

## Fonctions élémentaires sur GPU exploitant la localité de valeurs

Sylvain Collange<sup>1</sup>, Marc Dumas<sup>1</sup>, David Defour<sup>1</sup> & Régis Olivés<sup>2</sup>

ELIAUS<sup>1</sup>, UPVD & PROMES<sup>2</sup>, CNRS, UPVD  
52 avenue Paul Alduy — 66860 Perpignan — France  
prenom.nom@univ-perp.fr

---

### Résumé

Les processeurs graphiques sont de puissants coprocesseurs SIMD dédiés au traitement du parallélisme de donnée issue d'application multimédia ou du domaine du calcul généraliste sur processeur graphique (GPGPU). Ces processeurs intègrent différentes unités spécialisées dupliquées de nombreuses fois. Parmi ces unités spécialisées, on retrouve les unités d'évaluation de fonctions de base (inverse, inverse de la racine carrée, exponentielle, fonctions trigonométriques...). L'objectif de cet article est de proposer une modification architecturale destinée à réduire la surface totale occupée par ces unités tout en conservant des performances comparables. Cette modification exploite la localité de valeurs dans un contexte d'exécution SIMD pour partager les tables nécessaires. Deux versions sont proposées et testées : Un partage de table limité à chaque bloc SIMD et un partage global d'une table unique avec des caches partagés au niveau des blocs SIMD. Les tests sont menés à la fois sur les benchmarks classiques qui représentent le travail de développeurs experts et sur des simulations dans le domaine du développement durable qui représentent le travail d'utilisateurs d'outils intégrés. Les traces d'exécution de ces derniers codes sont disponibles auprès des auteurs.

**Mots-clés :** GPGPU, GPU, arithmétique virgule flottante, précision, fonctions élémentaires.

---

### 1. Introduction

Les processeurs graphiques sont aujourd'hui utilisés pour accélérer les applications graphiques ainsi que certaines applications généralistes avec un fort parallélisme de donnée (GPGPU). Pour obtenir de bonnes performances, le parallélisme de donnée est traité par un grand nombre d'unités de calcul qui sont intégrées au sein du processeur graphique. Celles-ci sont organisées selon un schéma SIMD (*single instruction, multiple data*) ou SPMD (*single program, multiple data*). On trouve des unités arithmétiques (principalement chargées des additions, multiplications et comparaisons), des unités de chargement et d'interpolation, des unités de prétraitement des données et enfin des unités d'évaluation des fonctions de base qui ont en charge les fonctions inverse, inverse de la racine carrée, sinus, cosinus, exponentielle et logarithme en base 2. L'évaluation de ces dernières fonctions en matériel sur GPU repose sur l'utilisation combinée ou non d'un cache de données et d'une approximation polynomiale.

Le cache de données tel qu'il est utilisé dans les unités d'évaluation des fonctions de base permet d'accélérer les calculs en mémorisant des résultats déjà calculés. On parle de mémoïzation. Ce principe est exploité dans la *Tree Machine* [13] et il aboutit par exemple à un mécanisme de détection de calculs complexes redondants associé à un cache de résultats [14]. Lipasti *et al.* décrivent un mécanisme de cache de valeurs et de chargement du résultat prédit [8]. Une étude sur les caractéristiques du prédicteur utilisé pour le chargement de résultat prédit est donnée dans [15]. La mémoïzation peut diminuer la latence de l'évaluation en matériel de la division, de l'inverse et de la racine carrée [11]. On peut aussi utiliser le cache uniquement sur les premières étapes de la division en l'adressant avec les bits de poids forts du diviseur [4]. On obtient alors un compromis entre la taille du cache, son taux de réussite et la latence de la division. Si l'on s'autorise des résultats moins précis, on peut mettre en œuvre un « cache flou » qui renvoie un résultat stocké même si l'argument ne correspond pas exactement à l'entrée du cache [3]. Enfin, on peut étendre le jeu d'instructions des processeurs avec des instructions permettant la mémoïzation efficace de fonctions logicielles sans effets de bord comme la racine carrée et les fonctions élémentaires usuelles (exponentielle, logarithme, puissance, fonctions trigonométriques...) [5].

Les processeurs graphiques actuels comptent jusqu'à 64 unités d'évaluation des fonctions de base. Pour accélérer les calculs, chacune de ces unités intègre une table adressée par les bits de poids forts de l'argument. Ces

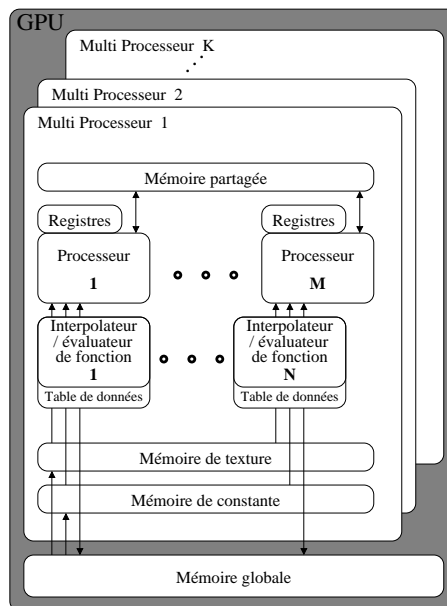


FIG. 1 – Architecture unifiée d'un processeur graphique

tables sont identiques entre chaque unité et contiennent la même information. Il est donc possible de partager ces tables entre toutes les unités. Cependant chaque unité peut accéder à n'importe quelle adresse de cette table. Cette contrainte impose d'avoir une mémoire avec autant de ports que d'unité, ce qui n'est pas envisageable. Nous proposons d'exploiter la localité de valeurs quand les valeurs passées en argument sont très proches sur un groupe de données traitées en SIMD. Cette forme de localité de valeurs permet d'envisager de nouvelles architectures où les tables utilisées pour l'approximation de ces fonctions sont mutualisées sans perte de performance entre les unités d'évaluation des fonctions de base.

Dans ce travail, nous proposons deux modifications de l'architecture des unités d'évaluations des fonctions de base sur GPU pour diminuer leur surface en exploitant la localité de valeurs des arguments. Nous examinons d'une part la possibilité de partager les tables des unités d'évaluations des fonctions de base. Nous étudions d'autre part l'effet de l'ajout de caches locaux. Nous présentons dans la première section l'architecture d'un processeur graphique dans ses grandes lignes. Dans une deuxième section, nous présentons une étude sur l'utilisation faite des fonctions élémentaires dans plusieurs applications graphiques et de calcul scientifique. Enfin nous présentons et discutons les résultats obtenus dans la troisième section.

## 2. Évaluation de fonction sur les processeurs graphiques

Avec l'arrivée des *shaders* programmables, les processeurs graphiques sont passés d'une architecture organisée autour d'un pipeline fixe à une architecture avec une grande variété d'unités spécialisées, présente chacune en grand nombre. Malgré les différences de conception interne, on observe une uniformisation des ressources de calcul disponibles avec l'arrivée de la norme DirectX 10. Les deux principaux constructeurs NVidia et ATI ont défini une structuration des ressources selon un schéma similaire. Ce schéma repose sur un découpage symétrique en blocs SIMD, eux-mêmes subdivisés en unités de traitement (*processing elements* ou *PE*). La mémoire disponible est, elle aussi, divisée en différentes catégories en relation avec le découpage en PE.

### 2.1. Description de l'architecture unifiée

L'architecture unifiée recommandée par la norme DirectX 10 a été mise en place par NVidia à partir des GeForce 8 et par ATI à partir des Radeon HD 2000. Ce type d'architecture est représentée par la figure 1. Le processeur graphique est alors vu comme un ensemble de blocs SIMD. À chaque cycle d'horloge, chaque PE exécute la même instruction mais sur des données différentes. Chaque bloc SIMD exécute une instruction sur plusieurs données. Chaque bloc SIMD intègre différents niveaux de cache dont un ensemble de registres propre à chaque PE, une

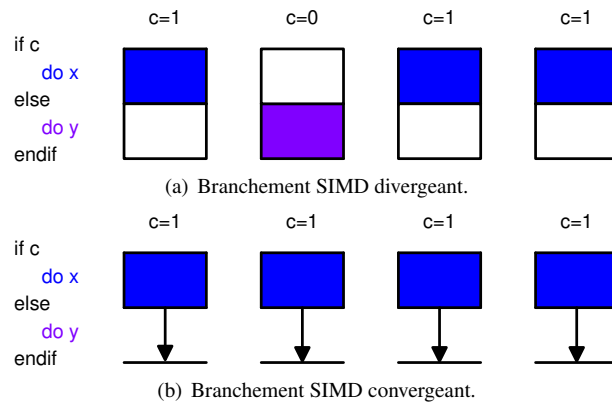


FIG. 2 – Types de branchements en SIMD, avec 4 voies.

mémoire partagée entre tous les PE du bloc SIMD, une mémoire de constantes plus une mémoire de textures accessibles en lecture seule. Enfin chaque PE peut accéder à une mémoire globale en lecture ou écriture.

Les blocs SIMD intègrent différentes unités de calcul (figure 1). Parmi toutes les unités disponibles on trouve des unités de traitement généralistes (PE) contenant des MAD (*Multiplication Addition*), des unités ROP (*Raster Output*), des unités de Z-culling, des unités de texturage/filtrage, des unités d'interpolation et d'évaluation de fonctions de base. Ces unités sont respectivement utilisées pour exécuter les *shaders*, écrire les pixels en mémoire, accéder et filtrer les textures, effectuer des interpolations et évaluer des fonctions de base. Le processeur graphique GeForce 8800 GTX de NVidia possède 16 blocs SIMD et chaque bloc SIMD est composé de 8 PE et de 2 unités d'évaluation de fonctions de base. Sur le Radeon HD 2900 XT de chez ATI, on recense 4 blocs SIMD composés chacun de 16 PE capables d'exécuter 5 instructions différentes simultanément.

Le fonctionnement en mode SIMD impose deux types de contraintes. D'une part, le flot de contrôle doit rester le même pour toutes les voies d'un bloc SIMD. De fait, un branchement dans le code n'est possible que si toutes les voies SIMD suivent le même chemin comme on le voit à la figure 2-b. Faute de quoi, il est nécessaire d'exécuter les deux branches de la condition, en appliquant un masque sur les résultats comme c'est le cas dans la figure 2-a. Ce mécanisme est nommé prédication. Certaines architectures SIMD, dont les GPU, sont capables de déterminer dynamiquement pour chaque branchement s'il peut être suivi par tout le bloc ou s'il faut avoir recours à la prédication. D'autre part, si les unités de calcul sont dupliquées entre toutes les voies SIMD, d'autres types d'unités, et en particulier celles d'accès à la mémoire sont partagées entre ces voies. Cela restreint les possibilités d'accès à la mémoire de la même façon que pour les branchements. Deux schémas principaux d'accès sont couramment acceptés. Le *broadcast* intervient lorsque toutes les voies SIMD accèdent en lecture à la même adresse. Le *coalescing* consiste en un regroupement de lectures à des adresses consécutives. En d'autres termes, la voie 0 accède à l'adresse  $n$ , la voie 1 à l'adresse  $n + 1$ , et ainsi de suite. Par exemple, le GeForce 8 dispose d'une mémoire de constantes partagée accessible en mode *broadcast* uniquement, d'une mémoire globale permettant le *coalescing* avec des contraintes d'alignement, et d'une mémoire partagée permettant ces deux modes ainsi que d'autres combinaisons [10]. Lorsqu'un accès ne respecte pas les schémas autorisés, il est remplacé par plusieurs accès successifs, conduisant à une dégradation des performances.

## 2.2. L'environnement Cuda

Différents environnements de programmations permettent d'accéder aux ressources de calcul des processeurs graphiques. La majorité des programmes graphiques sont écrits dans un langage de programmation graphique comme les langages de *shaders* d'OpenGL et DirectX. Ces programmes sont compilés par le pilote graphique avant d'être exécutés par le processeur graphique. De nouveaux environnements sont apparus avec la progression de l'utilisation des processeurs graphiques pour le calcul généraliste (GPGPU). ATI a par exemple développé un environnement de bas niveau appelé CTM (*Close To the Metal*) pour accéder aux ressources de calcul présentes dans les *shaders* programmables et offre ainsi un contrôle précis sur le matériel. De son côté NVidia propose pour les processeurs de la gamme GeForce 8 un environnement de programmation de haut niveau très proche du langage C appelée Cuda. Ce langage est composé de bibliothèques pour s'abstraire du matériel dont une bibliothèque mathématique four-

Listing 1 – Implantation de  $\sin(x)$  dans Cuda, version précise

```

0 float sinf(float x)
1 // L'algorithme a été réécrit pour ne pas violer les droits d'auteurs attachés à Cuda
2 {
3     int k;
4     float xr;
5     float r;
6
7     if(isinf(x) {
8         return nanf(NULL);
9     }
10    if(x == 0) {
11        return x; // conserver le signe du zéro
12    }
13
14    // Réduction d'argument par Cody & Waite avec 3 constantes [9, pp. 177-178]
15    xr = range_reduction_sincos(x, &k);
16    // Argument réduit  $xr = x - k(\pi/2)$ ,  $-\pi/4 < xr < \pi/4$ 
17
18    if(k & 1) // quadrant  $[\pi/4, 3\pi/4]$  ou  $[5\pi/4, 7\pi/4]$ 
19    {
20        // Approximation de  $\cos(xr)$  sur  $[-\pi/4, \pi/4]$  par un polynôme pair
21        r = cos_reduced(xr);
22    }
23    else // quadrant  $[-\pi/4, \pi/4]$  ou  $[3\pi/4, 5\pi/4]$ 
24    {
25        // Approximation de  $\sin(xr)$  sur  $[-\pi/4, \pi/4]$  par un polynôme impair
26        r = sin_reduced(xr);
27    }
28
29    if(k & 2) { //  $[3\pi/4, 7\pi/4[$ 
30        r = -r;
31    }
32    return r;
33 }

```

nissant les fonctions sinus, cosinus, exponentielle et autres, plus précises que les évaluations matérielles existantes. Nous présentons la fonction `sinf` dans le listing 1.

### 2.3. L'unité d'évaluations des fonctions de base

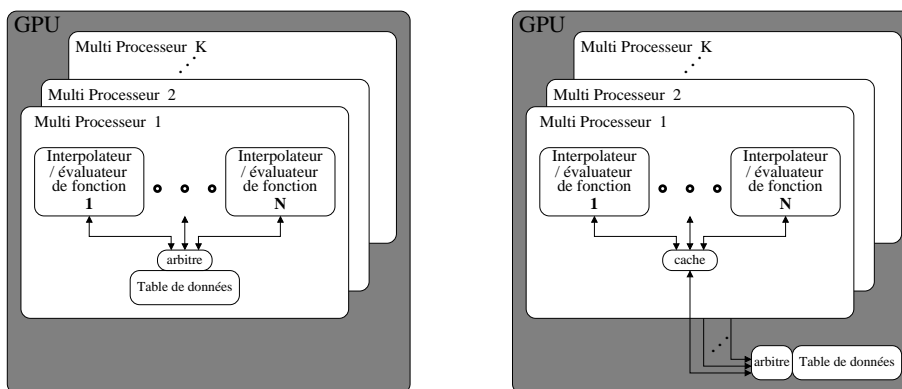
Depuis l'apparition des *shaders* programmables, les processeurs graphiques disposent de plusieurs unités capables d'évaluer les fonctions de base les plus utilisées en graphisme. Parmi ces fonctions, on retrouve l'évaluation en matériel en simple précision de l'inverse, de l'inverse de la racine carrée, du cosinus, du sinus, du logarithme et de l'exponentielle en base 2.

Oberman propose dans [12] de fusionner l'unité d'interpolation et l'unité d'évaluation des fonctions de base pour réduire l'emprunte matérielle tout en conservant une latence raisonnable. L'unité d'interpolation décrite dans l'article d'Oberman est capable d'évaluer la fonction  $U(x, y) = A \times x + B \times y + C$ . Pour évaluer les fonctions de base il propose d'utiliser l'interpolateur précédé d'un peu de matériel, pour évaluer des approximations polynomiales de degré 2. Les coefficients du polynôme dépendent de la fonction évaluée ainsi que de l'argument. Les 22,75 Kb de cette table représente 18% de la surface totale de l'ensemble interpolation et évaluation des fonctions de base ou encore 91% des circuits à ajouter pour faire en sorte que l'unité d'interpolation soit capable d'évaluer ces fonctions de base.

Une unité similaire est décrite dans un brevet d'ATI [7]. Elle repose sur un circuit *ad hoc* d'évaluation de polynômes et sur une table à 32 entrées par fonction. Chaque entrée contient un polynôme d'approximation de degré 3 dont les coefficients sont codés sur 72 bits au total.

## 3. Architectures alternatives envisagées

Chaque bloc SIMD d'un processeur graphique embarque plusieurs unités d'évaluation de fonctions de base. Les tables utilisées dans ces unités peuvent représenter 18% de la surface. Nous proposons d'exploiter la localité de valeurs des arguments en entrée pour diminuer la surface totale prise par ces tables tout en conservant de



Solution 1 : Table partagée intra-multiprocesseur      Solution 2 : Table partagée inter-multiprocesseur

FIG. 3 – Proposition de partage des tables entre les unités d'évaluation des fonctions élémentaires

bonnes performances. Le type d'architecture résultante mutualise les données mémorisées entre toutes les unités d'évaluation des fonctions de base. Nous envisageons deux solutions présentées dans la figure 3. La première solution utilise une table unique par bloc SIMD. La seconde solution propose un accès hiérarchique avec un cache par bloc SIMD et une table unique pour tout le processeur graphique.

### 3.1. Tables partagées au niveau de chaque bloc SIMD

Notre première proposition consiste à partager une table entre plusieurs unités d'évaluation d'un même bloc SIMD comme on le voit à la figure 3-1. On n'a plus une table par unité de calcul mais une seule table sur tout le bloc SIMD. Les accès à ces tables sont gérés par un arbitre. Ce circuit fait en sorte que le résultat est transmis à toutes les unités d'évaluation en un cycle si tous les accès se font à la même adresse. Sinon, les accès sont sérialisés. La lecture de la table nécessite alors autant de cycles qu'il y a d'adresses différentes dans les requêtes.

Ce mode de fonctionnement est actuellement mis en œuvre dans les mémoires de constantes et les mémoires partagées du GeForce 8 de NVidia. De plus, le partage de matériel entre ces unités ne réclamerait pas de ressources de routage excessives dans la mesure où les différentes unités d'évaluation sont déjà regroupées en blocs SIMD. La réduction dans le nombre de répliqués des tables partagées pourrait représenter un bon compromis entre la diminution du coût en mémoire et la perte d'efficacité engendrée par le partage. Le prix de ce partage est fonction du nombre de ports de la mémoire à utiliser ainsi que l'ajout d'un système d'arbitrage. Nous verrons par la suite qu'une seule mémoire à un port est suffisante pour beaucoup d'applications. Cela reste vrai dans le cas d'un partage entre un grand nombre d'unités.

### 3.2. Table unique et caches partagés au niveau de chaque bloc SIMD

La deuxième architecture proposée consiste à pousser le raisonnement précédent au niveau de l'ensemble des blocs SIMD en hiérarchisant les accès à la table de données. Dans cette solution, une table unique est partagée entre tous les blocs SIMD et chaque bloc SIMD dispose de son propre cache local contenant un sous-ensemble des entrées de la table partagée. On obtient ainsi une hiérarchie mémoire spécifique pour les constantes de la table. Bien que l'objectif initial des systèmes de caches soit une diminution de la latence des accès (qui représente une ressource peu critique dans le cas des GPU), ils permettent également une augmentation de la bande passante cumulée tout en diminuant les contraintes sur le débit de la mémoire partagée.

## 4. Applications étudiées

### 4.1. Benchmarks classiques représentant le travail de développeurs experts

L'utilisation principale d'un GPU est avant tout graphique et nous avons eu recours aux applications du benchmark SPECviewperf 8.1 [2]. Ce benchmark est construit à partir d'applications de conception assistée par ordinateur (Catia, Pro/ENGINEER, SolidWorks, TCVis, Unigraphics), d'infographie (3D Studio Max, Lightscape, Maya) et de visualisation (EnSight) utilisées dans l'industrie. Bien que ces applications aient des fonctionnalités allant

bien au-delà de l'affichage d'images de synthèse en temps réel, seule la partie visualisation est considérée dans le benchmark SPECviewperf. L'objectif est de tester spécifiquement le sous-système graphique (GPU, mémoire et bus graphiques). Ainsi, la majeure partie du calcul effectué par ces applications est exécutée sur le GPU. Les applications graphiques incluses dans SPECviewperf réalisent des appels très fréquents à la fonction racine carrée pour les calculs géométriques et à la fonction puissance pour les calculs d'éclairage. La fonction puissance est souvent décomposée en une exponentielle suivi d'un logarithme en base 2.

Le GPU est de plus en plus sollicité pour accélérer l'exécution d'application de calcul généraliste avec un important parallélisme de donnée (GPGPU). Nous avons sélectionné Spacetrack [16], une application combinant parallélisme de donnée et appels aux fonctions de base. Spacetrack est un code utilisé par le département de la défense américaine pour la simulation de trajectoires de satellites. Ce code utilise intensément les fonctions trigonométriques.

#### **4.2. Simulations représentant le travail d'utilisateurs d'outils intégrés**

Nous distinguons les applications précédentes de celles que nous présentons maintenant. Nous venons d'examiner des applications mises en place par des experts où tout est fait pour éviter des recours inutiles aux fonctions élémentaires coûteuses. Nous nous intéressons maintenant à des applications de simulations dans des environnements de travail capables de répondre à des demandes variées en matière de simulation. Ces derniers logiciels privilégient la polyvalence parfois au détriment des performances. Ils représentent toutefois de gros consommateurs en matière de calcul intensif et certains d'entre eux sont bien adaptés à une implantation sur GPU.

GPU4RE [6] est un programme de simulation des transferts radiatifs d'un récepteur solaire à gaz pressurisés. Les différentes raies d'émission des gaz considérés créent un énorme parallélisme de données et les lois physiques nécessitent un recours abondant à la fonction exponentielle.

Comsol [1] est un environnement de simulation commercial pouvant être utilisé à toutes les étapes d'un processus de modélisation. Nous avons sélectionné une étude de stockage de l'énergie par changement de phase. L'essor des énergies renouvelables et, en particulier, de l'énergie solaire est tributaire du caractère intermittent et aléatoire de la ressource. Ainsi, l'alternance jour-nuit, les passages nuageux, les journées ensoleillées ou non... exigent de mettre en place des moyens de stockage et de gestion de l'énergie.

Le changement d'état liquide-solide d'un matériau pur est caractérisé par la transformation, à température constante, d'une phase liquide en une phase solide ou inversement. Ce phénomène réversible s'accompagne d'une consommation (fusion) ou d'une restitution (solidification) d'énergie, modélisé par des fonctions gaussiennes : la chaleur latente de fusion-solidification. Les problèmes de transfert thermique avec changement d'état solide-liquide ont une importance considérable dans de nombreuses applications techniques ou processus naturels. Dans le domaine de la gestion de l'énergie, le changement d'état solide-liquide se révèle un moyen particulièrement intéressant pour le stockage de chaleur (ou de froid). La simulation doit permettre la validation de réservoir de stockage et l'optimisation de configurations géométriques.

Les traces d'exécution des applications GPU4RE et Comsol sont disponibles auprès des auteurs. Elles constituent les premiers éléments d'un ensemble de benchmarks liés à l'utilisation des fonctions élémentaires.

### **5. Simulations et résultats**

#### **5.1. Instrumentalisation du code et outils d'analyse des traces**

Pour quantifier l'intérêt des deux architectures proposées, nous avons simulé l'exécution des applications présentées en section 4 sur un processeur graphique. Pour cela nous avons instrumenté les appels aux fonctions racine carrée, inverse de la racine carrée, exponentielles, logarithmes, sinus et cosinus pour générer des traces d'exécution. Les autres fonctions, comme la fonction puissance utilisée dans le calcul de l'éclairage, sont des combinaisons des fonctions présentes en matérielles.

L'instrumentation de ces applications a été réalisée différemment selon leurs langages de programmations. Les processeurs graphiques n'ont pas les mêmes entrées-sorties qu'un processeur généraliste. Il est donc très difficile d'instrumenter du code s'exécutant sur un GPU. Pour les applications graphiques, nous avons modifié et instrumenté la bibliothèque graphique Mesa 3D version 7.0.1 ([www.mesa.org](http://www.mesa.org)), une implantation open source de la norme OpenGL. Cette bibliothèque a rendu possible l'exécution des codes sur CPU et la génération des traces sur la part du rendu graphique habituellement effectuée par le GPU. Pour les applications de type GPGPU, nous avons instrumenté directement le code s'exécutant sur le CPU pour simuler son exécution sur GPU. Pour les applications sur Comsol, nous avons utilisé des fonctions Maple externes.

Pour la simulation de la première architecture, correspondant à l'utilisation de la mémoire de constante pour le

TAB. 1 – Conditions d’invocation des fonctions de base

Application	Fonction	Appel (module : ligne)	# d’appels SIMD	# d’adresses par requête
3DSMax	inv_sqrtf	math/m_norm_tmp.h:71	818866	1.00066
	inv_sqrtf	tnl/t_vb_lighttmp.h:386	846128	1.17539
	sqrtf	tnl/t_vb_lighttmp.h:315	1637732	1.94516
Catia	inv_sqrtf	main/light.c:1111	1208	1
	inv_sqrtf	main/light.c:1116	1208	1.00083
	inv_sqrtf	math/m_norm_tmp.h:71	2383	1
Maya	sqrtf	main/light.c:1172	1001	1.40559
Pro/ENGINEER	exp2	main/light.c:961	6120	14.0861
	inv_sqrtf	main/light.c:1111	1644	1
	inv_sqrtf	main/light.c:1116	1644	1
	log2	main/light.c:961	6120	11.8275
	sqrtf	main/light.c:1172	3451	1.02637
TCVis	inv_sqrtf	main/light.c:1111	32194	1
	inv_sqrtf	main/light.c:1116	32194	3
	inv_sqrtf	main/light.c:1170	3151	1
Unigraphics	sqrtf	swrast/s_aalinetemp.h:137	741140	12.8904
GPU4RE	exp	soft_comp.cpp:170	6229	5.10901
	exp	soft_comp.cpp:171	6229	1.00209
Comsol	exp		516652	1.86562

stockage des tables, nous avons considéré un cas similaire à une architecture réelle de type NVidia GeForce 8. Les tables considérées disposent de 64 entrées et sont partagées entre 16 voies SIMD.

Pour la simulation de la deuxième architecture avec une seule table pour tous les blocs SIMD et un cache partagé par bloc SIMD, nous nous sommes basés sur une table à 64 entrées par fonction similaire à celle décrite dans [12]. Nous avons ensuite testé différents caches de taille 1 à 32 avec des associativités de 1 à 8, adressés par les bits de poids fort de l’argument réduit. Nous avons considéré une politique de remplacement LRU, avec des lignes de caches ne contenant qu’une valeur.

## 5.2. Résultats pour des tables partagées au niveau de chaque bloc SIMD

Nous cherchons à savoir s’il est possible de mutualiser la table des données de l’évaluateur de fonctions de base et nous voulons déterminer le nombre de ports nécessaires. La table 1 présente le nombre d’appels SIMD et le nombre moyen d’adresses par requête à la table observé au cours de ces appels SIMD. La première colonne de résultat donne le nombre d’appel SIMD correspondant à 16 évaluations effectuées en parallèle. La seconde colonne de résultat donne le nombre d’adresses différentes par bloc de 16 requêtes SIMD. Il peut varier de 1 (parallélisme maximal) à 16 (sérialisation complète). Ces résultats sont donnés en fonction du module et numéro de ligne de l’appel. Nous n’avons pas considéré les fonctions appelées moins de 1 000 fois au cours d’une exécution. Cela nous a conduits à exclure le programme Spacetrack des résultats de cette section car aucun appel de fonction élémentaire dans cet exemple ne dépasse ce seuil.

Une analyse plus fine du code montre que :

- Le protocole de test actuel n’est pas adapté pour l’étude de l’appel à `inv_sqrtf` dans `main/light.c:1116` du benchmark TCVis. Le nombre d’adresses par requête est de 1.
- Pour les lignes du tableau où le nombre d’accès est exactement égal à 1, le code calcule toujours la même valeur. Richardson parle dans ce cas de calcul trivial [14]. Ce calcul manifestement inutile est dû au fait que le programme appelle des routines normalisée par OpenGL pour un modèle d’éclairage général alors que le logiciel étudie un cas particulier.
- L’appel à la fonction `pow` dans `main/light.c:961` dans Pro/ENGINEER, est décomposé en  $\log_2$  et  $\exp_2$  et crée très peu de localité. En fait, l’application remplit une table pour la fonction `pow` destinée à accélérer les

TAB. 2 – Taux de convergence des branchements dans les fonctions sin et cos, par appel de fonction

Application	Appel (module :ligne(fonction))	Convergenents	Divergenents	Taux de convergence
Spacetrack	SGP4UNIT.CPP :1872(sin)	52	84	0.382353
	SGP4UNIT.CPP :1873(cos)	52	84	0.382353

calculs d'éclairage qui ne nécessite pas une grande précision. Dans ce benchmark, le remplissage de la table nécessite plus d'appels à la fonction pow qu'il n'aurait été nécessaires si on avait évalué cette fonction pour chaque sommet. Les opérateurs matériels d'évaluation des fonctions  $\log_2$  et  $\exp_2$  du GPU rendent ces calculs moins coûteux que des accès à la mémoire. On peut donc supposer que cette méthode est utilisée pour le rendu sur CPU et ne l'aurait pas été pour une exécution sur GPU.

Sur l'ensemble des programmes testés, on observe qu'en moyenne une requête ne concerne que 3.3 adresses différentes pour des tables composées de 64 entrées. Si nous éliminons les cas que nous venons d'aborder cette moyenne retombe à 1.9. Ainsi ces travaux montrent qu'il nous faut maintenant étudier le cas d'une mémoire double port par bloc SIMD pour mémoriser la table nécessaire à l'évaluation des fonctions de base au lieu d'une mémoire par unité d'évaluation.

### 5.3. Résultats pour une table unique et des caches partagés au niveau de chaque bloc SIMD

Les figures 4, 5 et 6 présentent les taux de réussite des caches présent dans les blocs SIMD sur les applications considérées, en fonction de la taille et de l'associativité du cache. Le contexte dans lequel ces tests ont été réalisés correspond à un démarrage à froid des caches.

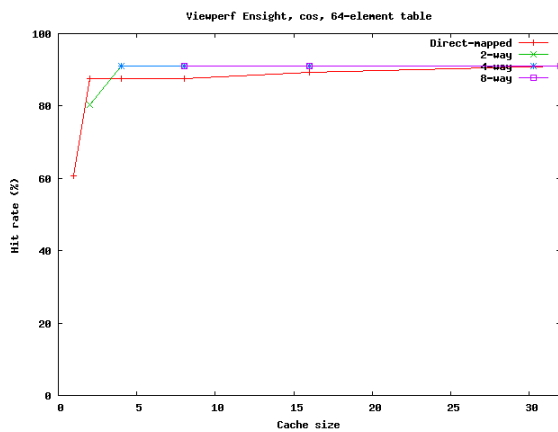
Le concept de localité dans un contexte d'exécution SIMD est difficile à définir et à modéliser. Néanmoins on remarque que l'inconvénient majeur des méthodes à base de cache est qu'à l'inverse de la problématique du partage des tables entre unités SIMD, différents appels non-corrélés se retrouvent en compétition dans le cache, menant à des conflits fréquents. Ces appels non-corrélés correspondent à des instructions différentes où à des instructions traitant des données avec une très faible localité spatiale et donc une faible localité de valeur dans un contexte de donnée graphique (des pixels voisins ont souvent les mêmes attributs). Ce phénomène rend les associativités les plus faibles peu efficaces. On remarque également que les performances augmentent beaucoup lorsque l'on passe d'un cache de 2 éléments à un cache à 4 éléments pour ensuite plafonner à partir de 8. Les taux de succès des caches ne permettent donc pas d'envisager un partage hiérarchique efficace de la table entre les blocs SIMD. Néanmoins, le démarrage à froid des caches impacte certainement les performances des caches et d'autres tests intégrant un modèle plus réaliste avec des caches déjà remplis restent à réaliser.

### 5.4. Résultat sur les branchements

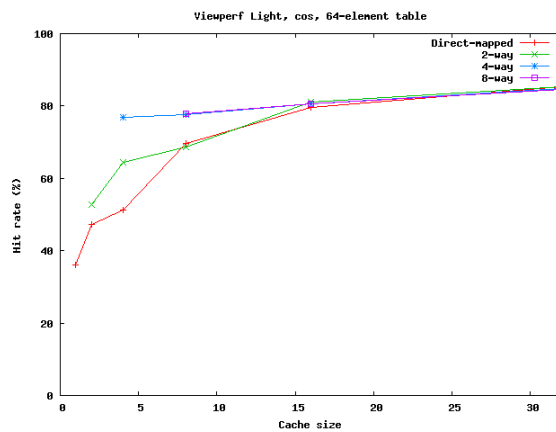
Il est également possible de tirer parti de la localité de valeurs avec des méthodes d'évaluation logicielles sur les GPU actuels. Revenons par exemple sur l'évaluation de la fonction sinus donné dans le listing 1, qui correspond à l'algorithme d'évaluation du sinus précis dans la bibliothèque mathématique de Cuda. On observe qu'en dehors du traitement des situations exceptionnelles, ce code contient un branchement choisissant le polynôme d'approximation à évaluer en fonction du quadrant trigonométrique dans lequel se trouve l'argument. Si tous les arguments des sinus évalués en SIMD se trouvent dans le même quadrant, un seul polynôme sera évalué par le bloc SIMD. À l'inverse, si les quadrants diffèrent, il sera nécessaire d'évaluer le polynôme pour la fonction sinus puis le polynôme pour la fonction cosinus successivement. Il faudra ensuite recourir à la prédication pour ne conserver qu'un seul des résultats. La version 1.0 du compilateur Cuda élimine totalement ce branchement de la fonction sinus pour le remplacer par des instructions prédiquées. Le compilateur considère en effet que la probabilité d'un branchement convergent est trop faible pour que le gain dépasse le surcoût du branchement.

Utiliser la mémoire de constantes du GeForce 8 pour stocker des tables dans le cadre d'un processus d'évaluation logiciel représenterait un algorithme analogue à celui que nous proposons de mettre en œuvre en matériel. En effet, la mémoire de constante partageant un port de lecture unique entre toutes les voies d'un bloc SIMD.

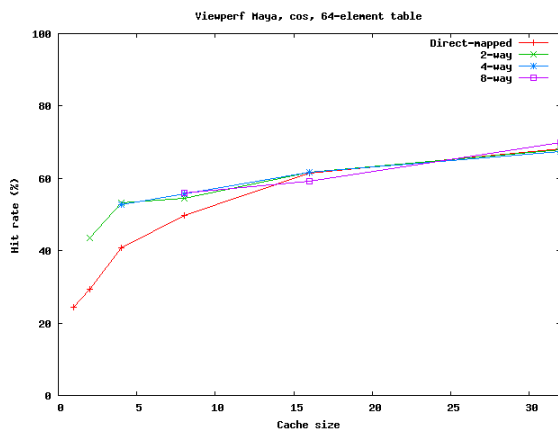
Nous avons simulé le fonctionnement d'une implantation des fonctions sinus et cosinus telle que celle de Cuda 1.0 (listing 1) sur une architecture SIMD à 16 voies (correspondant au NVidia GeForce 8). Pour les tests sur les architectures SIMD, nous regroupons les appels successifs à une fonction donnée en groupe de 16. Pour cela, nous faisons l'hypothèse que ces 16 appels sont indépendants et peuvent être parallélisés. Nous devons également



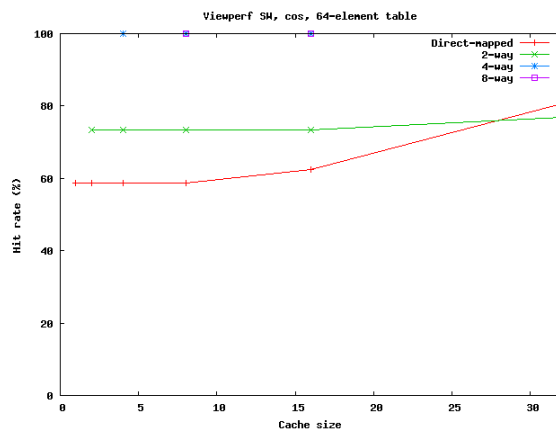
(a) Viewperf Enight.



(b) Viewperf Lightscape.

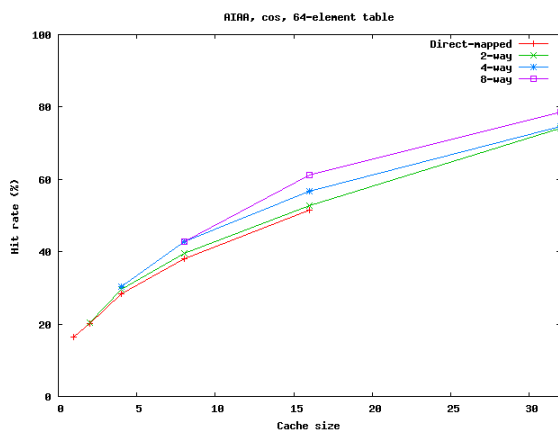


(c) Viewperf Maya.

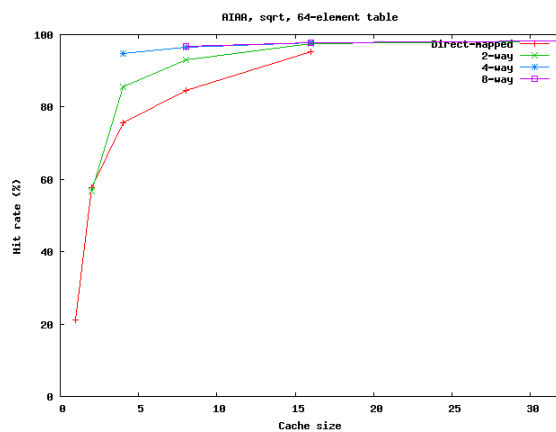


(d) Viewperf SolidWorks.

FIG. 4 – Simulation d'un cache sur les appels à la fonction cos dans les SPEC Viewperf.



(a) Spacetrack, cos.



(b) Spacetrack, sqrt.

FIG. 5 – Simulation d'un cache sur les appels aux fonctions cos et sqrt dans Spacetrack.

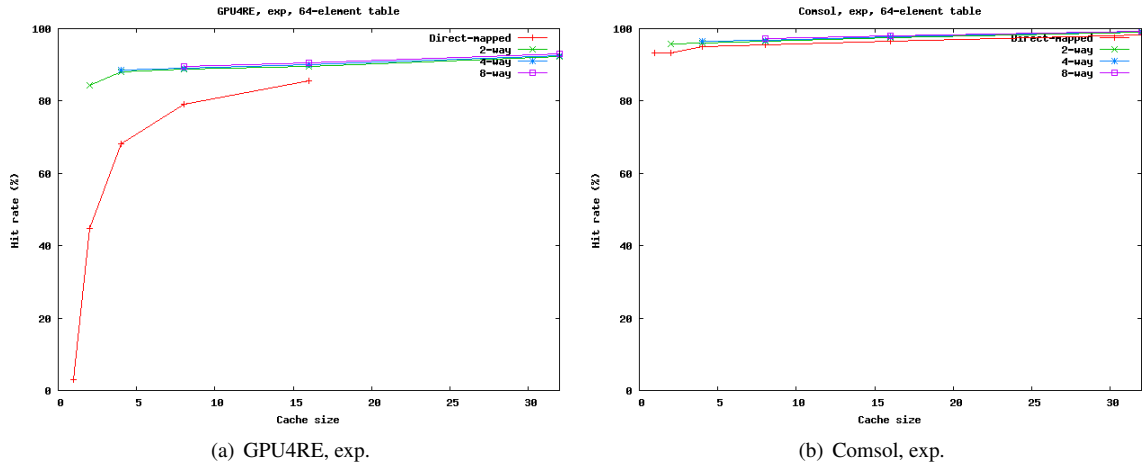


FIG. 6 – Simulation d'un cache sur les appels à la fonction exp dans GPU4RE et Cmsol.

distinguer les différents appels statiques, ce que nous faisons en retenant le nom du fichier source et le numéro de ligne de l'appel. Nous étudions le taux de convergence moyen des branchements en fonction du quadrant dans lequel se trouve l'argument réduit. Nous avons uniquement considéré les appels de fonctions exécutés plus de 100 fois lors des tests.

Les SPEC Viewperf ne faisant pas appel aux fonctions trigonométriques en nombre suffisant, seul Spacetrack est inclu dans ces résultats. La table 2 présente le nombre d'appels convergents et divergents pour chacun de ces appels SIMD. Un appel SIMD représente 16 évaluations simultanées indépendantes de la fonction considérée. En étudiant le code, on remarque qu'il fait appel à des sinus et cosinus avec exactement les mêmes arguments. Le branchement dans l'évaluation de la fonction cosinus prendra le chemin opposé de celui de la fonction sinus évaluée précédemment avec le même argument. Il serait donc possible d'éliminer totalement le branchement tout en évitant de recalculer la réduction d'argument en substituant les appels individuels à sin et cos par un seul appel à la fonction sincos. La lecture du tableau nous permet d'observer que le taux de convergence des appels aux fonctions sinus et cosinus dans cet exemple reste faible, rendant les méthodes d'évaluation utilisant des branchements peu efficaces. Les données dont nous disposons sont encore insuffisantes pour pouvoir dégager une tendance générale.

## 6. Conclusion et perspectives

Sur les processeurs graphiques, les unités d'évaluations de fonctions de base (sinus, cosinus, inverse, racine carrée inverse, exponentielle et logarithme en base 2) utilisent des données tabulées combinées à un évaluateur polynomial. Les tables utilisées pour stocker ces données peuvent représenter 20% de la surface totale de l'opérateur. Nous avons présenté deux modifications architecturales de l'unité d'évaluation des fonctions élémentaires dans un contexte d'architecture SIMD. Ces modifications reposent sur la mutualisation des données entre toutes les unités disponibles en matériels.

La première modification correspond à utiliser une seule table par bloc SIMD pouvant compter jusqu'à 16 unités d'évaluation. Les tests réalisés sur les traces d'exécutions de programme issu d'applications graphique ou du domaine du GPGPU montre qu'en moyenne 2 ports d'accès à cette unique table suffisent. La perte de performance lié à la sérialisation des accès aux adresses différentes devrait être imperceptible pour la majorité des applications testées.

La deuxième modification correspond à étendre le concept précédent à une unique table pour toutes les unités d'évaluation de fonctions de base présentes dans le GPU. Pour cela des caches de différentes tailles sont ajoutés dans les blocs SIMD permettant d'accéder à une unique table. Les faibles taux de succès de ces caches démontrent que les conflits générés par des accès non-corrélés dans le cadre d'une exécution SIMD ne permettent pas de mutualiser efficacement les tables entre les blocs SIMD.

Il reste cependant à quantifier les gains apportés par cette architecture dans des applications réelles aussi bien que son surcoût. Nous nous proposons également de simuler différents compromis entre le nombre d'unités d'interpo-

lation et le nombre de ports de la mémoire partagée entre ces unités dans des travaux futurs. Nous devons aussi tenir compte de la tendance actuelle suivie par les GPU qui intègre de plus en plus d'unités de calcul et simuler le gain obtenu lors de l'augmentation du nombre d'unités.

## Bibliographie

1. Comsol Multiphysics modeling environment. <http://www.comsol.fr>.
2. SPECviewperf 8.1 : Standard Performance Evaluation Corporation. <http://www.spec.org/gwpg/gpc.static/vp81info.html>.
3. Carlos Álvarez, Jesús Corbal, and Mateo Valero. Fuzzy memoization for floating-point multimedia applications. *IEEE Trans. Comput.*, 54(7) :922–927, 2005.
4. Edward Benowitz, Miloš Ercegovac, and Farzan Fallah. Reducing the latency of division operations with partial caching. *Signals, Systems and Computers, 2002. Conference Record of the Thirty-Sixth Asilomar Conference on*, 2 :1598–1602 vol.2, 3-6 Nov. 2002.
5. Daniel Citron and Dror G. Feitelson. Hardware memoization of mathematical and trigonometric functions, 2000.
6. Sylvain Collange, Marc Daumas, and David Defour. Line-by-line spectroscopic simulations on graphics processing units. *Computer Physics Communications*, 2007.
7. ATI Technologies Inc Daniel B. Clifton. Method and system for approximating sine and cosine functions. US Patent 6 976 043, US Patent Office, 2001.
8. Mikko H. Lipasti, Christopher B. Wilkerson, and John Paul Shen. Value locality and load value prediction. *SIGOPS Oper. Syst. Rev.*, 30(5) :138–147, 1996.
9. Jean-Michel Muller. *Elementary functions, algorithms and implementation*. Birkhauser, 2006.
10. nVidia. *NVIDIA CUDA Compute Unified Device Architecture Programming Guide, Version 1.0*, 2007.
11. Stuart F. Oberman and Michael J. Flynn. On division and reciprocal caches. Technical Report CSL-TR-95-666, 1995.
12. Stuart F. Oberman and Michael Siu. A high-performance area-efficient multifunction interpolator. In Koren and Kornerup, editors, *Proceedings of the 17th IEEE Symposium on Computer Arithmetic (Cap Cod, USA)*, pages 272–279, Los Alamitos, CA, July 2005. IEEE Computer Society Press.
13. Samuel Pollock Harbison III. *A computer architecture for the dynamic optimization of high-level language programs*. PhD thesis, Pittsburgh, PA, USA, 1980.
14. Stephen E. Richardson. Exploiting trivial and redundant computation. In E. E. Swartzlander, M. J. Irwin, and J. Jullien, editors, *Proceedings of the 11th IEEE Symposium on Computer Arithmetic*, pages 220–227, Windsor, Canada, June 1993. IEEE Computer Society Press, Los Alamitos, CA.
15. Carmen Turinici, Christine Rochange, and Pascal Sainrat. Prédicteurs mixtes pour l'anticipation des instructions. In *5e Symposium sur les Architectures Nouvelles de Machines (SYMPA'5)*, Rennes, 08/06/99-11/06/99, pages 165–174. INRIA, juin 1999.
16. David A. Vallado, Paul Crawford, Richard Hujsak, and T.S. Kelso. Revisiting spacetrack report #3. In *Proceedings of the AIAA/AAS Astrodynamics Specialist Conference, Keystone, CO*, August 2006.