Caractéristiques arithmétiques des processeurs graphiques

Marc Daumas, Guillaume da Graça, David Defour

Caractéristiques arithmétiques des processeurs graphiques

Marc Daumas, Guillaume Da Graça et David Defour

DALI-LP2A (Université de Perpignan)
52 avenue Paul Alduy — 66860 Perpignan — France
{marc.daumas, guillaume.dagrac, david.defour}@univ-perp.fr

LIRMM (CNRS, Université de Montpellier 2)
161 rue Ada — 34392 Montpellier Cedex 5 — France
marc.daumas@lirmm.fr

Résumé

Les unités graphiques (*Graphic Processing Units* — GPU) sont désormais des processeurs puissants et flexibles. Les dernières générations de GPU contiennent des unités programmables de traitement des sommets (*vertex shader*) et des pixels (*pixel shader*) supportant des opérations en virgule flottante sur 8, 16 ou 32 bits. La représentation flottante sur 32 bits correspond à la simple précision de la norme IEEE sur l'arithmétique en virgule flottante (IEEE-754). Les GPU sont bien adaptés aux applications avec un fort parallélisme de données. Cependant ils ne sont que peu utilisés en dehors des calculs graphiques (*General Purpose computation on GPU* — GPGPU). Une des raisons de cet état de faits est la pauvreté des documentations techniques fournies par les fabricants (ATI et Nvidia), particulièrement en ce qui concerne l'implantation des différents opérateurs arithmétiques embarqués dans les différentes unités de traitement. Or ces informations sont essentielles pour estimer et contrôler les erreurs d'arrondi ou pour mettre en œuvre des techniques de réduction ou de compensation afin de travailler en précision double, quadruple ou arbitrairement étendue. Nous proposons dans cet article un ensemble de programmes qui permettent de découvrir les caractéristiques principales des GPU en ce qui concerne l'arithmétique à virgule flottante. Nous donnons les résultats obtenus sur deux cartes graphiques récentes : la Nvidia 7800GTX et l'ATI RX1800XL.

1 Introduction et présentation des unités graphiques (GPU)

Les unités graphiques (GPU) sont des unités spécialisées dans le calcul intensif et régulier qui développent une puissance de calcul bien supérieure à celle disponible sur les processeurs généralistes [24]. Avec l'arrivée des GPU de dernière génération, il devient intéressant d'utiliser ces processeurs GPU pour faire des calculs généralistes (GPGPU) [21] [1]. Les GPU deviennent alors des processeurs spécialisés pour des applications régulières et à fort parallélisme de données [20].
Après une présentation des unités graphiques (section 1.1) et des différentes implantations de l'arithmétique à virgule flottante (section 1.2), cette introduction se termine par un état de l'art des travaux liés aux tests des propriétés flottantes d'un système (section 1.3). Nous présentons ensuite nos algorithmes, leurs implantations et les résultats obtenus (section 2).

1.1 Modèle de fonctionnement des cartes graphiques : le pipeline graphique

Les GPU traitent principalement 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 à la figure 1. La machine hôte envoie des sommets pour positionner

[1] Voir aussi le site `http://www.gpgpu.org/`.

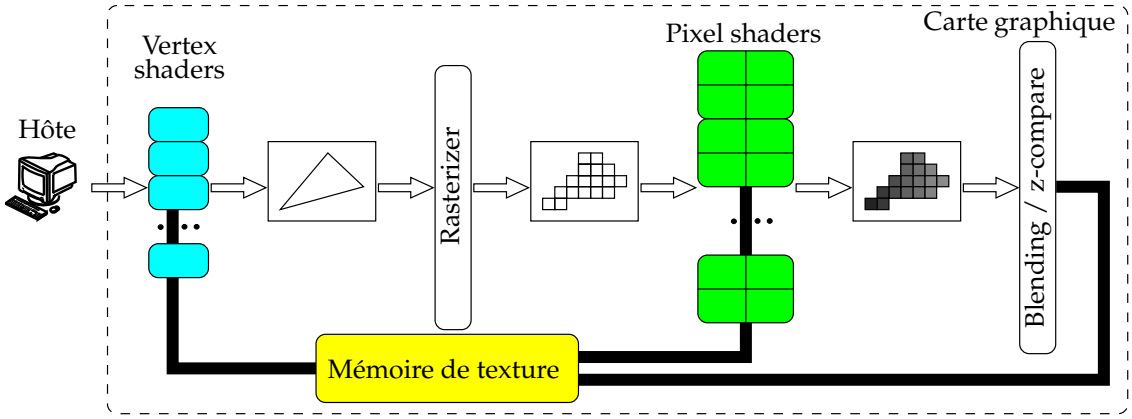

FIG. 1 – Vue d'ensemble du pipeline graphique

dans l'espace des objets géométriques primitifs (polygones, lignes, points). Ces objets primitifs subissent des transformations (rotations, translation, illumination...) avant d'être assemblés pour créer un objet plus complexe. Ces opérations sont réalisées 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 traitement des pixels (*pixel shader*) qui n'effectue pas des transformations géométriques mais des tâches d'affichage comme par exemple appliquer une texture ou calculer la couleur d'un pixel.
Le pipeline effectivement implanté diffère légèrement de celui représenté dans la figure 1. Selon les cartes et les circuits, les constructeurs déplacent, partagent, dupliquent ou ajoutent certaines ressources. La figure montre les différentes étapes sur l'exemple d'un triangle. Les vertex shaders traitent 3 sommets alors que les pixel shaders traitent 17 pixels. Pour une figure donnée, le nombre de pixels est presque toujours plus important que le nombre de sommets et les architectures modernes contiennent plus d'unités de traitement des pixels que d'unités de traitement des sommets. Le ratio actuel est par exemple de 24 pour 8.
La figure 2 reprise de [24] présente l'un des 8 vertex shaders de la Nvidia 7800 GTX. Dans la classification de Flynn [12, 13], les 8 shaders fonctionnement en mode MIMD (*Multiple Instruction Multiple Data*) entre eux. Chaque vertex shader est capable d'initier à chaque cycle une opération *Multiply and Accumulate* (MAD) sur 4 triplés 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 Direct 3D, les shaders sont capables d'accéder à la mémoire de texture grâce à une unité dédiée.
La figure 3 décrit l'un des 24 pixel shaders de la Nvidia 7800 GTX. Les 24 shaders fonctionnent en mode SIMD (*Single Instruction Multiple Data*) entre eux. La première unité flottante exécute selon le programme 4 MAD ou un accès à la texture 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 7800 GTX, 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.

1.2 Implantation sur GPU de l'arithmétique à virgule flottante en regard des normes ANSI et ISO

Les GPU travaillent sur différents formats de représentation des nombres. L'arithmétique à virgule flottante offre un spectre de nombres représentables plus étendu que l'arithmétique entière

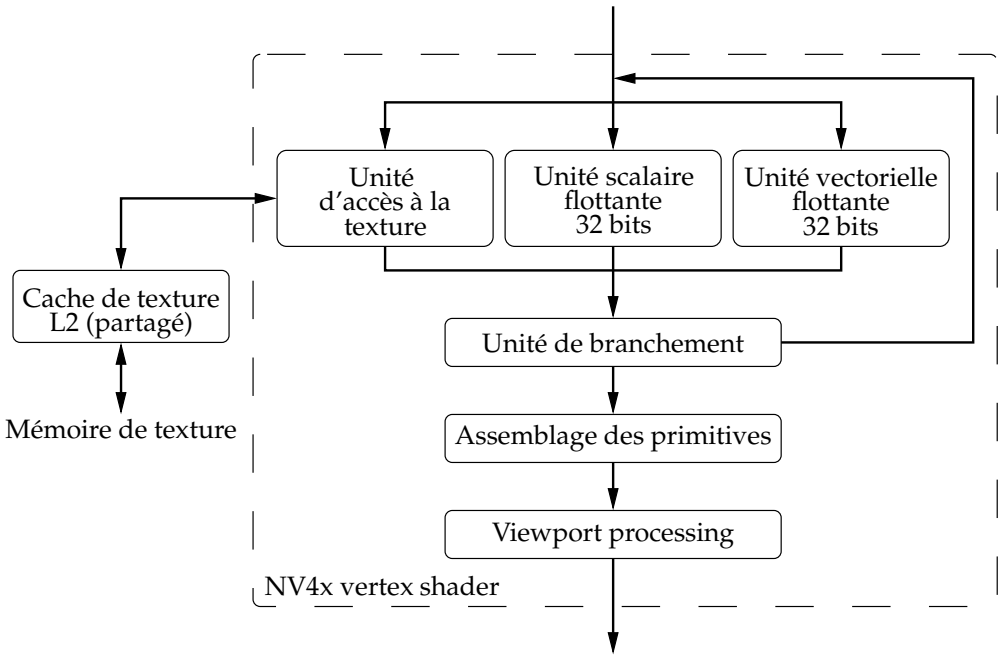

FIG. 2 – Détails d'un vertex shader présent dans la Nvidia 7800GTX

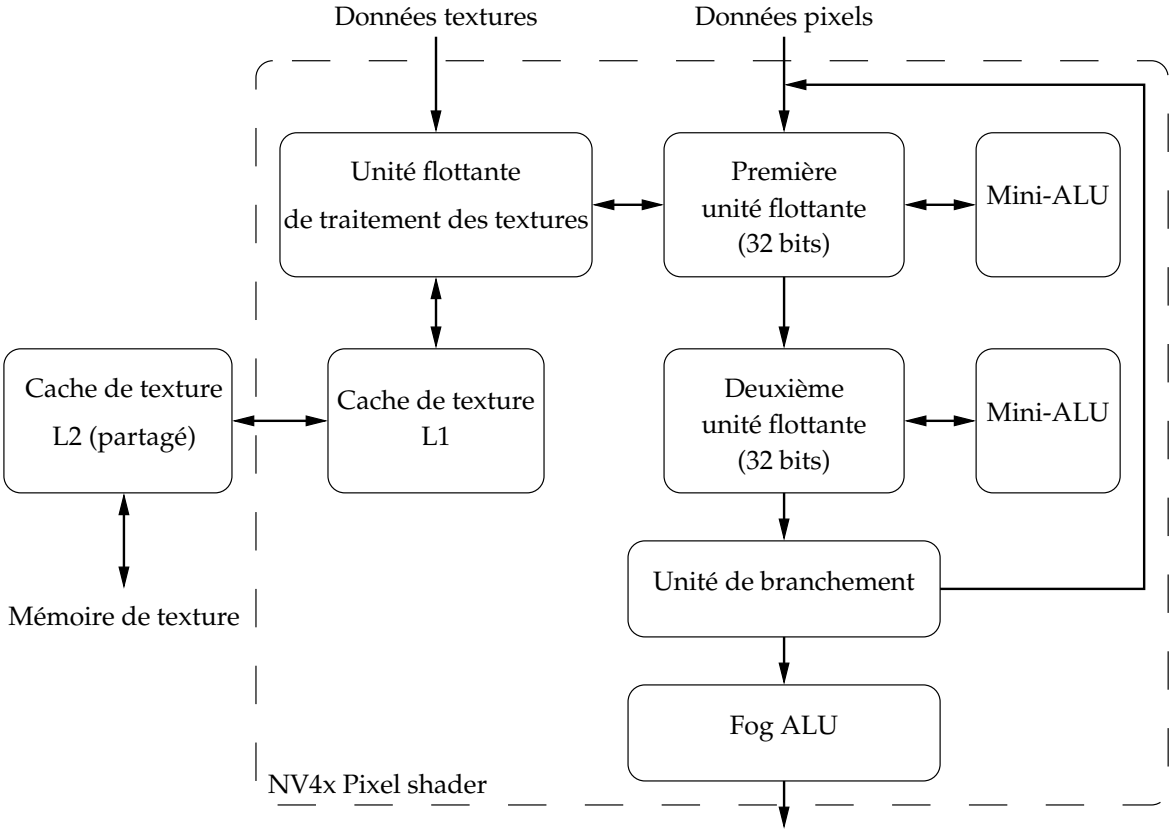

FIG. 3 – Détails d'un pixel shader présent dans la Nvidia 7800GTX

TAB. 1 – Format de représentation des nombres à virgule flottante sur GPU et CPU

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	

ou à virgule fixe sans que le programmeur ait besoin de gérer manuellement décalages et recadrages. L'implantation de l'arithmétique à virgule flottante [10] est normalisée par deux normes américaines en cours de révision[2] IEEE-ANSI 754 [31, 32] et 854 [8] et par une norme internationale IEC-ISO 60559 [1]. Des implantations très différentes de l'arithmétique à virgule flottante existent [2] mais elles ne sont pas utilisées par les GPU.
Les normes précédentes représentent un nombre par trois champs, une mantisse m, un exposant e et un signe s, la base de l'exposant est fixée à 2 ou à 10. Les GPU utilisent tous la base 2. La mantisse est un nombre en virgule fixe avec 1 bit avant la virgule, l'exposant est un entier biaisé. Les normes précédentes imposent que le bit de la mantisse à gauche de la virgule soit égal à 1 ce qui est toujours possible sauf si le nombre à représenter est trop petit. On parle selon les cas de représentation normalisée ou dénormalisée. Pour économiser de la mémoire, on ne stocke pas ce premier bit de la mantisse égal à 1 pour les nombres normalisés et on appelle fraction f les bits de la mantisse à droite de la virgule. Ainsi, un nombre flottant normalisé x, s'écrit

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

Les normes définissent deux formats présentés dans le tableau 1. Le format simple précision qui est codé sur 32 bits et le format double précision qui est codé sur 64 bits. Pour traiter les dépassements de capacité ou les situations exceptionnelles (division par zéro), les normes définissent les valeurs non numériques que sont les infinis ou les non-nombres (*Not a Number* — NaN).
Le résultat d'une opération arithmétique ($+$, $\times$, $/$, $\sqrt{}$) n'est presque jamais représentable bien que les opérandes sont représentables. Il faut l'arrondir. Les normes imposent que l'on utilise un des quatre modes d'arrondi prédéfinis : par défaut, par excès, tronqué, ou au plus près. Des travaux anciens [9] montrent que l'on peut arrondir précisément les quatre opérations usuelles précédentes, en utilisant uniquement trois bits supplémentaires par rapport au format du résultat. C'est-à-dire comme si les opérations intermédiaires avaient été infiniment précises. On parle des bits *guard*, *round* et *sticky*.
Une règle d'arrondi au plus proche ne définit pas un résultat unique quand le résultat intermédiaire exact se situe à égales distances entre deux nombres représentables. Pour des raisons statistiques, l'arrondi normalisé favorise le nombre dont la mantisse se termine par un 0 et on parle d'arrondi pair [28].
Le tableau 1 reprend les formats implantés sur les cartes graphiques que nous avons étudiées. Les données concernant le support pour les valeurs non numériques sont issues de [4]. Le format sur 32 bits est lié à la version 3.0 de Direct 3D. Ce format s'inspire du format simple précision des 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. Il est donc nécessaire de mieux comprendre le fonctionnement de l'arithmétique flottante sur GPU pour maîtriser le comportement numérique des applications scientifiques exécutées sur GPU.

[2] Voir le site `http://grouper.ieee.org/groups/754/`.

1.3 État de l'art et travaux antérieurs sur CPU et sur GPU

Les premiers logiciels réalisés pour tester l'arithmétique à virgule flottante sur CPU avaient pour unique but de découvrir les caractéristiques des fonctionnalités implantées [5, 15, 29]. Suite à l'adoption massive de la norme IEEE-754, certains logiciels sont apparus pour vérifier la bonne implantation de cette norme par une série de tests bien choisis [19, 23, 33] alors que d'autres proposaient d'implanter et de tester des fonctions élémentaires, spéciales ou complexes [6, 7]. Certains logiciels tels UCBTest[3] testent à la fois la conformité des fonctionnalités normalisées, la qualité des autres fonctionnalités usuelles et le bon fonctionnement de la chaîne de compilation.

Comme nous l'avons évoqué à la section précédente, les GPU ne respectent pas la norme IEEE-754 contrairement à la majorité des CPU. Certaines hypothèses sur le comportement arithmétique des opérateurs flottants doivent être vérifiées avant de pouvoir porter un programme pour CPU sur un GPU comme, par exemple, émuler des opérateurs arithmétiques en précision multiple [16].

Paranoia [19] est un outil qui teste certaines propriétés de l'arithmétique flottante des CPU. Un sous-ensemble de ces tests a été adapté pour pouvoir être exécuté sur certains GPU [17]. Les résultats complets du test de la carte graphique Nvidia 7800GTX sont publiés en annexe. En plus des binaires du logiciel, les auteurs fournissent un tableau de synthèse sur l'ATI R300 [17]. À ce jour, ce programme ne fonctionne pas sur notre ATI RX1800XL. Ses sources ne sont pas disponibles ce qui nous empêche d'aller plus loin que les seuls messages générés par son exécution. Par exemple, nous ne pouvons pas déterminer quels opérateurs ont été testés parmi tous les opérateurs disponibles dans les vertex et pixel shaders et quels vecteurs de test ont été utilisés au delà de *Paranoia*.

Certaines caractéristiques se dégagent de ces premiers tests :
– L'addition et la multiplication sont tronquées sur les deux GPU.
– La soustraction semble bénéficier d'un bit de garde sur Nvidia mais pas sur ATI.
– La multiplication implante un arrondi fidèle sur les deux GPU [27].
– L'erreur de division semble indiquer que la division est implantée par la multiplication du dividende par une approximation de l'inverse du diviseur [3].

En conclusion, les travaux initiés montrent que de nombreuses questions restent sans réponse et que les réponses ne pourront être apportées que par un logiciel dont les sources sont disponibles. C'est le cas pour les logiciels cités précédemment et ciblant les CPU et pour les implantations des algorithmes décrits dans la suite qui sont disponibles sur demande auprès des auteurs pour évaluation.

2 Découverte des caractéristiques des unités arithmétiques des GPU

Nous avons défini et implanté des algorithmes pour mieux comprendre le fonctionnement de l'addition, de la multiplication et le stockage des nombres à virgule flottante dans les registres et en mémoire. Bien que ces algorithmes peuvent être adaptés à d'autres formats de données, nous avons ciblé le format simple précision sur 32 bits. Les algorithmes ont été écrits dans une version préliminaire en OpenGL et utilisent les *Frame Buffer Object* pour le stockage dans les textures. Ces premiers programmes fonctionnent parfaitement sur la carte Nvidia 7800 GTX avec le driver ForceWare 81.98. En revanche ils ne fonctionnent pas sur la carte ATI RX1800XL et le driver Catalyst 6.3 et nous avons réécrit les programmes en DirectX. Nos algorithmes sont ainsi disponibles pour OpenGL et DirectX.

[3] Ce programme est disponible sur le dépôt Netlib `http://www.netlib.org`.

2.1 Stockage des nombres à virgule flottante dans les registres et en mémoire

Le GPU est souvent utilisé comme coprocesseur. Des données sont générées dans le CPU pour être transférées dans la mémoire de texture avant d'être traitées par le GPU. Pour savoir si des conversions interviennent lors des transferts du CPU vers le 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. Nous avons aussi effectué des opérations sur GPU générant ces valeurs spéciales et récupéré le résultat sur CPU.
Les résultats obtenus montrent que lors d'un transfert sans aucune opération, les nombres dénormalisés sont remplacés par 0 et les infinis ne sont pas modifiés sur Nvidia ou ATI. Les NaN ne sont pas modifiés sur Nvidia mais ATI transforme les sNaN (*signaling NaN*) en qNaN (*quiet NaN*).
Sur certaines architectures, les registres internes stockent les nombres avec une précision supérieure à celle des données en mémoire [18] et avec une dynamique plus grande pour l'exposant [22]. La conversion vers le format final est réalisée lors de l'écriture en mémoire. Nous avons testé si certaines unités des GPU se comportent de manière similaire. Pour tester l'exposant maximum, nous avons utilisé des vecteurs pour calculer

$$(\text{MAX\_FLOAT} + \text{MAX\_FLOAT}) - \text{MAX\_FLOAT}$$

où MAX_FLOAT est le plus grand nombre représentable au format simple précision. Nous avons vérifié le nombre de bits de la mantisse en utilisant des vecteurs pour calculer

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

pour i variant de 1 à 64.
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 implantent un FMA en arithmétique à virgule flottante [22, 14]. 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.
L'exécution des tests montre qu'aucun des registres temporaires des vertex et pixel 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 conserve le produit sur une précision étendue au delà des 24 bits de précision des nombres flottants simple précision.

2.2 La multiplication

Les résultats de l'exécution de *Paranoia* nous laissent penser que la multiplication est tronquée à la fois sur ATI et sur Nvidia. Nous avons testé la façon dont cette troncature était effectuée à l'intérieur des pixel shaders. Pour cela nous avons souhaité déterminer si tous les produits partiels était générés ou si pour une question de rapidité, les bits de poids faibles de certains produits partiels étaient ignorés comme présenté à la figure 4 [30]. 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.
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 $i > 0$

$$y_1 = 1 \qquad y_{2i} = y_{2i-1} + 1 \qquad y_{2i+1} = 4y_{2i-1} + 2.$$

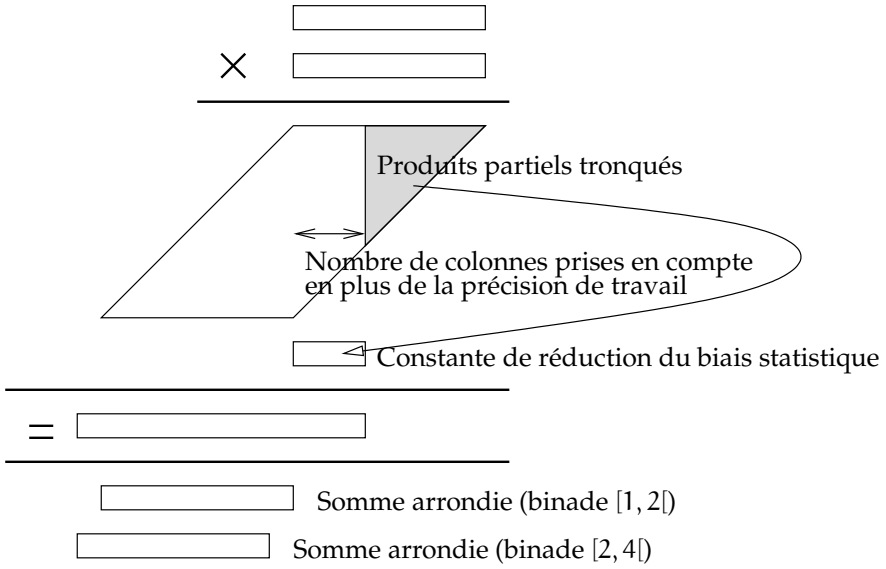

FIG. 4 – Produits partiels tronqués pour accélérer le multiplieur et réduire sa taille

Cette suite peut aussi s'interpréter à l'aide des opérateurs binaires *ou logique* (|) et *décalage vers la gauche* (<<) en :

$$y_1 = 1 \qquad y_{2i} = y_{2i-1} \mid 01_2 \qquad y_{2i+1} = (y_{2i-1} << 2) \mid 10_2.$$

Ce deuxième vecteur alterne une valeur qui crée une chaîne de 1 sans propagation dans les bits de poids faibles 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 attendu.
Les résultats des tests nous indiquent que le multiplieur des pixel shaders traite les produits partiels jusqu'à la 9ème colonne pour ceux d'ATI et jusqu'à la 6ème colonne pour Nvidia. Ces résultats sont confirmés par des tests aléatoires qui montrent qu'un biais est ajouté pour compenser les produits partiels manquants.
Un deuxième test composé de 2 vecteurs de 2^{23} nombres nous donne plus de précision. Il calcule les produits

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

Ce test détermine la valeur de la constante de réduction du biais statistique.
Nous avons testé sur deux milliards de valeurs aléatoires sur ATI et Nvidia que $A \times B$, $(-A) \times (-B)$ et $-(A \times (-B))$ sont égaux, ainsi que $(-A) \times B$ et $A \times (-B)$. 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.

2.3 L'addition

Lors de l'utilisation de l'outil *Paranoia*, nous avons remarqué que la soustraction sur les processeurs Nvidia semblait disposer d'un bit de garde. En l'absence d'informations supplémentaires, nous avons cherché à clarifier le comportement de la soustraction des pixel et vertex shaders de l'ATI et de la Nvidia. Les pixel shaders des deux architectures testées comportent deux MAD cascadés. Nous avons déterminé le comportement du premier additionneur en calculant

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

pour i variant de 1 à 64. Nous avons remarqué que le résultat obtenu était égal à $1,5$ dès que $i \geq 26$ sur ATI et Nvidia dans les vertex et pixel shaders. Ce résultat laisse penser que la première soustraction bénéficie de deux bits de garde.

De façon similaire nous avons testé le deuxième additionneur des pixel shaders en calculant les deux soustractions suivantes

$$(1,5 - 2^{-i}) - 1.5$$

pour i variant de 1 à 64. Le résultat retourné était égal à 0 lorsque $i \geq 25$ sur ATI et $i \geq 26$ sur Nvidia. Les deux additionneurs cascadés des pixel shaders de la Nvidia se comportent donc de façon similaire. En revanche, nous pouvons penser que dans 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 deux bits de garde alors que lorsque deux additions sont lancées, les deux additionneurs sont utilisés et le premier ne dispose que d'un seul bit de garde.
L'utilisation d'un bit de garde permet de ne pas commettre d'erreur d'arrondi quand on calcule la différence de deux nombres proches mais dont les exposants diffèrent d'une unité. En revanche, la présence d'un ou de plusieurs bits de garde supplémentaires dans une arithmétique tronquée peut faire que la propriété suivante n'est plus vérifiée [25, 26]

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

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

$$x + y - x \oplus y$$

est toujours représentable en machine sauf dépassement de capacité. Il faut par la suite modifier certains algorithmes de calcul à précision multiple [27, 16]. Nous continuons nos travaux pour comprendre pourquoi les additionneurs des GPU testés disposent de deux bits de garde et non pas d'un seul.
Nous avons lancé les mêmes additions que précédemment mais en utilisant des flottants sur 16 bits au lieu de 32. Les résultats obtenus sur Nvidia montrent que les additions sont effectuées à précision maximale (26 bits de précision) pour ensuite être arrondies dans le format de destination souhaité (11 bits).

3 Conclusion et perspectives

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 7800 GTX ainsi que 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 additionneurs disposent de 2 bits de garde ce qui explique certaines surprises lors du calcul de l'erreur d'arrondi. Ces résultats constituent une première étape pour la construction d'algorithmes numériques plus précis et plus rapides sur GPU. Cependant, de nombreux tests complémentaires restent à écrire pour caractériser précisément le comportement de ces opérateurs et de ceux que nous n'avons pas encore testés (fonctions trigonométriques, logarithmiques et inverses).

Références

[1] Binary floating-point arithmetic for microprocessor systems. ISO / IEC Standard 60559, International Electrotechnical Commission, 1989.

[2] Sylvie Boldo and Marc Daumas. Properties of two's complement floating point notations. *International Journal on Software Tools for Technology Transfer*, 5(2-3) :237–246, 2004.

[3] 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.

[4] Cem Cebenoyan. Floating point specials on the GPU. Technical report, Nvidia, february 2005.

[5] William J. Cody. MACHAR : a subroutine to dynamically determine machine parameters. *ACM Transactions on Mathematical Software*, 14(4) :303–311, 1988.

[6] William J. Cody. Algorithm 714 : CELEFUNT : a portable test package for complex elementary functions. *ACM Transactions on Mathematical Software*, 19(1) :1–21, 1993.

[7] 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.

[8] William J. Cody, Richard Karpinski, et al. A proposed radix and word-length independent standard for floating point arithmetic. *IEEE Micro*, 4(4) :86–100, 1984.

[9] Jerome T. Coonen. Specification for a proposed standard for floating point arithmetic. Memorandum ERL M78/72, University of California, Berkeley, 1978.

[10] Marc Daumas. *Informatique répartie*, chapter Implantations de la norme IEEE 754 de l'arithmétique à virgule flottante. Hermès, 2005.

[11] Marc Daumas and David W. Matula. Rounding of floating point intervals. *Interval Computations*, (4) :28–45, 1994.

[12] Michael Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12) :1901–1909, 1966.

[13] Michael Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, 21(9) :948–960, 1972.

[14] Freescale Semiconductor. *DSP56800 : Family Manual*, 2005.

[15] 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.

[16] Guillaume Da Graça and David Defour. Implementation of float-float operators on graphics hardware. Technical Report ccsd-00021443, HAL-CCSD, 2006.

[17] Karl Hillesland and Anselmo Lastra. GPU floating-point paranoia. In *ACM Workshop on General Purpose Computing on Graphics Processors*, page C8, August 2004.

[18] Intel. *Pentium II Processor : Developer's Manual*, 1997.

[19] Richard Karpinski. PARANOIA : a floating-point benchmark. *Byte*, 10(2) :223–235, 1985.

[20] Arnaud Legrand and Yves Robert. *Algorithmique parallèle*. Dunod, 2003.

[21] Dinesh Manocha. General purpose computations using graphics processors. *IEEE Computer*, 38(8) :85–88, 2005.

[22] Peter Markstein. *IA-64 and elementary functions : speed and precision*. Prentice Hall, 2000.

[23] 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.

[24] Matt Pharr, editor. *GPUGems 2 : Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley, 2005.

[25] Michèle Pichat. *Contributions à l'etude des erreurs d'arrondi en arithmétique à virgule flottante*. PhD thesis, Université Scientifique et Médicale de Grenoble, Grenoble, France, 1976.

[26] Michèle Pichat and Jean Vignes. *Ingénierie du contrôle de la précision des calculs sur ordinateur*. Editions Technip, 1993.

[27] 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.

[28] John F. Reiser and Donald E. Knuth. Evading the drift in floating point addition. *Information Processing Letters*, 3(3) :84–87, 1975.

[29] N. L. Schryer. A test of computer's floating-point arithmetic unit. Technical report 89, AT&T Bell Laboratories, 1981.

[30] 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.

[31] David Stevenson et al. A proposed standard for binary floating point arithmetic. *IEEE Computer*, 14(3) :51–62, 1981.

[32] David Stevenson et al. An American national standard : IEEE standard for binary floating point arithmetic. *ACM SIGPLAN Notices*, 22(2) :9–25, 1987.

[33] Brigitte Verdonk, Annie Cuyt, and Dennis Verschaeren. A precision and range independent tool for testing floating-point arithmetic I : basic operations, square root and remainder. *ACM Transactions on Mathematical Software*, 27(1) :92–118, 2001.

A Messages générés par Paranoia pour la Nvidia 7800GTX

```
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

----BEGINNING PARANOIA TESTS----

FUZZY TEST
There are two pairs of conditions checked.  One of the two must be true.
The first of the pair is X != 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 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 -> 1.5 test: failed
(1.5+2*U2)(1-U2) = 1.5 + 0.5*U2 -U2^2 -> 1.5 test:  passed
(1.5-2*U2)(1+U2) = 1.5 - 0.5*U2 - 2*U2^2 -> 1.5 - U2 test:  failed
(1.5+U2)(1-U2) = 1.5 -0.5*U2 -U2^2 <= 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 -> up to 1.5+3*U2 so EXACT
(1.5-2*U2)(1-U2) = 1.5 - 3.5*U2 + U2^2 -> Rounded to 1.5-4*U2, so CHOPPED
(1.5 + 2*U2)*(1+U2) = 1.5 + 3.5*U2 + U2^2 -> up to 1.5+4*U2 so EXACT
(1.5-U2)*(1-U2) = 1.5 - 2.5*U2 + U2^2 -> Rounded to 1.5-3*U2, so CHOPPED
(1+2*U2)(1-U2) = 1 + U2 - 2*U2^2 -> Rounded to 1.0+1*U2, so NEITHER
(1+U2)(1-U2) = 1-U2^2 -> 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 <= 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) <= 0 test:  failed
DIVISION rounding: ((1.5+U2+U2)+U2)/(1+U2) <= 1.5+U2 test:  failed
DIVISION rounding: (1.5-U2-U2)/(1-U2) <= 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 operandsup
     at least sometimes (?]

----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 operandsup
     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
```