, California, 1992.
- [44] N. L. Schryer. A test of computer's floating-point arithmetic unit. Technical report 89, AT&T Bell Laboratories, 1981.
- [45] Michael J. Schulte and Earle E. Swartzlander. Truncated multiplication with correction constant. In *Proceedings of the 6th IEEE Workshop on VLSI Signal Processing*, pages 388–396. IEEE Computer Society Press, 1993.
- [46] Jonathan R. Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. In *Discrete and Computational Geometry*, volume 18, pages 305 – 363, 1997.
- [47] Pat H. Sterbenz. *Floating point computation*. Prentice Hall, 1974.
- [48] R. Strzodska. Virtual bit precise operations on rgba8 textures. In *Vision, Modeling and Visualization*, pages 171 – 178, 2002.
- [49] Brigitte Verdonk, Annie Cuyt, and Dennis Verschaeren. A precision and range independent tool for testing floating-point arithmetic II : conversions. *ACM Transactions on Mathematical Software*, 27(1) :119–140, 2001.

Index

- , 7
- ◇, 7
- ⊖, 27
- ⊕, 27
- ⊗, 27
- ⊗, 27

- API, 14
- arrondi au plus près, 7
- arrondi correcte, 7
- arrondi fidèle, 7
- arrondi vers $+\infty$, 7
- arrondi vers $-\infty$, 7
- arrondi vers 0, 7

- bit implicite, 4
- Brook, 13

- Cg, 15

- DirectX, 15

- entier biaisé, 4
- erreur absolu, 7
- erreur relative, 7
- exception division par zéro, 6
- exception opération invalide, 6
- exception résultat inexact, 6
- exposant, 4

- fraction, 4

- GPU, 7

- Infini, 6
- interface de programmation, 14

- mantisse, 4

- nombres dénormalisés, 5
- nombres normalisés, 5

- Not a Number, 6

- OpenGL, 15
- overflow, 6

- pipeline graphique, 8
- pixel shader, 9, 11

- représentation des nombres IEEE, 4

- shader, 10
- signe, 4

- texture, 9

- underflow, 6

- vertex shader, 9, 10

- zéro signé, 6

Résumé

Les unités graphiques (*Graphics Processing Units*, ou GPU) sont devenues avec le temps des processeurs alliant puissance et flexibilité. En effet, dans les dernières générations de GPU, ces derniers ont été modifiés de telle façon qu'ils contiennent aujourd'hui des unités programmables de traitements des primitives (appelées *vertex shader*) et des pixels (appelées *pixel shader*). Ces unités programmables supportent des opérations à virgule flottante sur 8, 16 ou 32 bits. Cette dernière précision correspond à la simple précision de la norme IEEE sur l'arithmétique flottante (IEEE-754). De plus, les GPU actuels sont bien adaptés à l'exécution d'applications avec un important parallélisme de données. Pourtant, les GPU ne sont que peu utilisés par les applications numériques actuelles (*General Purpose on GPU*, ou GPGPU) pour plusieurs raisons. D'abord, nous ne disposons que peu d'informations techniques fournies par les fabricants (ATI et Nvidia), plus particulièrement sur l'implémentation des différents opérateurs arithmétiques à virgules flottantes embarqués dans les différentes unités de traitement. De ce fait, nous ne pouvons estimer et contrôler les erreurs de calcul, ou mettre en œuvre des techniques de réduction ou de compensation des erreurs d'arrondi par exemple, ce qui nous permettrait de faire en sorte d'augmenter la faible précision des GPU pour le calcul numérique. Enfin, les applications numériques actuelles fonctionnent avec des nombres flottants d'une précision de 64 bits, précision actuellement indisponible sur les GPU. Dans ce rapport, nous allons dans un premier temps chercher à découvrir certaines caractéristiques des GPU au travers d'un ensemble de tests, puis dans un deuxième temps essayer d'augmenter de manière logicielle la précision de calcul sur GPU.

Abstract

The Graphics Processing Unit (*GPU*) is now powerful and flexible processor. The latest generation contains multiple programmable vector units of treatments of data, what support 32 bits floating-point operations. Unfortunately, without informations about processors by builders, and with a weak precision, the transfert of numeric applications on GPU is not already possible. We propose a set of tests of arithmetic operators adapted to the characteristics of GPU and we will present their results. We also show how we have risen the precision of floating-point calculations for addition and multiplication with tiny overcost.