550, rue Sherbrooke Ouest, bureau 100
Montréal (Québec) H3A 1B9
Tél. : 514 840-1234; Téléc. : 514 840-1244
Place de la Cité – Tour de la Cité
2600, boul. Laurier, bureau 625
Québec (Québec) G1V 2W1
Tél. : 418 648-8080; téléc. : 418 648-8141
http://www.crim.ca

Rapport technique

Calculs généraux sur processeurs graphiques (GPGPU)

Première version

CRIM-07/10-19

Pascal Jetté
Stagiaire
Marc Lalonde
Conseiller
Équipe Vision et imagerie

25 octobre 2007

Collection scientifique et technique

ISBN-13 : 978-2-89522-104-3
ISBN-10 : 2-89522-104-9

Pour tout renseignement, communiquer avec:
CRIM Centre de documentation
CRIM
550, rue Sherbrooke Ouest, bureau 100
Montréal (Québec) H3A 1B9

Téléphone : (514) 840-1234
Télécopieur : (514) 840-1244

ISBN-13 : 978-2-89522-104-3
ISBN-10 : 2-89522-104-9
Dépôt légal - Bibliothèque et Archives nationales du Québec, 2007
Dépôt légal - Bibliothèque et Archives Canada, 2007

TABLE DES MATIÈRES

1. INTRODUCTION

La vitesse, la bande passante et le faible coût des GPU (***G***raphics ***P***rocessing ***U***nit) sur le marché en font une solution attrayante pour des applications nécessitant des calculs complexes. De par son architecture hautement parallèle, le GPU est effectivement capable de résoudre des calculs complexes beaucoup plus rapidement.

Le graphique suivant nous montre l'évolution de la rapidité des différentes architectures. Nous pouvons y voir qu'en 2006, les CPU *dual-core* affichent un maximum d'environ 30 GFlops (milliards d'opérations de point flottant par seconde) alors que les cartes graphiques d'ATI et de NVIDIA affichent un rendement semblable de 200 GFlops. Il est cependant à noter que les valeurs affichées sont dites de pointe (*peak*), soit les valeurs de performance maximale atteinte par l'architecture, et non la performance soutenue (*sustained performance)*.

[1]

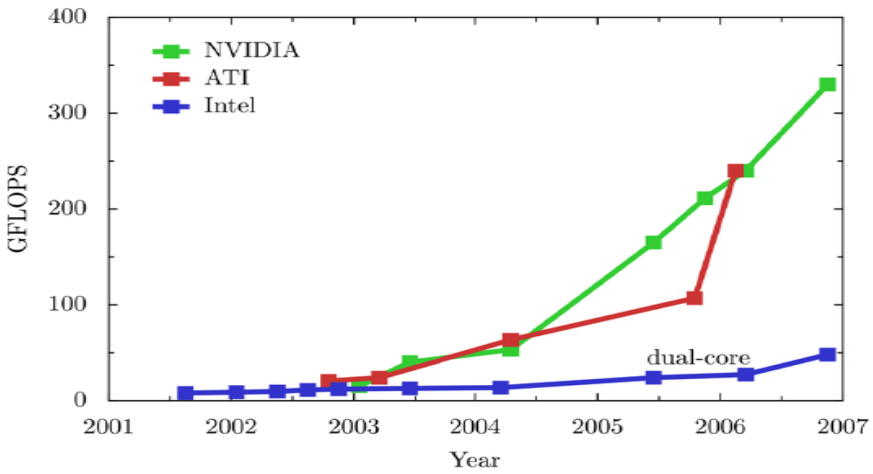


Figure 1 - Rendement CPU vs. GPU

L'objectif principal du présent document est de trouver et de présenter une librairie existante qui pourra permettre aux programmeurs d'utiliser le GPU sans avoir à apprendre un tout nouveau langage et devoir se familiariser avec les astuces et trucs de compilation propres au GPGPU. De plus, la librairie devra être portable de plateforme en plateforme (Windows ou Linux) et de fournisseur en fournisseur de GPU (ATI, NVIDIA). Les quatre critères de sélection sont donc, en ordre d'importance : coût, performance, facilité

[1] LUEBKE, David, *General-Purpose Computation on Graphics Hardware*, Supercomputing 2006 [conf]

d'utilisation, portabilité. Pour ce faire, nous survolerons donc les détails de l'architecture de GPU, évaluerons les différentes façons de s'en servir et les obstacles à son utilisation et nous évaluerons finalement les performances de différents algorithmes implémentés sur GPU.

Il est primordial de noter que pour la plupart des résultats présentés, plusieurs données sont manquantes. Par exemple, lorsque l'on présente les performances d'un algorithme sur GPU, souvent, les temps de transferts de mémoire sont omis et, puisque ces temps de transfert sont souvent ce qui est le plus coûteux sur GPU, la performance est grossièrement surestimée. Par conséquent, il est important de prendre les évaluations de performance d'un produit (surtout par la compagnie qui en fait la promotion) avec un grain de sel.

2. ARCHITECTURE DU GPU

Nous expliquerons ici les détails du GPU selon la façon dont il est vu par les APIs graphiques existants. La pipeline graphique (DirectX9, OpenGL) avec ses *shaders* programmables est le modèle le plus couramment utilisé et nécessite par conséquent une explication approfondie. Certains langages spécialisés (CTM d'ATI/AMD et CUDA de NVIDIA) ne respectent pas ce modèle et agissent directement sur la carte graphique au niveau des données et des instructions.

2.1 Textures et rendu

Les structures de données natives d'un GPU sont les textures. Dans les applications graphiques traditionnelles, les textures sont affichées à l'écran après avoir subi un nombre de transformations. Les textures sont toujours vues par le GPU comme un tableau de deux dimensions (largeur x hauteur). Une texture est donc l'équivalent d'un tableau de deux dimensions traditionnel (soit array[x][y]) mais en mémoire vidéo (vRAM) plutôt qu'en mémoire conventionnelle (RAM). Chaque point d'une texture, défini par sa position en x et y dans l'espace mémoire, s'appelle un *texel*.

Dans le cas de GPGPU, nous ne voulons généralement pas afficher les résultats de nos calculs sur l'écran, mais bien les garder en mémoire. Pour ce faire, il est possible d'effectuer des transformations sur une texture d'entrée et sauver le résultat dans une texture de sortie. Le GPU garde alors toutes ses données dans sa mémoire vidéo (vRAM) et n'affiche rien à l'écran. Il est ensuite possible de copier cette texture de sortie en mémoire globale (RAM) et d'accéder aux données de façon traditionnelle (comme dans un tableau).

Il est important de noter qu'à moins d'utiliser certaines astuces présentées plus tard dans ce document (section 3.4), les textures d'entrée et de sortie doivent avoir les mêmes dimensions.

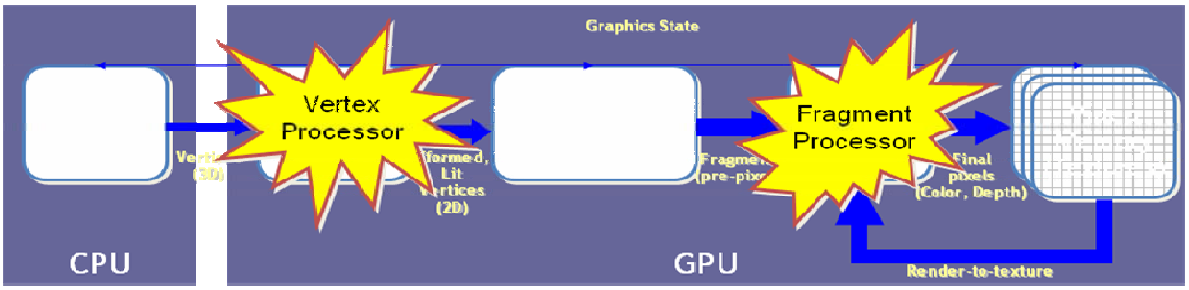
2.2 Format des données

Il existe deux formats pour chaque texel : RGBA ou LUMINANCE. Les texels de format RGBA contiennent quatre données en point flottant alors que les texels de luminance n'en contiennent qu'une. Par souci de simplicité, les données peuvent être exprimées selon quatre formats : *float4*, *float3*, *float2* et *float*. Ce ne sont que des structures qui contiennent le nombre de champs correspondant au chiffre à la fin de leur nom. Un *float4*, par exemple, est une structure qui contient des champs .x, .y, .z et .w, dans cet ordre, qui sont chacun des *floats*.

Actuellement, tous les GPU fonctionnent en simple précision (32 bits). Lorsque l'on programme pour le CPU, le processeur FPU (***F***loating ***P***oint ***U***nit) calcule toujours en double précision (64 bits) et tronque ensuite le résultat pour l'ajuster au conteneur de 32 bits. Cela peut occasionner des différences importantes entre les résultats d'un calcul sur CPU et sur GPU.

2.3 Transformations (traitement des données)

La pipeline graphique contient deux processeurs programmables, le *fragment processor* (ou *pixel processor*) et le *vertex processor*. Le programme qui est exécuté sur le *fragment processor* est appelé *fragment shader* (*ou pixel shader)* et le programme exécuté sur le *vertex processor* est le *vertex shader*. Il est important de noter que les *shaders* sont appliqués sur des éléments graphiques, et non sur des textures directement.



2.3.1 Vertex shaders[2]

Les *vertex shaders* permettent d'exécuter des opérations mathématiques sur des objets tridimensionnels, ou, plus précisément, sur les sommets de ces objets. Un sommet est un coin d'un polygone ou polyèdre (un triangle a trois sommets alors qu'un tétraèdre en a quatre). Par exemple, un sommet est toujours défini par sa position spatiale (en coordon-

[2] http://en.wikipedia.org/wiki/Vertex_shader

nées cartésiennes) mais peut aussi être défini par sa couleur, l'éclairage auquel il est soumis et, surtout, à quelles coordonnées de texture il doit correspondre.

2.3.2 Fragment shader

Les *fragment shaders* permettent d'exécuter des opérations mathématiques sur chaque pixel d'un objet graphique de façon parallèle et indépendante.

2.3.3 Rasterizer

Le *rasterizer* sert surtout à interpoler des coordonnées de texture et n'est que peu ou pas utilisé dans les applications de GPGPU.

2.3.4 Application en vision par ordinateur

En vision par ordinateur, nous allons généralement charger une image (ou une trame de séquence vidéo) et la transmettre directement sous forme de texture dans la mémoire vidéo pour ensuite la traiter. Pour ce faire, il faut s'assurer que chaque élément de notre groupement de données soit traité individuellement par le *fragment shader*. En d'autres termes, chaque élément de nos données d'entrée doit correspondre à un texel (point dans la texture). En programmation graphique, le fait de dessiner une forme en lui appliquant une texture aura pour effet d'appliquer les *shaders* sur cette forme. Pour dessiner une forme simple, on définit des sommets qui correspondront aux coins de cette forme. Donc, pour dessiner un carré, on déclarera quatre sommets qui correspondront aux coins du carré et l'API graphique se chargera de remplir le carré, de lui appliquer la texture et de rouler les *shaders* sur les différents sommets/points de ladite forme. Par conséquent, l'astuce est de dessiner un carré de la même dimension que notre image et de lui attacher une texture qui contient nos données à traiter. Si l'on définit un *vertex shader*, il n'agira que sur les quatre coins du carré. Si l'on définit un *fragment shader*, il agira sur tous les points du carré, donc tous les points de la texture (puisque les dimensions du carré et la texture sont les mêmes), donc tous nos éléments de données. Le programme inclut en annexe pourra éclaircir cette façon de faire.

Voici donc ce que nous devons faire étape par étape :

- Charger nos données dans un tableau
- Créer une texture de la même dimension que notre tableau
- Charger les données de notre tableau dans la texture

- Appliquer la texture sur la prochaine forme dessinée
- Dessiner un rectangle de la même dimension que notre tableau/texture (le rectangle ne sera jamais affiché sur l'écran si on lui dit de récupérer les résultats graphiques dans une autre texture; le fait de dessiner ne fait qu'indiquer à l'API graphique que l'on est prêt à exécuter tous nos *shaders* sur nos primitives graphiques).
- Le résultat est récupéré dans une autre texture
- La texture peut être copiée dans un tableau pour retrouver nos données en mémoire globale.

3. PARADIGME DE *STREAMING*[3]

Puisque la force des GPU réside dans leur capacité à effectuer des calculs en parallèle, il est nécessaire d'adopter un nouveau modèle de pensée lors de la programmation d'algorithme : le *streaming*. Ce paradigme est basé sur le fonctionnement d'une boucle *for_each*[4] excepté que les données sont traitées de façon parallèle. On définit d'abord deux termes importants : un *stream* est un groupement de données sur lequel des opérations vont être effectuées en parallèle, alors qu'un *kernel* est une fonction qui contient les instructions à effectuer sur un *stream.*

3.1 Lecture/écriture

Lorsqu'un *kernel* est appliqué à un *stream*, il applique toutes ses instructions sur chaque élément du *stream* et écrit à la même place sur le *stream* de sortie. En d'autres termes, supposons que le *kernel* soit rendu à lire la donnée située aux coordonnées (5,7) dans le *stream* d'entrée; il effectuera ses instructions et écrira à la même position dans le *stream* de sortie (5,7). Pour cette raison, même si ces lectures et écritures sont effectuées en parallèle, on les appellera lecture et écriture séquentielle, à cause de la correspondance entre le *stream* de sortie de celui d'entrée.

3.2 Gather

En langage de *streaming*, le *gather* est une lecture directe, c'est-à-dire une lecture d'une valeur à un indice donné dans le *stream*. En termes généraux, le *gather* est analogue

[3] http://graphics.stanford.edu/projects/brookgpu/lang.html
[4] http://www.sgi.com/tech/stl/for_each.html

à l'opération a = b[i]; Le *gather* est permis et supporté dans les *fragment shaders*, mais pas dans les *vertex shaders*. Par conséquent, il sera facile à implémenter et à utiliser dans nos applications de GPGPU.

3.3 Scatter

Le *scatter* est une écriture directe, c'est-à-dire l'écriture d'une valeur à un indice donné dans le *stream*. En termes généraux, le *scatter* peut être représenté par l'opération a[i] = b; Puisque les *fragment shaders* écrivent au même indice en mémoire qu'ils ont lu (voir lecture/écriture ci-haut), ils ne supporteront pas le *scatter*. Le seul moyen intuitif de faire du *scatter* en GPGPU avec les *vertex shaders* est de dessiner des points au lieu d'un carré (voir application en vision par ordinateur ci-haut). Il est alors possible d'assigner à chacun de ces points une coordonnée de texture et ainsi d'aller écrire aux coordonnées appropriées dans la texture de sortie.

Puisqu'un sommet contient la position, la luminance, la couleur et les coordonnées de texture, il est possible, en dessinant (w*h) points au lieu d'un rectangle de dimension (w * h), de contenir notre image dans un nuage de point au lieu d'un rectangle plein. Ainsi, chaque pixel correspondra à la fois à un sommet et à un fragment. Nos données seront donc contenues dans la couleur du sommet, par exemple, et l'indice où il doit être écrit sera sa coordonnée de texture. Cette méthode, toutefois, est souvent très lourde et peu performante et donc n'est utilisée que si elle n'est pas contournable (par exemple, pour la génération d'un histogramme[5]).

3.4 Réductions

Une réduction est une opération où le *stream* de sortie doit être plus petit que le *stream* d'entrée. Les opérations effectuées dans une réduction doivent être à la fois commutatives (a * b == b * a) et associatives (a * (b * c) = (a * b) * c). Par exemple, la somme des éléments d'un *stream* répond à ces deux contraintes parce que a+b+c+d = (a+b)+(c+d) = c+a+d+b.

De fait, certaines opérations de réduction courantes sont la somme, le produit, le maximum et le minimum. Certaines opérations de réduction qui ne répondent pas aux contraintes susmentionnées sont la soustraction et la division.

Il est possible d'effectuer une réduction d'un *stream* à une valeur simple (par exemple, pour trouver la valeur maximale dans tous les éléments du *stream*) ou à un *stream* plus petit. Soit un *stream* d'entrée *E* de dimension (100 x 200) et un *stream* de sortie *S* de di-

[5] http://ati.amd.com/developer/gdc/2007/GPUHistogramGeneration_preprint.pdf

mension (50 x 20). *S* est 2 fois plus petit que *E* dans la dimension X et 10 fois plus petit que E dans la dimension Y. Si l'on applique une réduction de somme sur E et S, chaque élément de S sera la somme d'une tuile de (2x10) valeurs du *stream* d'entrée.

3.5 Correspondance

Dans notre cas, le paradigme de *streaming* n'est qu'un niveau d'abstraction de la programmation sur GPU. Par souci de simplicité, nous allons donc utiliser les termes du *streaming* dans le présent document, selon le tableau de correspondance suivante :

ANSI C++	API du GPU (directX ou OpenGL)	Streaming
Tableau (array)	Texture	Stream
Foncteur de la fonction for_each	Fragment Shader	Kernel
Lecture directe : a = b[i]	Lecture directe (varie selon le langage de shader utilisé)	Gather
Écriture directe : a[i] = b	Écriture directe (varie selon le langage de shader utilisé)	Scatter

4. APIS

L'un des premiers dilemmes auxquels fera face le programmeur GPGPU est le choix de l'API à utiliser pour sa carte graphique. Nous séparons les APIs en trois catégories : les APIs généraux, qui permettent de gérer les données (tableaux, textures et transferts de mémoire), les langages de *shaders*, qui permettent de définir différents *kernels* et les APIs spécialisés, qui permettent d'accéder directement à la carte vidéo.

4.1 Généraux (graphique)[6]

Les deux APIs généraux permettant de gérer les textures et le rendu de nos données sont DirectX et OpenGL. La communauté académique tend à préférer OpenGL pour la portabilité qu'elle permet entre les plateformes ainsi que son mécanisme d'extensions. Ce mécanisme permet aux fournisseurs d'ajouter des fonctionnalités à l'API dès que le matériel supporte ces fonctionnalités, par opposition à DirectX de Microsoft, qui dépend des mi-

[6] http://www.gpgpu.org/wiki/FAQ

ses à jour de la compagnie. De fait, très peu de documentation existe pour DirectX. Il est à noter que les APIs généraux ne sont pas conçus pour la programmation GPGPU et qu'il faut utiliser quelques astuces pour obtenir les résultats désirés. Cela nécessite une compréhension assez large de l'API choisi et alourdit considérablement le code. Les APIs généraux sont généralement plus ajustables et donc plus rapides que les langages de haut niveau, mais moins que les APIs spécialisés.

4.1.1 DirectX9

DirectX9, développé par Microsoft, utilise les langages de *shaders* HLSL ou Cg. Les deux désavantages principaux de DirectX, tels que susmentionnés, sont que les mises à jour dépendent de Microsoft et que les applications développées avec cet API ne pourront être utilisées que sur Windows. Puisqu'aucun tutoriel en ligne n'est basé sur directX9 et que l'arrivée de DirectX10 est imminente et changera complètement le paradigme de programmation (langage unifié de *shaders*, arrivée du *Geometry Shader*, etc.), cet API n'a pas été exploré en profondeur.

4.1.2 OpenGL

OpenGL est l'API le mieux supporté par les académiciens et divers chercheurs publiant sur Internet. Pour se familiariser avec l'API, il existe d'excellents tutoriels.

Pour comprendre les fonctionnalités de base de l'API : http://nehe.gamedev.net/

Pour comprendre les principes de base du GPGPU avec OpenGL : http://www.mathematik.uni-dortmund.de/~goeddeke/gpgpu/index.html

Il y aura de plus, en annexe, un programme OpenGL qui calcule la transformée Census d'une image, en guise d'exemple de synthèse de plusieurs concepts des tutoriels (notamment les FBO et les PBO). Nous décrirons ici quelques concepts (extensions) plus obscurs qui sont essentiels à la compréhension de la programmation sur OpenGL.

4.1.2.1 *pbuffers*[7]

Les *pbuffers* sont (étaient) des espaces mémoire utilisés comme cible de rendu. En d'autres termes, ils contenaient le résultat d'un *kernel* et permettaient d'effectuer des calculs sans les afficher à l'écran. Les *pbuffers* généraient un changement de contexte (analogue à un changement de *thread*) et étaient généralement complexes à utiliser. Ils ont été, à toutes

[7] http://lwjgl.org/wiki/doku.php/lwjgl/tutorials/opengl/pbuffervsfbo

fins pratiques, remplacés par les FBO et ne devraient jamais être utilisés, à moins d'être contraint à utiliser une vieille librairie qui les utilise déjà.

4.1.2.2 FBO[8]

Les FBO (***F**rame **B**uffer **O**bjects)* remplacent maintenant les *pbuffers*. Ils contiennent un espace mémoire (texture) où seront placés les résultats d'un *shader* (*kernel*). Ils sont plus rapides et plus faciles à utiliser que les *pbuffers*. Le seul désavantage qu'ils ont par rapport aux *pbuffers* est qu'ils ne présentent pas autant de compatibilité avec les cartes ATI : le multi-échantillonnage et le stencil ne sont pas disponibles sur toutes ces cartes. Cependant, puisque ces deux fonctionnalités ne sont pas utilisées pour le GPGPU, les FBO remplacent avantageusement les *pbuffers*.

4.1.2.3 PBO[9]

Les PBO (***P**ixel **B**uffer **O**bjects*) sont un espace mémoire qui permet d'éviter une copie inutile en mémoire lors du transfert des données de la RAM vers la vRAM; le transfert s'effectue 4 fois plus rapidement sur une NVIDIA Geforce 7800 GTX. Le gain obtenu lors du transfert de la vRAM vers la RAM est marginal. Ils permettent aussi que ces transferts soient asynchrones, c'est-à-dire que le CPU pourra travailler pendant que les données sont transférées au lieu de bloquer en attendant le transfert. Il existe un excellent tutoriel présentant leur utilisation à l'adresse suivante : http://www.mathematik.uni-dortmund.de/~goeddeke/gpgpu/tutorial3.html. Puisque les transferts de mémoire sont presque toujours le goulot d'étranglement dans les applications GPGPU, les PBOs sont probablement l'optimisation la plus significative et facile à implanter.

4.2 Langages de *Shaders*

4.2.1 Assembleur

Puisque l'assembleur n'est plus tellement utilisé pour coder des *shaders* (ceux-ci devenant de plus en plus complexes), nous n'énumérerons que les profils disponibles les plus récents lors de la compilation avec Cg. Un profil de compilation est en fait le langage d'assembleur dans lequel la librairie compilera le programme codé en langage de plus haut niveau. Par exemple, le profil ps20 (Pixel Shader 2.0) correspond au langage d'assembleur du même nom, utilisé seulement sur les plus vieilles cartes qui ne supportent pas le ps30 (Pixel Shader 3.0). Les profils (ou langages) débutant par les lettres 'fp' sont propres à

[8] http://www.gamedev.net/community/forums/topic.asp?topic_id=452135&whichpage=1&#2988431
[9] http://www.gpgpu.org/w/index.php/Glossary

NVIDIA. Ces profils correspondent d'ailleurs aux différents profils disponibles lors de la compilation avec *brook*.

Langage	Plate-forme	API	GPU supportés	Maximum d'instructions	Notes
Arb	Windows Linux	OpenGL	Tous	64	
Ps20	Windows	DirectX9	Tous	64	Supporte la prédication.
Ps30	Windows	DirectX9	Tous	65536	
Fp40	Windows Linux	OpenGL	NVIDIA	65536	Idem Fp30 Supporte les branches sans prédication. Supporte les boucles non déroulées.
Fp30	Windows Linux	OpenGL	NVIDIA	65536	Inclus des fonctions de packing, par exemple pour empaqueter 4 chars dans un *float*. Toutes les branches sont résolues par prédication. Toutes les boucles sont déroulées.
Fp20	Windows Linux	OpenGL	NVIDIA	65536	

4.2.2 Cg

Cg (C for Graphics) est un compilateur développé par NVIDIA qui permet d'écrire des *shaders* dans un langage très semblable au standard C. Le compilateur traduit ensuite le code en assembleur pour GPU. Le code compilé fonctionnera sur des cartes ATI et NVIDIA, mais sera optimisé pour les cartes NVIDIA.

4.2.3 HLSL

HLSL (***H***igh ***L***evel ***S***hading ***L***anguage) est le langage de programmation de *shaders* de haut niveau de DirectX9.0.

4.2.4 GLSL

GLSL (OpenGL ***S***hading ***L***anguage) est l'équivalent HLSL mais pour la librairie OpenGL. C'est, par conséquent, le seul langage qui soit complètement indépendant des fournisseurs de GPU et d'APIs et donc le plus portable et efficace. En effet, Cg fonctionne sur Linux et Windows et supporte NVIDIA et ATI, mais est optimisé pour NVIDIA et HLSL oblige l'utilisation de DirectX9.

4.3 Spécialisés (GPGPU)

Les APIs spécialisés ont été conçus spécifiquement pour le GPGPU. De fait, ils ne sont pas limités par la couche d'abstraction graphique, qui a été créée a priori pour les jeux, et fonctionnent plus rapidement. Toutefois, puisqu'ils sont développés par les fournisseurs de cartes vidéo, ils ne sont pas portables entre différentes architectures. Cette technologie est encore très jeune (la plupart des API sont sortis début 2007) et nécessite d'utiliser des supports matériels très récents et dispendieux. Cependant, puisque les APIs spécialisés ne dépendent pas d'un modèle de programmation conçu pour les graphiques et donc étroitement restreint et difficile à utiliser, ils sont probablement la solution de l'avenir.

API	Plate-forme	Langage	GPU supportés	Coût	Date de parution
CUDA (NVIDIA)	Windows Linux	Standard C	NVIDIA G8X	Gratuit	15 février 2007
CTM (AMD/ATI)	Windows Linux	Assembleur	ATI R580	Gratuit	2 février 2007
Accelerator (Microsoft)	Windows	C#	Toutes les cartes supportant DirectX9, avec au moins 256MB de vRAM et supportant le Shader Model 3.0	Évaluation gratuite, contacter la compagnie pour le coût d'une licence	16 octobre 2006

4.3.1 CUDA[10]

CUDA est un API spécialisé développé par NVIDIA qui donne un accès direct aux instructions natives du GPU, sans avoir à passer par les *shaders*. Il permet de coder en lan-

[10] http://developer.nvidia.com/object/cuda.html

gage standard C et de compiler ensuite pour le GPU. Il permet le *scatter*, c'est-à-dire l'écriture directe en mémoire vidéo, ce qui est un énorme avantage par rapport aux autres APIs et permet de traduire beaucoup plus efficacement des algorithmes complexes. Il est aussi supposé permettre des transferts plus rapides entre la RAM et la vRAM. CUDA a été développé par Ian Buck, qui est aussi le créateur de brookGPU et donc partage des similarités avec ce dernier langage. La couche graphique est complètement cachée, mais il est nécessaire de bien connaître l'architecture matérielle d'un GPU pour pouvoir programmer de façon optimale. Toutefois, puisque l'API est très jeune, il ne fonctionne que sur les plus récentes cartes NVIDIA (G8X) et donc, ses performances et sa facilité d'utilisation n'ont pu être testées. Aussi, il est possible qu'une mise à jour du pilote rende un programme inutilisable[11].

4.3.2 CTM[12]

CTM est un API spécialisé développé par AMD/ATI, basé sur leur *Data parallel virtual machine* (DPVM) qui donne accès direct aux instructions natives du GPU, sans avoir à passer par les *shaders*. L'API utilise un tout nouveau langage, semblable à l'assembleur et, par conséquent, très complexe et peu intuitif à utiliser; cependant, il est possible d'utiliser brookGPU ou HLSL et de compiler en langage CTM. Il faut alors faire confiance au compilateur pour optimiser notre code. Tout comme CUDA, il supporte le *scatter* et est supposé permettre des transferts de mémoire plus rapides. En effet, il permet de lire un *float4* ou quatre *float* en un cycle d'horloge et permet de lire ou d'écrire directement à partir de la mémoire principale, ce qui devrait donner des performances relativement semblables (quoiqu'un peu supérieures) aux PBO avec OpenGL. Toutefois, puisqu'il nécessite une carte ATI Radeon R580, il n'a pas pu être testé. Puisque l'on accède directement à la carte graphique, une mise à jour du pilote ne peut pas potentiellement rendre un programme inutilisable.

[11] http://forum.beyond3d.com/showthread.php?t=41083
[12] http://www.gpgpu.org/sc2006/slides/08a.segal.ctm.pdf

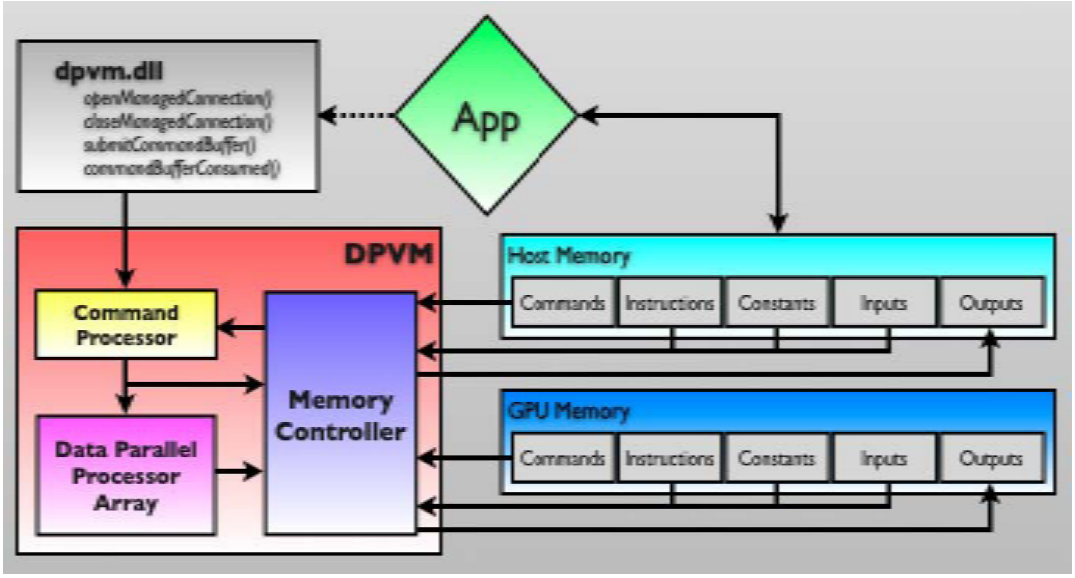


Figure 2 - "Data Parallel Virtual Machine" (CTM)

App	Benefit	Features
Matrix-Matrix Multiply	x10	CB, ISA, mem-formats, mem-offsets, interleaving, fetch4
FFT	x2	CB, ISA, interleaving
GPURay	x2	CB, mem-formats
QJulia	x2	CB, mem-formats

Figure 3 – Performance de CTM[13]

4.3.3 Accelerator[14]

Accelerator, développé par Microsoft, sert à cacher la couche graphique et à accéder au GPU dans un environnement .NET (C#). Il nécessite DirectX9 et Visual Studio 2005 et donc n'est utilisable que sur des machines Windows. L'API est gratuit pour un usage non-

[13] App est un traitement numérique (par exemple FFT sont les transformées de Fourier). Le *benefit* n'est pas bien défini en ce sens qu'on ne sait pas trop à quoi CTM est comparé. Les algorithmes de comparaison sont probablement implémentés de façon optimisée sur OpenGL.

[14] http://research.microsoft.com/research/downloads/Details/25e1bea3-142e-4694-bde5-f0d44f9d8709/Details.aspx

commercial seulement. Le coût pour un usage commercial n'est pas fixe et doit être discuté avec la compagnie. À cause des coûts éventuels reliés à son utilisation ainsi que les coûts de performance et de portabilité reliés à l'utilisation du C#, cette solution n'a pas été explorée.

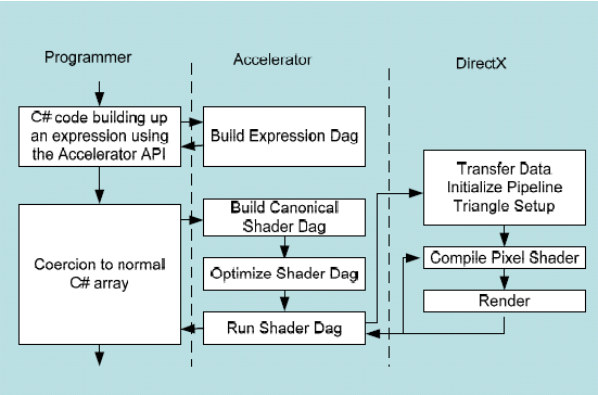


Figure 4 - Accelerator Workflow (Just in time compiler)

5. LIBRAIRIES DE HAUT NIVEAU

5.1 BrookGPU[15]

BrookGPU est une librairie permettant de faire complètement abstraction de la couche graphique pour programmer sur GPU. Le programmeur peut coder ses *kernels* dans un fichier et compiler avec le compilateur de brookGPU qui traduira les *kernels* en langage assembleur de *shaders*. Une librairie permet de gérer les textures et les accès mémoire via DirectX9, OpenGL ou CTM. La librairie est portable sur la plupart des cartes NVIDIA et ATI et peut rouler sur Windows (DirectX ou OpenGL) ou Linux (OpenGL seulement). Il est à noter cependant que la section OpenGL est désuète car elle utilise les *pbuffers* au lieu d'utiliser les FBO et PBO (plus rapides et plus sécuritaires). La section directX9 a été utilisée dans des applications de grande envergure comme Folding@Home. Des forums sont mis en place et il est possible d'obtenir une réponse à ses questions dans les 24 heures. Des tests indiquent que brookGPU atteint 80% de la performance de code optimisé à la main[16], quoiqu'il ne soit pas clair si ces tests tiennent compte des optimisations possibles de transfert de mémoire (PBO). BrookGPU ne supporte pas le *scatter* sur GPU, même si on l'utilise avec CTM : le streamScatterOp inclus dans la librairie est une émulation de la fonctionnalité et est très lent. En raison de son excellente documentation et de sa portabili-

[15] http://graphics.stanford.edu/projects/brookgpu/
[16] http://www.gpgpu.org/vis2005/

té, la librairie brookGPU est celle qui a été utilisée pour la plupart des tests subséquemment présentés dans ce document. Une section sera réservée dans l'annexe pour démontrer comment interfacer brookGPU avec du code C++ dans Visual Studio. Le code source de BrookGPU est libre et gratuit.

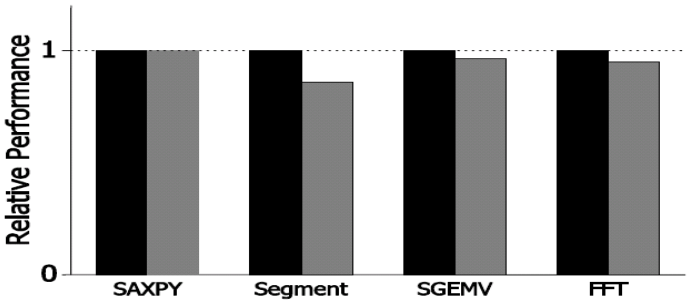


Figure 5 - Performance de code optimisé à la main (en noir) vs brookGPU (en gris).

5.2 Sh[17]

Sh est une librairie permettant de programmer des *shaders* (*kernels*) directement en C++ sans passer par les autres langages de *shaders*. Elle diffère de brookGPU en ce sens qu'elle ne fait pas abstraction de la couche graphique, et ne nécessite pas de compilateur externe. Il s'agit d'une librairie méta programmée, c'est-à-dire qu'elle génère du code à partir du code C++, notamment en se servant des propriétés des macros et des types génériques (*templates*). Elle permet surtout d'éviter d'avoir à coder du *glue code*, i.e. du code servant à passer des paramètres aux *shaders*. Sh est supportée sur Linux et Windows, et n'a pas d'attache particulière avec un API graphique (OpenGL et DirectX). Cependant, la librairie est très pauvrement documentée et le support est inexistant. En effet, la librairie a cessé d'être maintenue en 2006, au profit de sa version commerciale, RapidMind[18].

5.3 RapidMind[19]

RapidMind est le successeur commercial de Sh. La librairie est gratuite pour l'utilisation personnelle et/ou de recherche, mais doit être achetée pour toute commercialisation ou déploiement d'un produit qui l'utilise (à un prix qui doit être discuté avec les vendeurs). Il s'agit d'une librairie de méta programmation et donc ne nécessite pas non plus de compilateur externe. La librairie démontre des performances entre deux et 32 fois plus rapides que des implémentations sur des CPU de haut niveau et est supposément plus rapide que du code optimisé à la main pour les *shaders* : *"The performance of the RapidMind implementations of these benchmarks was equivalent to the best available GPU im-*

[17] http://libsh.org/wiki/index.php/Frequently_Asked_Questions
[18] http://en.wikipedia.org/wiki/Lib_Sh
[19] http://www.rapidmind.net/product.php

plementations done through OpenGL. Specifically, Govindaraju et al [11] report peak performance numbers of 6.1 Gflop/s and 17.6 Gflop/s for the FFT and SGEMM respectively. Govindaraju's implementations claim to be faster than any existing GPU implementations of the same algorithms. However, the peak performance numbers of the RapidMind implementations are 7.6 Gflop/s and 31.3 Gflop/s, respectively. The RapidMind versions of these benchmarks were also significantly easier to implement, since the algorithms could be expressed more directly. [20]" La librairie est disponible sur Linux et Windows et est compatible avec la plupart des cartes NVIDIA et les cartes ATI de famille x1x00.

5.4 PeakStream

PeakStream était une librairie apparemment intuitive, rapide et facile à utiliser[21]. Toutefois, la compagnie qui le produit a été achetée par Google[22] et n'est plus disponible publiquement.

5.5 GPUCV[23]

La librairie GPUCV est une extension d'OpenCV, avec laquelle elle possède une connectivité idéale. Bâtie sur l'API OpenGL, elle peut détecter les extensions (FBO, PBO, etc.) disponibles sur la carte utilisée et optimiser ses algorithmes conséquemment. Toutefois, son développement a été arrêté en 2006 au stade alpha et le support est inexistant. La documentation est très limitée mais les échantillons de code fournis avec la librairie devraient permettre de comprendre comment l'utiliser. La librairie fait beaucoup de travail caché, comme la décision d'exécuter un algorithme sur le CPU ou le GPU et les copies de RAM vers vRAM et vice-versa, ne laissant pas le contrôle à l'utilisateur. De fait, il est difficile de déterminer si l'algorithme utilisé est optimal. Aussi, il est complexe et fastidieux d'ajouter de nouveaux opérateurs et la stabilité est médiocre : lors des premiers essais, un programme d'exemple provenant de la librairie boguait sur le premier appel à la classe 'cout'.

5.6 OpenVIDIA[24]

La librairie OpenVIDIA est en fait un *wrapper* autour d'OpenGL qui permet de cacher une partie de la couche graphique lors de la programmation sur GPGPU. Elle n'utilise pas les PBO, donc ses temps de transfert ne sont pas optimisés pour OpenGL. Une partie

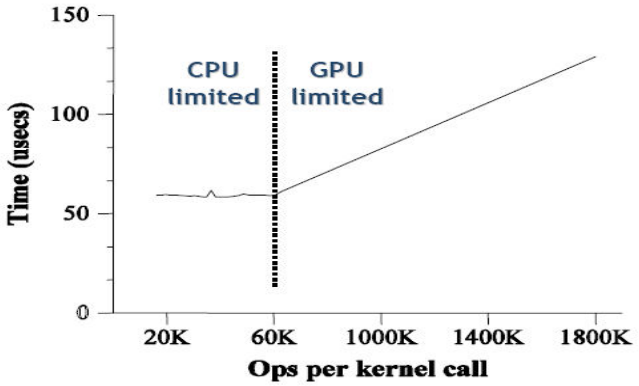
[20] http://www.rapidmind.net/pdfs/RapidMindGPU.pdf
[21] http://episteme.arstechnica.com/eve/forums/a/tpc/f/174096756/m/277001683831/inc/1
[22] http://arstechnica.com/news.ars/post/20070605-google-buys-peakstream-inc.html
[23] http://picolibre.int-evry.fr/projects/gpucv/
[24] http://openvidia.sourceforge.net/

de la couche graphique est cachée (création et compilation des *shaders*/*kernels*), mais il est nécessaire d'en utiliser une autre (création des textures, manipulation des *buffers*). Il est donc préférable de connaître le fonctionnement complet d'OpenGL pour comprendre tout ce qui se passe et ainsi, l'économie de temps et de code faite par les appels à la librairie OpenVIDIA est marginale. Bref, la librairie n'est pas optimisée, est mal documentée, et nécessite que l'on comprenne de toutes façons tout ce qu'elle implémente. Par conséquent, OpenVIDIA ne devrait être utilisée qu'à titre d'exemple (elle contient plusieurs *fragment shaders* qui, eux, sont optimisés). En effet, les algorithmes qui y sont déjà implémentés (*features* et *canny*) sont efficaces, mais il est inutilement complexe et non optimal d'essayer d'y ajouter d'autres algorithmes. Un algorithme implémenté de la même façon sur brookGPU aura des performances similaires et une complexité moindre. En d'autres termes, coder pour OpenVIDIA est presque identique à coder sur OpenGL au niveau de connaissances et moins efficace en termes de performance. De plus, la librairie est plus ou moins stable et peut occasionner des plantages qui sont difficiles à déboguer : notre implémentation de C*anny* pour OpenVIDIA plante à la création du premier FBO_FILTER et on ne peut pas déboguer le code pas à pas.

6. ÉLÉMENTS DE PERFORMANCE

6.1 Kernel overhead[25]

Le *kernel overhead* est le temps requis pour un appel à un *kernel*. Il dépend du temps requis pour que le CPU génère la géométrie avec laquelle il peut utiliser le GPU, le temps d'accès au pilote graphique et le temps de copie des *kernels* dans la mémoire d'instructions du GPU.

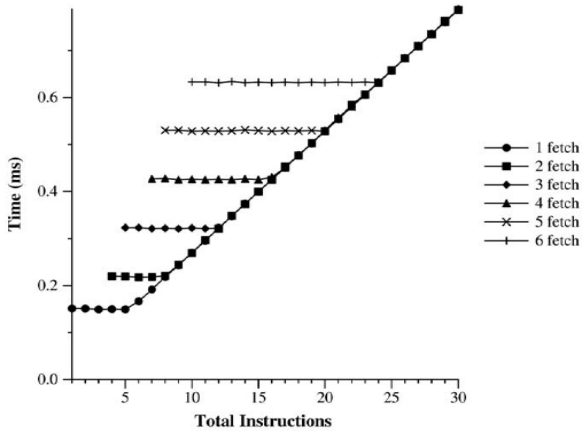


[25] http://www.gpgpu.org/s2005/slides/buck.Strategies.ppt

6.2 Bande passante

6.2.1 Coût d'un *gather*

Le graphe suivant sert à illustrer le faible coût d'un *gather* dans une texture pour une carte relativement récente (ATI X1800XT)[26]. L'on peut en effet observer qu'avec seulement 25 instructions (d'assembleur, sur le GPU), le coût de six *gathers* est caché par les calculs arithmétiques, c'est-à-dire que le temps requis pour exécuter une passe est indépendant du nombre de *gathers*.



Nos tests ont démontré qu'une passe (image 640x480 RGB, format 32F) sur un *kernel* qui fait neuf *gathers* prend 0.7ms alors qu'une passe sur un *kernel* qui fait une simple addition prend 0.1ms. Cela signifie que pour une passe sur l'image, chaque *gather* coûte environ 0.067ms.

6.2.2 Coût d'une passe

La carte à l'étude (NVIDIA 7800) possède une bande passante théorique de 54.4 GB/s, ce qui signifie qu'une passe pour une image RGB 640x480 en format 32F sans aucun calcul devrait être d'approximativement 0.0676ms. Les tests pratiques avec brookGPU indiquent que cette passe prend 0.114 ms.

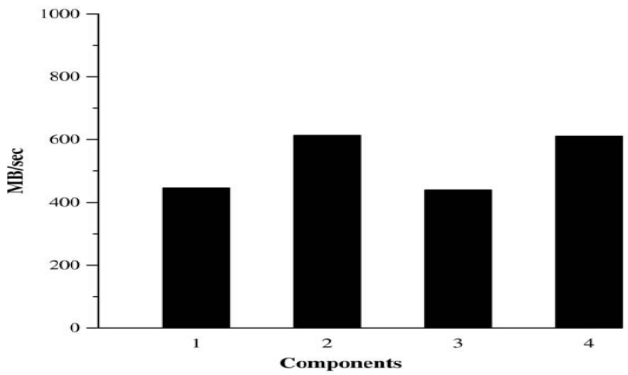
[26] http://www.gpgpu.org/sc2006/slides/10.houston-understanding.pdf

6.3 Transfert de mémoire[27]

Les transferts de mémoire de la mémoire principale (RAM) vers la mémoire vidéo (vRAM) et vice-versa sont souvent l'opération la plus coûteuse lors de la programmation GPGPU. Ainsi, souvent, pour mieux promouvoir leurs produits, certaines compagnies omettent d'inclure ces temps dans les calculs de performance de leur algorithme. Sur les graphiques présentés, on a la vitesse de transfert sur l'axe des ordonnées (en Mo / sec) et la taille des composantes sur l'axe des abscisses (*float*, *float2*, *float3*, *float4*). On remarque qu'en général, il est plus rapide de transférer les données en *float* ou en *float4*.

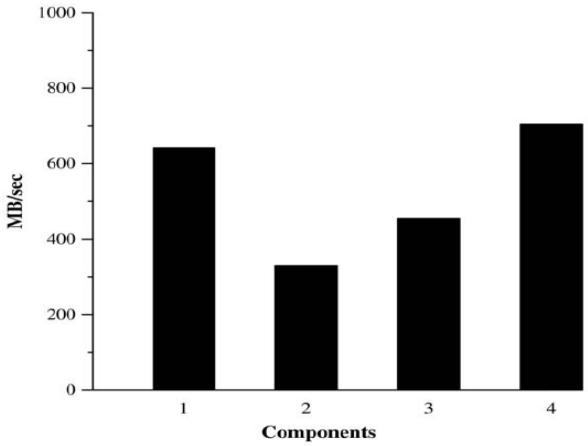
6.3.1 Download

Ce qu'on appelle le *download* est le transfert de la mémoire à partir de la mémoire principale RAM vers la mémoire vidéo (vRAM). Il est important de noter que les résultats suivants ont été réalisés sans l'utilisation des PBO sur OpenGL, qui pourraient théoriquement multiplier la vitesse par un facteur allant jusqu'à quatre. L'utilisation d'un API spécialisé comme CUDA ou CTM pourrait générer des gains semblables.



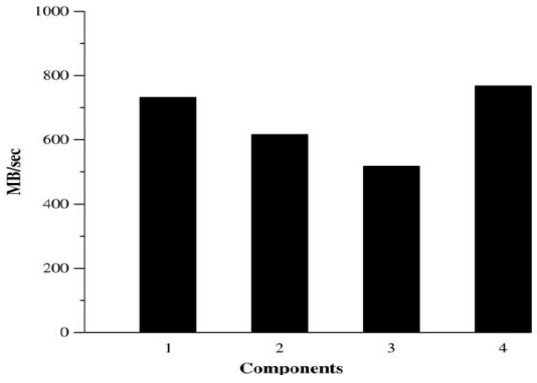
***Download* sur la carte ATI x1900XTX**

[27] http://www.gpgpu.org/sc2006/slides/10.houston-understanding.pdf



***Download* sur la carte NVIDIA 7900GTX**

Il est possible d'observer sur les deux figures précédentes que le transfert s'effectue à une rapidité de l'ordre de 500 MB par seconde. Cela signifie que pour une image RGB de 640x480 en format 32f, le temps de transfert serait de 7.37ms. Il est difficile de reproduire ces tests avec brookGPU, car une lecture répétée du même *stream* occasionne des optimisations au niveau du pilote (ce qui nous donne un temps de lecture d'environ 1.3ms) et une lecture d'une série d'images nous donne un temps supérieur, à cause du temps passé par le CPU à lire les images.

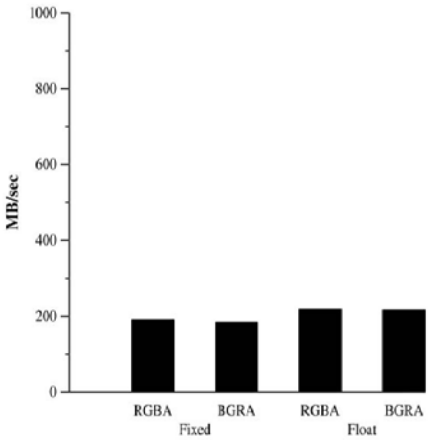


Download sur la carte NVIDIA 8800GTX

La figure précédente sert simplement à illustrer que la prochaine génération de GPU ne représentera pas un gain significatif en termes de vitesse de *download*.

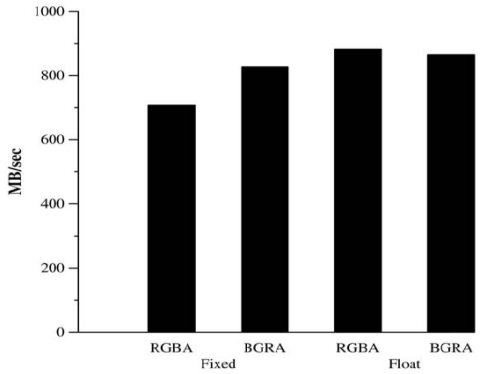
6.3.2 Readback

Le *readback*, par opposition au *download*, est le transfert de la mémoire à partir de la mémoire vidéo (vRAM) vers la mémoire principale (RAM). L'utilisation de librairies spécialisées comme CUDA et CTM pourraient générer des gains non négligeables.



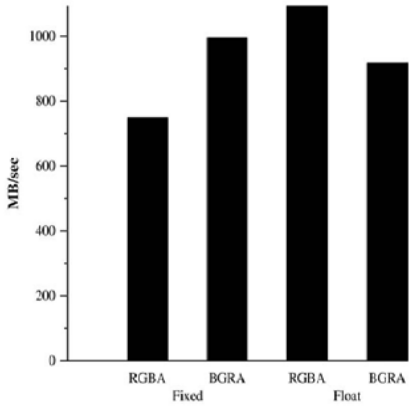
***Readback* sur la carte ATI X1900XTX**

La figure précédente illustre la faible performance de transfert de mémoire vers la mémoire hôte. La vitesse de la carte ATI est de l'ordre de 200 MB, ce qui signifie que pour une image 640x480 RGB en format 32F, le temps de transfert serait de 18.432ms. La carte ATI n'a pas été testée avec brook.



***Readback* sur la carte NVIDIA 7900GTX**

On peut voir que la vitesse de *readback* sur la carte NVIDIA est nettement supérieure, de l'ordre de 700 MB/s. Le temps de transfert de la même image serait donc d'environ 5.266ms. Pour les mêmes raisons que le *download*, les tests n'ont pas pu être effectué avec brookGPU.



***Readback* sur la carte NVIDIA 8800GTX**

On peut simplement voir sur la figure précédente que la carte de prochaine génération n'offre qu'un gain marginal de performances de *readback*. En effet, même si on peut observer un gain d'environ 20% pour le format RGBA *float*, les transferts de mémoire demeurent le goulot d'étranglement dans les algorithmes de traitement GPGPU. Cela devient plus clair dans la section suivante.

6.4 Sommaire

En omettant les calculs, qui ont une durée difficilement quantifiable (ils peuvent être optimisés par le pilote et donner des résultats dépassant même le maximum théorique), voici les temps d'un appel typique à un *kernel*, avec une carte NVIDIA 7800, pour une image RGB de 640x480 en format 32f :

1) *Kernel overhead* : 0.06ms
2) *Download* de la mémoire : 7.37ms
3) Passe (sans aucun calcul) : 0.0676ms
4) *Readback* de la mémoire 5.266ms
5) Total : 12.7636ms

Puisque nous devons prendre au maximum 33ms pour avoir des images en temps réel, il nous reste 20.2364ms par image pour effectuer des calculs. Aussi, ajouter un *kernel* et une passe à un algorithme existant ne le ralentit que de 0.1276ms, si on omet les calculs. Même si on accélérait le *readback* de 20% (avec les cartes de prochaines générations), on aurait un temps de *readback* de 4.2128ms et il serait toujours significativement plus long que les traitements de données. Par conséquent, même une librairie comme brookGPU qui pourrait générer du code non optimal n'aurait que peu d'incidence sur les performances vues et réelles de l'algorithme. En effet, les calculs prennent toujours très peu de temps par rapport aux transferts de mémoire.

6.5 Cas hypothétique – truquage des données

Comme il a été mentionné dans l'introduction, certaines personnes rapportent leurs performances en omettant le transfert de données entre la sysRAM et la vRAM. En effet, puisqu'il n'y a pas d'équivalent de ces transferts de mémoire sur le CPU (ce ne serait pas nécessaire), les comparaisons d'algorithme omettent souvent ces détails. Si l'on considère par exemple un cas hypothétique où les calculs prendraient 1ms sur GPU et 50ms sur CPU pour une image 640x480 RGB en 32f, l'on pourrait croire que l'algorithme est 50 fois plus rapide sur GPU. Toutefois, si l'on tient compte des transferts de mémoire, nous avons 7.37ms de *download* et 5.266ms de *readback* pour un total de 12.636ms. Le temps de l'algorithme de GPU prend alors 13.636ms et n'est que 3.667 fois plus rapide que son équivalent sur CPU. Avec une image de plus grande dimension, il serait même possible que l'algorithme devienne effectivement plus lent sur GPU.

7. OBSTACLES AU GPGPU

Puisque le GPGPU est une technologie relativement jeune, plusieurs fonctionnalités ne sont pas encore implantées correctement et le programmeur fait face à des contraintes nouvelles. Nous étudierons certaines de ces importantes contraintes dans la présente section.

7.1 Difficulté de débogage

Puisque la plupart des solutions pour programmer les GPU dépendent de librairies et non de 'studios' (Visual Studio, gcc + gdb), les débogueurs actuels ne permettent pas de visualiser la mémoire vidéo. De toute façon, l'écriture dans la mémoire vidéo se fait au niveau du pilote[28] et donc serait très difficile à visualiser et à gérer.

[28] http://www.gpgpu.org/forums/viewtopic.php?t=2516&view=next&sid=4526eb72f36b95613f1b9490cb261477

Avec brookGPU, il suffit de mettre plusieurs *streams* en sortie d'un *kernel* et d'évaluer le résultat de chacun de ces *streams* pour détecter le problème. Il est toutefois impossible d'exécuter les *kernels* instruction par instruction avec brookGPU et cela peut occasionner des coûts considérables au niveau du temps de débogage.

En programmant directement sur l'API graphique (Directx9 ou OpenGL), plusieurs autres choses peuvent être problématiques : il faut garder connaissance des textures actives, de quelles textures sont associées avec quels *buffers*, des *kernels* exécutés, etc.

Bref, faute d'outils spécialisés, le débogage sur GPU est beaucoup plus long et difficile que le débogage sur CPU.

7.2 Simple précision

Tel que mentionné précédemment, le GPU calcule tout en simple précision alors que le CPU (FPU) fait ses calculs en double précision et puis tronque le résultat. Utilisons comme exemple le code suivant :

```
float pkmean = 66950.000f;
float pkstdv = 12425024.0f;
float pkone_by_size_pattern = 0.0027700830f;
float pkresult = 0.0f;
pkresult = pkstdv - pkmean*pkmean*pkone_by_size_pattern;
```

GPU	CPU

Calcul intermédiaire en simple precision:
`pkstdv - pkmean*pkmean*pkone_by_size_pattern;`
`== 12425024.0f – 12416350.0f`
`== 8674.00f`
	Calcul intermédiaire en double précision
	`pkstdv - pkmean*pkmean*pkone_by_size_pattern;`
	`== 12425024.0f – 12416350.0684f`
	`== 8673.9316f`

7.3 Scatter

L'impossibilité de faire des écritures directes avec les *kernels* de Brook et les *fragment* (ou *pixel*) *shaders* est un handicap sérieux dans la programmation pour GPGPU. Considérons par exemple la génération d'un histogramme (dans le cas de l'algorithme de Canny)

```
for(r=0;r<COLUMNS*ROWS;r++) hist[r] = 0;
for(r=0,pos=0;r<rows;r++){
   for(c=0;c<cols;c++,pos++){
      if(edge[pos] == POSSIBLE_EDGE) hist[mag[pos]]++;
   }
```

```
    }
```

Le bout de code surligné en rouge est un cas typique de *scatter*, qu'il est virtuellement impossible de faire efficacement sans utiliser les propriétés de *scatter* du *vertex shader*. Malheureusement, le *vertex shader* n'est absolument pas implémenté dans Brook.

Le problème se pose aussi lorsqu'on veut diviser une image en plusieurs couches. Par exemple, dans l'algorithme de modélisation d'arrière-plan par approche bayésienne, où les variables r et c représentent les rangées et les colonnes :

```
layerno= confs[k].rank;
m_bkgmodels[r][c].vt[layerno] ++;
m_bkgmodels[r][c].kt[layerno] ++;
```

Même si écrire à la position [r][c] peut sembler être un *scatter*, il n'en est rien, car ces variables sont incrémentées par deux boucles imbriquées qui passent tous les pixels de l'image, ce qui correspond au comportement par défaut d'un *stream*.

Le problème peut être contourné en utilisant une librairie spécialisée (CUDA ou CTM), en utilisant OpenGL directement avec les *vertex shaders* ou en changeant complètement l'algorithme.

7.4 Nombre de textures actives simultanément

Les APIs limitent le nombre de textures actives simultanément. Dans brookGPU, chaque *stream* équivaut à une texture, excepté si le *stream* en question est un *stream* de *struct*, dans lequel cas chaque champ de la *struct* équivaut à une structure. Cela peut être un problème si on veut faire un gros *kernel* ayant plusieurs *streams* en entrée et plusieurs en sortie. Par exemple, dans le cas du modèle d'arrière-plan, il était nécessaire d'avoir trois matrices 3x3 (3*9 = 27) en entrée et le même nombre en sortie (*2 = 54). Cela fait donc 54 composantes. Si on empaquette les données dans des *float3*, cela fait tout de même 18 *streams*, ce qui dépasse la limite de 16 textures actives simultanément avec la carte utilisée et les versions d'OpenGL et DirectX. Il est donc nécessaire d'écrire plus de plus petits *kernels* donc le fonctionnement peut devenir contre-intuitif. Il est à noter qu'avec brookGPU, en utilisant DirectX9, l'exécution du *kernel* donnait un message d'erreur alors qu'avec OpenGL, le seul résultat était que les données étaient erronées.

7.5 Type de données de sortie[29]

Les GPU actuels ne supportent pas différents types de données pour les textures comme cibles de rendu. Concrètement, cela signifie qu'en utilisant DirectX ou OpenGL, si on utilise les MRT (*Multiple Render Targets*), elles doivent toutes être du même type. Il est impossible actuellement d'avoir une texture de sortie de luminance et une autre de RGBA. En brookGPU, cela signifie que tous nos *streams* de sortie doivent avoir le même format : il ne peut pas y avoir un *stream* de sortie en *float4* et un autre en *float3*, par exemple.

Pour ce qui est de brookGPU, il y a deux solutions. Premièrement, on peut empaqueter toutes nos données dans des *float4*, ce qui peut rendre le code difficile à lire. Par exemple, si on a deux matrices 3x3 (2*9 données), on a 18 données et donc besoin de cinq *float4*. Soit les *streams* de *float4* Stream_a, Stream_b, Stream_c, etc. On aura alors :

Première matrice

Stream_a.x	Stream_a.y	Stream_a.z
Stream_a.w	Stream_b.x	Stream_b.y
Stream_b.z	Stream_b.w	Stream_c.x

Seconde matrice

Stream_c.y	Stream_c.z	Stream_c.w
Stream_d.x	Stream_d.y	Stream_d.z
Stream_d.w	Stream_e.x	Stream_e.y

Ce qui est évidemment difficile à manipuler, et devient encore plus difficile si Stream_e.z et Stream_e.w sont utilisées pour d'autres données.

Dans l'exemple précédent, il aurait évidemment été possible d'utiliser des *float3* qui auraient correspondu à une rangée de la matrice. Cependant, nous supposons qu'une autre donnée du problème nécessitait un *float4* pour mieux illustrer notre problème.

7.6 Différence entre les cibles de rendu (ATI vs NVIDIA)[30]

Nous avons déjà vu qu'il existe principalement deux formats de données pouvant s'inscrire dans une texture : les RGBA (*float4*) ou bien LUMINANCE (*float*). Bien que toutes ces textures existent pour tous les fournisseurs, ces derniers n'offrent pas de support pour tous les formats quand les textures sont utilisées comme cible de rendu. En d'autres termes, on peut utiliser la texture que l'on veut comme entrée mais pas celle que l'on veut comme sor-

[29] http://www.gpgpu.org/forums/viewtopic.php?t=4614
[30] http://www.mathematik.uni-dortmund.de/~goeddeke/gpgpu/tutorial.html

tie. Le principal problème vient du fait qu'ATI ne supporte pas les textures de LUMINANCE comme cible de rendu. On est donc forcé d'utiliser des *float4* comme données de sorties. Supposons que nous voulions lisser une image en prenant, pour chaque point de sortie, la moyenne du point d'entrée et des huit autres points qui lui sont contigus.

Puisque chaque pixel correspond à un *float*, nous prenons une texture de LUMINANCE en entrée et la même chose en sortie. Donc, le point à la position (1;1) en sortie sera la moyenne des points aux positions suivantes :

(0,0)	(1,0)	(2,0)
(0,1)	Index de sortie (1,1)	(2,1)
(0,2)	(1,2)	(2,2)

Par contre, si l'on est contraint d'utiliser une texture de RGBA, nos données seront groupées de façon différente. Si nous supposons que nous avons une texture bidimensionnelle contenant 36 données (une largeur de 12 par une hauteur de 3). L'équivalent en *float4* sera une texture d'une largeur de 3 (* 4 données par *float4* = 12) et d'une hauteur de 3, représentée comme suit :

x	y	z	w	x	y	z	w	x	Y	z	w
(0,0) (upleft)				(1,0) (up)				(2,0) (upright)			
0	1	2	3	4	5	6	7	8	9	10	11
(0,1) (left)				(1,1) (center)				(2,1) (right)			
12	13	14	15	16	17	18	19	20	21	22	23
(0,2) (downleft)				(1,2) (down)				(2,2) (downright)			
24	25	26	27	28	29	30	31	32	33	35	35

Supposons que nous sommes rendus au *float4* surligné en jaune, correspondant aux coordonnées (1,1) dans la texture (appelons-le *outdata*). Nous devrons alors calculer la moyenne pour chaque champ de la structure *float4* (x,y,z,w, qui correspondent ici aux carrés 16,17,18,19). Nous avons donc :

16 = 3+4+5 + 15+16+17 + 27+28+29;

Le 3 correspond au champ .w de la coordonnée (0,0) de la texture. En procédant ainsi, l'on remarque que l'on a besoin d'aller chercher en mémoire dans la texture des données dont on ne se sert pas. Puisque l'on est obligé d'aller chercher le *float4* correspondant aux coordonnées (0,0) pour utiliser le carré numéroté 3, on ira aussi chercher le carré numéroté 0

dont on ne se servira jamais. La pénalité de performance est toutefois minime. Le vrai problème réside dans l'écriture du programme.

Traduire l'équation ci-haut en utilisant les noms (upleft, up, upright, etc.) pour décrire les coordonnées de texture devient:

outdata.x = upleft.w+up.x+up.y + left.w+center.x+center.y + downleft.w+down.x+down.y;

Et notre code s'en considérablement complexifié. Il est à noter que brookGPU cache efficacement cette portion de la couche graphique et qu'il est possible d'utiliser des *float* même dans une texture de sortie, mais cela se fait parce que l'on utilise d'autres techniques, moins rapides, mais plus compatibles, pour effectuer le rendu (les *pbuffers*).

7.7 Différences entre les optimisations[31]

Si l'on utilise directement OpenGL pour communiquer avec les cartes, il existe différentes façons d'obtenir les mêmes résultats et certains blocs de code seront optimisés de différentes façons selon le fournisseur. Par exemple, tel que présenté sur le site de Dominik Göddeke, le code suivant remplit une texture et est accéléré sur les cartes NVIDIA:

```
glBindTexture(texture_target, texID);
glTexSubImage2D(texture_target,0,0,0,texSize,texSize,
                texture_format,GL_FLOAT,data);
```

Sur les cartes ATI, le code suivant, qui a exactement les mêmes résultats, est accéléré :

```
glDrawBuffer(GL_COLOR_ATTACHMENT0_EXT);
glRasterPos2i(0,0);
glDrawPixels(texSize,texSize,texture_format,GL_FLOAT,data);
```

7.8 Complexité du code

La complexité du code est une conséquence directe des problèmes énoncés précédemment. Si on programme directement avec OpenGL sur NVIDIA et ATI, il est facile d'imaginer le code devenant illisible à cause des différentes optimisations propres à chacun des fournisseurs. Même en utilisant brookGPU, si l'on est contraint à séparer notre gros *kernel* en plusieurs petits *kernels* dont le fonctionnement n'est pas clair, à limiter notre nombre de textures (*streams* dans le cas de brookGPU) actives et à s'assurer que tous les *streams* de sortie aient le même format, le code peut rapidement devenir difficile à maintenir. Ainsi, des outils de débogage limités couplés à du code difficile à lire et à maintenir

[31] http://www.mathematik.uni-dortmund.de/~goeddeke/gpgpu/tutorial.html

peuvent engendrer des temps de production considérablement supérieurs à l'équivalent sur CPU.

8. PRATIQUES DE PROGRAMMATION

À la lumière de ce que nous avons appris, il est important, pour obtenir des résultats optimaux, de respecter certaines pratiques de programmation. Ces pratiques découlent toutes de ce que nous avons mentionné antérieurement par rapport aux performances des GPU. Nous parlerons surtout en termes de *stream* ou de programmation OpenGL, puisque ce sont les deux aspects les plus étudiés dans ce document.

8.1 Quand utiliser le GPU ?

Puisqu'il y a un coût fixe certain (en temps de traitement) à utiliser un GPU, il ne sera pas efficace sur de petites quantités de données. On évalue à environ 4 000 la quantité de données minimales que l'on doit traiter pour observer un gain[32]. Les *gathers* et les passes sont très peu dispendieux et donc ne devraient pas être considérés lors de la décision de faire ou non un algorithme sur GPU. Toutefois, puisque la bande passante est relativement semblable à celle du CPU, l'aspect qui affirmera le plus la supériorité du GPU est l'**intensité arithmétique**. Plus il y a de calculs à faire sur chaque donnée, plus l'écart de vitesse entre le GPU et le CPU sera grand, à la faveur du GPU naturellement. Cela s'explique par sa nature hautement parallèle (les GPU actuels se comparent à un CPU avec 24 cœurs).

Si l'on utilise OpenGL ou brookGPU, les *scatters* sont très coûteux. Le simple fait d'avoir à écrire à un indice particulier peut engendrer des coûts déraisonnables. Par conséquent, si un algorithme nécessite de faire des *scatters*, il est nécessaire de le modifier pour contourner le problème. En utilisant CUDA ou CTM, ce problème est inexistant.

8.2 Éviter les transferts de mémoire

Comme on a pu le voir à la Section 6, les transferts de mémoire de RAM vers vRAM (*Download*) et de vRAM vers RAM (*Readback*) sont très coûteux. Ces transferts sont presque assurément le goulot d'étranglement dans un algorithme de GPU et doivent être évités à tout prix. Quand les données nécessaires ont été chargées dans la vRAM, il est essentiel de manipuler ces données le plus possible à même la mémoire vidéo avant de les re-

[32] http://www.gpgpu.org/forums/viewtopic.php?t=3172

transmettre au CPU. La vitesse des accès mémoire vidéo par le GPU est d'environ 50GB/s alors que les accès à la mémoire système se font environ à 0.5GB/s.

8.3 Écrire de gros *kernels* ?

Chaque nouveau *kernel* écrit correspond à une nouvelle passe et un coût fixe en temps de traitement. Nous avons déterminé que le coût d'ajout d'un *kernel* est d'environ 0.12ms (sans tenir compte des calculs). Par conséquent, écrire de gros *kernels* peut être plus rapide qu'écrire beaucoup de petits *kernels*. Toutefois, si ces gros *kernels* doivent faire appel à des fonctions coûteuses comme des conditionnelles (*if*), il peut s'avérer être plus rapide de faire plus de petits *kernels*. De toute façon, cette dernière pratique est préférable pour déboguer et n'engendre que des coûts marginaux au niveau de la performance.

8.4 Quand optimiser ?

Les algorithmes sur CPU se traduisent souvent mal directement sur GPU. Par conséquent, les écrire sur CPU d'abord pour ensuite avoir à les repenser entièrement peut sembler être une perte de temps. Toutefois, puisque les outils de débogage du GPU sont médiocres, il est recommandé d'avoir une version de l'algorithme fonctionnelle sur CPU. Il sera possible d'utiliser cet algorithme comme prototype et comme modèle pour déboguer, par exemple en comparant les résultats intermédiaires. Bref, il est recommandé d'écrire son application pour le CPU d'abord, s'assurer qu'elle fonctionne, et ensuite optimiser pour le GPU.

9. ÉVALUATION DE PERFORMANCE

Dans cette section, nous évaluerons différents algorithmes et leurs performances respectives sur GPU et sur CPU. Nous comparerons aussi les performances avec nos prédictions. Il est à noter que nous prévoyons que les résultats de brookGPU avec OpenGL seront médiocres puisque cette fonctionnalité de la librairie est désuète (utilisation des *pbuffers* au lieu de PBO/FBO). Aussi, tous nos résultats avec le GPU incluent le transfert de données entre la vRAM et la sysRAM.

9.1 Canny

L'algorithme est surtout intense au niveau des accès mémoire (*gather*). En effet, les calculs de gradient et de filtrage gaussien demandent surtout des accès mémoire. Les opéra-

tions mathématiques sont souvent uniquement des soustractions ou des additions. Nous nous attendons à un gain marginal de performance.

CPU (optimisation openCV): 6735ms
CPU (implémentation naïve) : 32235ms
brookGPU (OpenGL) : 14919ms
brookGPU (directX9) : 7218ms

9.2 Transformée *census*

L'algorithme est peu intense arithmétiquement. Il ne fait que faire une moyenne et une comparaison avec les *texels* environnants. Nous prévoyons un gain marginal de performance en traduisant sur GPU. Le code écrit à même OpenGL est optimisé avec les PBO et donc nous devrions avoir un gain significatif de performance par rapport à l'implémentation de brookGPU, à cause de la vitesse de transfert de mémoire accrue.

CPU : 11347ms
brookGPU (OpenGL) : 5117ms
brookGPU (directX9) : 5367ms
GPU (optimisé OpenGL) : 3923ms

9.3 Pyramide d'images

L'algorithme ne fait que copier une partie de l'image dans une mémoire tampon plus petite. Si tout l'algorithme se déroulait sur le GPU et que l'on utilisait les différents niveaux de la pyramide dans un algorithme aussi sur GPU, il pourrait y avoir des gains significatifs. Par contre, si l'on copie chaque niveau de la pyramide de la vRAM vers la sysRAM, on fait une copie de mémoire additionnelle qui n'est pas requise sur CPU et donc l'algorithme devrait être plus lent que sur CPU.

CPU : 11758ms
brookGPU (OpenGL) : 4025ms
brookGPU (directX9) : 6506ms

9.4 Discussion des résultats

Il semblerait que Brook avec OpenGL obtienne de meilleurs résultats que prévus, surtout pour les algorithmes simples répétés souvent; peut-être qu'OpenGL et le pilote de la carte communiquent mieux au niveau de la cache et de l'optimisation des résultats à garder en mémoire (élimination du code mort). Quoiqu'il en soit, quand l'algorithme devient complexe (canny), on remarque qu'OpenGL est deux fois plus lent que DirectX.

10. CONCLUSION

En conclusion, les gains que nous avons réalisés en utilisant le GPU ne sont pas si encourageants. Compte tenu de la période d'adaptation à ce nouveau modèle de programmation, la technologie semble être trop jeune encore pour répondre à nos critères de sélection. En effet, nous voulions une façon de programmer qui soit gratuite, simple (en cachant la couche graphique), portable et performante. Il semblerait qu'une telle librairie n'existe pas encore. Si la priorité est vraiment la performance, il pourrait être intéressant d'explorer CUDA ou CTM, mais ces deux API sont dépendants du fournisseur et sont très difficiles à utiliser. En effet, CUDA utilise un modèle de mémoire avec trois hiérarchies, complètement différent de celui auquel nous sommes habitués, et CTM est essentiellement un langage d'assembleur. Il se peut aussi que les algorithmes testés aient été tout simplement mal choisis pour être portés sur GPU. Il faudrait analyser plus en profondeur quel genre d'algorithme, avec échantillons de code, est optimal sur les cartes graphiques. Une fois encore, la technologie semble être trop jeune et cette documentation est inexistante.

Voici donc un tableau, très approximatif, des librairies vues et de leur réponse aux quatre critères principaux. Nous omettons délibérément Accelerator parce qu'elle nécessite un environnement de programmation en C# (langage peu utilisé en vision par ordinateur). Nous omettons aussi PeakStream car elle n'est plus disponible. Les critères sont cotés entre 1 et 5, où 1 est la plus faible note et 5 la meilleure. La facilité d'utilisation dépend de la documentation disponible et de la qualité des compilateurs.

Librairie	Coût	Performance	Facilité d'utilisation	Portabilité
brookGPU	5	2	5	4
Sh	5	2	2	4
RapidMind	3	3	4	4
CUDA	5	5	1	1
CTM	5	5	1	1
OpenGL	5	4	3	4
DirectX9	4 (nécessite Windows)	3	2	2

ANNEXE I - CALCUL DE LA TRANSFORMÉE «CENSUS» OPTIMISÉE À LA MAIN AVEC OPENGL

```
/*
 * Ce tutoriel est une combinaison des tutoriels de PBOs et de FBOs
 * utilisés pour calculer de façon optimale le census d'une image
 * Pascal Jetté, 2007

   Basé sur les tutoriels présents à cette adresse
  www.mathematik.uni-dortmund.de/~goeddeke/gpgpu/tutorial3.html
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <GL/glew.h>
#include <GL/glut.h>
#include <assert.h>
#include "common.h"



/*****************************************************
 *    macros, problem size definition etc.           *
 *****************************************************/

// PBO macro (see spec for details)
#define BUFFER_OFFSET(i) ((char *)NULL + (i))

/*****************************************************************************/
// Checks error while compiling GLSL program
void printInfoLogs(GLuint obj, GLuint shader) {
    int infologLength = 0;
    int charsWritten  = 0;
    char *infoLog;
    glGetProgramiv(obj, GL_INFO_LOG_LENGTH, &infologLength);
    if (infologLength > 1) {
        infoLog = (char *)malloc(infologLength);
        glGetProgramInfoLog(obj, infologLength, &charsWritten, infoLog);
        printf(infoLog);
        printf("\n");
        free(infoLog);
    }
    glGetShaderiv(shader, GL_INFO_LOG_LENGTH, &infologLength);
    if (infologLength > 1) {
        infoLog = (char *)malloc(infologLength);
        glGetShaderInfoLog(shader, infologLength, &charsWritten, infoLog);
        printf(infoLog);
        printf("\n");
        free(infoLog);
    }
}
```

```
/*****************************************************************************/
// General OpenGL error checking
void checkErrors(const char *label)
{
    GLenum errCode;
    const GLubyte *errStr;

    if ((errCode = glGetError()) != GL_NO_ERROR)
    {
        errStr = gluErrorString(errCode);
            printf("%s: OpenGL ERROR ",label);
            printf((char*)errStr);
        printf("\n");
            exit(-2);
    }
}

/*********************************************************/
// fragment program: Compute Census Transform
/*
    float2 ul={-1,-1};
    float2 ur={1,-1};
    float2 dl={-1,1};
    float2 dr={1,1};
    float2 r={1,0};
    float2 u={0,-1};
    float2 l={-1,0};
*/
const char* kernelSource = \
        "  uniform samplerRect image;                                    " \
        "void main(void) {                                               " \
        "float uleft = textureRect(image, (vec2(-1,-1)+ gl_TexCoord[0].st)).x; "\
        "float up = textureRect(image, (vec2(0,-1)+ gl_TexCoord[0].st)).x; "\
        "float uright = textureRect(image, (vec2(1,-1)+ gl_TexCoord[0].st)).x; "\
        "float left = textureRect(image, (vec2(-1,0)+ gl_TexCoord[0].st)).x; "\
        "float center = textureRect(image, gl_TexCoord[0].st).x; "\
        "float right = textureRect(image, (vec2(1,0)+ gl_TexCoord[0].st)).x; "\
        "float dleft = textureRect(image, (vec2(-1,1)+ gl_TexCoord[0].st)).x; "\
        "float down = textureRect(image, (vec2(0,1)+ gl_TexCoord[0].st)).x; "\
        "float dright = textureRect(image, (vec2(1,1)+ gl_TexCoord[0].st)).x; "\
        "float mean = (uleft+up+uright+left+center+right+dleft+down+dright)/9;"\
        "  gl_FragColor.x = 0;" \
        "  gl_FragColor.x += (1.0*float(dright >= mean));" \
        "  gl_FragColor.x += (2.0*float(down >= mean)); " \
        "  gl_FragColor.x += (4.0*float(dleft >= mean));" \
        "  gl_FragColor.x += (8.0*float(right >= mean));" \
        "  gl_FragColor.x += (16.0*float(center >= mean));" \
        "  gl_FragColor.x += (32.0*float(left >= mean));" \
        "  gl_FragColor.x += (64.0*float(uright >= mean)); " \
        "  gl_FragColor.x += (128.0*float(up >= mean)); " \
        "  gl_FragColor.x += (256.0*float(uleft >= mean)); }";


/*************************************************/
// main program
void compareGPU_LL(unsigned char * image,
                   int** result,
                   int& rows,
```

```
                  int& cols,
                  int argc,
                  char **argv)
{
    int i = 0;
    int iterations = 0;

    // set up glut to get valid GL context and
    // get extension entry points
    glutInit (&argc, argv);
    glutCreateWindow("STREAMING TUTORIAL");
    glewInit();
    (*result) = new int[rows*cols];

    // viewport transform for 1:1 'pixel=texel=data' mapping
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0.0,cols,0.0,rows);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glViewport(0,0,cols,rows);

    // create FBO and bind it
    GLuint fb;
    glGenFramebuffersEXT(1,&fb);
    glBindFramebufferEXT(GL_FRAMEBUFFER_EXT,fb);

    // host memory for output data
    // we will need this only later
    float* tmpfloat = new float[rows*cols];


    //create output texture
    GLuint outputTexture;
    glGenTextures(1,&outputTexture);

    glBindTexture(GL_TEXTURE_RECTANGLE_ARB,outputTexture);
    glTexParameteri(GL_TEXTURE_RECTANGLE_ARB, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_RECTANGLE_ARB, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_RECTANGLE_ARB, GL_TEXTURE_WRAP_S, GL_CLAMP);
    glTexParameteri(GL_TEXTURE_RECTANGLE_ARB, GL_TEXTURE_WRAP_T, GL_CLAMP);
    glTexImage2D(GL_TEXTURE_RECTANGLE_ARB,
                 0,
                 GL_FLOAT_R32_NV,
                 cols,
                 rows,
                 0,
                 GL_LUMINANCE,
                 GL_FLOAT,
                 tmpfloat);

    // create input texture, set parameters and allocate space. do not //populate it!
    GLuint inputTexture;
    glGenTextures(1, &inputTexture);
    glBindTexture(GL_TEXTURE_RECTANGLE_ARB,inputTexture);
    glTexParameteri(GL_TEXTURE_RECTANGLE_ARB, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_RECTANGLE_ARB, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_RECTANGLE_ARB, GL_TEXTURE_WRAP_S, GL_CLAMP);
    glTexParameteri(GL_TEXTURE_RECTANGLE_ARB, GL_TEXTURE_WRAP_T, GL_CLAMP);
```

```
glTexImage2D(GL_TEXTURE_RECTANGLE_ARB,
             0,
             GL_FLOAT_R32_NV,
             cols,
             rows,
             0,
             GL_LUMINANCE,
             GL_FLOAT,
             NULL);



//
// create buffer objects (one for download, one for readback)
//
GLuint ioBuf[2];
glGenBuffers(2, ioBuf);

//
// set up GLSL runtime
// create kernel program, load, compile and link it
//
GLuint glsl_shader = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(glsl_shader, 1, &kernelSource, NULL);
glCompileShader(glsl_shader);
GLuint glsl_program = glCreateProgram();
glAttachShader (glsl_program, glsl_shader);
glLinkProgram(glsl_program);
glUseProgram(glsl_program);

checkErrors("compile GLSL");
printInfoLogs (glsl_program, glsl_shader);
//
// assemble input parameters to kernel
//
GLenum texunits[] = {GL_TEXTURE0, GL_TEXTURE1};
GLint texParam = glGetUniformLocation(glsl_program, "image");


//attach output texture to FBO
GLenum attachmentpoints[] = { GL_COLOR_ATTACHMENT0_EXT };
glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
                          attachmentpoints[0],
                          GL_TEXTURE_RECTANGLE_ARB,
                          outputTexture,
                          0);


//
// perform computation by looping over all chunks of input data,
// streaming them in as we proceed.
//
glFinish();

for(iterations = 0; iterations < COMPARE_ITERATIONS; ++iterations)
{

    // populate input textures for current chunk and transfer
    // data to driver-controlled memory.
```

```
        glActiveTexture(GL_TEXTURE0);
        glBindTexture(GL_TEXTURE_RECTANGLE_ARB, inputTexture);

        // bind buffer object
        glBindBuffer(GL_PIXEL_UNPACK_BUFFER_ARB, ioBuf[0]);

        // invalidate this buffer object by passing NULL as data
        // Note: according to the spec at
        // http://www.opengl.org/registry/specs/ARB/vertex_buffer_object.txt
        // GL_STREAM_DRAW hints at the driver that the data store contents
        // will be specified once by the application, and used at most a
        // few times as the source of a GL (drawing) command.
        glBufferData(GL_PIXEL_UNPACK_BUFFER_ARB,rows*cols*sizeof(float), NULL,
GL_STREAM_DRAW);

        // Acquire a pointer to the first data item in this buffer object
        // by mapping it to the so-called unpack buffer target. The unpack
        // buffer refers to a location in memory that gets "unpacked"
        // from CPU (main) memory into GPU memory, hence the somewhat
        // confusing language.
        // Important: This is a pointer to a chunk of memory in what I
        // like to call "driver-controlled memory". Depending on the
        // driver, this might be PCIe/AGP memory, or ideally onboard
        // memory.
        // GL_WRITE_ONLY tells the GL that while we have control of the
        // memory, we will access it write-only. This allows for internal
        // optimisations and increases our chances to get good performance.
        // If we mapped this buffer read/write, it would almost always be
        // located in system memory, from which reading is much faster.
        void* ioMem = glMapBuffer(GL_PIXEL_UNPACK_BUFFER_ARB, GL_WRITE_ONLY);
        assert(ioMem); // we are in trouble

        // "memcpy" the double precision array into the driver memory,
        // doing the explicit conversion to single precision.
        for(i = 0; i < rows*cols; ++i)
        {
            ((float*)ioMem)[i] = (float)image[i];
        }

        // release memory, i.e. give control back to the driver
        glUnmapBuffer(GL_PIXEL_UNPACK_BUFFER_ARB);

        // Since the buffer object is still bound, this call populates our
        // texture by sourcing the buffer object. In effect, this means
        // that ideally no actual memcpy takes place (if the driver
        // decided to map the buffer in onboard memory previously), or at
        // most one memcpy inside driver-controlled memory which is much
        // faster since the previous copy of our data into driver-
        // controlled memory already resulted in laying out the data for
        // optimal performance.
        // In other words: This call is at most a real DMA transfer,
        // without any (expensive) interference by the CPU.
        glTexSubImage2D(GL_TEXTURE_RECTANGLE_ARB,
                        0,
                        0,
                        0,
                        cols,
                        rows,
                        GL_LUMINANCE,
```

```
                        GL_FLOAT,
                        BUFFER_OFFSET(0));

        // Unbind buffer object by binding it to zero.
        // This call is crucial, as doing the following computations while
        // the buffer object is still bound will result in all sorts of
        // weird side effects, eventually even delivering wrong results.
        // Refer to the specification to learn more about which
        // GL calls act differently while a buffer is bound.
        glBindBuffer(GL_PIXEL_UNPACK_BUFFER_ARB, 0);

        // assign texture as input to shader
        glUniform1i(texParam,0);

        // render viewport-sized quad to perform actual computation
        glBegin(GL_QUADS);
        glTexCoord2f(0.0, 0.0); glVertex2f(0.0, 0.0);
        glTexCoord2f(cols, 0.0); glVertex2f(cols, 0.0);
        glTexCoord2f(cols, rows); glVertex2f(cols,rows);
        glTexCoord2f(0.0, rows); glVertex2f(0.0, rows);
        glEnd();

        glFinish();

        ////////////////////////////////////////////////////////////////
        // Readback results
        glFinish();

        // Readback using PBOs is performed in a similar way as download is.
        // First, we set one texture attached to our PBO as source for the
        // readback and bind the buffer to the pixel-pack target.
        // Note: PBO 0 were used during download, hence #1
        glReadBuffer(attachmentpoints[0]);
        glBindBuffer(GL_PIXEL_PACK_BUFFER_ARB, ioBuf[1]);

        // invalidate this buffer by passing NULL
        // Note: according to the spec at
        // http://www.opengl.org/registry/specs/ARB/vertex_buffer_object.txt
        // GL_STREAM_READ hints at the driver that the data store contents
        // will be specified once by the application, and used at most a few
        // times by the application, which is exactly what we will do.
        //
        glBufferData(GL_PIXEL_PACK_BUFFER_ARB,cols*rows*sizeof(float), NULL,
GL_STREAM_READ);

        // In contrast to conventional glReadPixels(), this call returns
        // immediately and just triggers a DMA transfer into the buffer //object we
        // allocated preciously.
        glReadPixels (0, 0, cols, rows, GL_LUMINANCE, GL_FLOAT, BUFFER_OFFSET(0));

        // Readbacks are only asynchroneous on the CPU side (since the above //call
        // returns immediately). To take advantage of this, a real //application
        // would have to perform a lot of independent work on some other data
        // while the data is DMA'ed in the background.
        // In our case, there is no work to be done, so we acquire a pointer //to
        // the data by mapping the buffer to the pixel pack target. We again
        // indicate what we want to do with the pointer (accessing the data
        // read-only). Note that this call will block until the data is //available.
        void* mem = glMapBuffer(GL_PIXEL_PACK_BUFFER_ARB, GL_READ_ONLY);
```

```
        assert(mem);
        //mem contains floats of our values.
        for(i = 0; i < rows*cols; ++i)
        {
            (*result)[i] = (int)((float*)mem)[i];
        }

        // convert data back to double precision since the underlying
        //"application"
        // will continue using the data in double precision.
        // Release pointer and buffer object back to driver control, and unbind the
        // buffer object
        glUnmapBuffer(GL_PIXEL_PACK_BUFFER_ARB);
        glBindBuffer(GL_PIXEL_PACK_BUFFER_ARB, 0);

        //clean up
        glFinish();

    }//for iterations

    delete tmpfloat;

    glDeleteFramebuffersEXT (1,&fb);
    glDeleteBuffers(2, ioBuf);
    glDeleteTextures(1,&inputTexture);
    glDeleteTextures(1,&outputTexture);
}
```

ANNEXE II – TÉLÉCHARGER BROOK SUR CVS

1) ouvrir une console cygwin
2) cvs -d:pserver:anonymous@brook.cvs.sourceforge.net:/cvsroot/brook login
3) cvs -z3 -d:pserver:anonymous@brook.cvs.sourceforge.net:/cvsroot/brook co -P brook

ANNEXE III - INTERFACE BROOK AVEC VC++ (EXEMPLE)

Fichier de kernels (xxxx.br) :

```
kernel void BRblur3x3(float input[][], out float output<>, float oneNinth)
{
    float2 ul={-1,1};
    float2 ur={1,1};
    float2 ll={-1,-1};
```

```
    float2 lr={1,-1};
    float2 r={1,0};
    float2 u={0,1};
    float2 l={-1,0};
    float2 d={0,-1};

    float center = input[indexof output.xy];
    float upleft = input[ul+indexof output.xy];
    float upright = input[ur+indexof output.xy];
    float botleft = input[ll+indexof output.xy];
    float botright = input[lr+indexof output.xy];
    float right = input[r+indexof output.xy];
    float left = input[l+indexof output.xy];
    float down = input[d+indexof output.xy];
    float up = input[u+indexof output.xy];

    output = (center + upleft + upright + botleft + botright + right + left +
down + up) * oneNinth;
}
```

Fichier d'en-tête de kernels (xxxx.brh) :

```
kernel void BRblur3x3(float input[][], out float output<>, float oneNinth);
```

Compilation : les fichiers .br et .brh sont compilés par le compilateur de brook (brcc)

```
brcc.exe -o ../built/kernel_bayesian  kernel_bayesian.br
brcc.exe -o ../built/kernel_bayesian  kernel_bayesian.brh
pause
```

En-têtes à ajouter :

```
#include <brook/brook.hpp>  //brook stream definitions
#include <brook/brt.hpp>    //brook runtime

//include our compiled brook code
#include "built/xxxx.hpp"

using namespace brook;
using namespace std;
```

Création et utilisation des streams :

```
float* infloatbuffer = new float[rows*cols];
float* outfloatbuffer = new float[rows*cols];
brook::stream stream_f_a;
brook::stream stream_f_b;
stream_f_a = brook::stream::create<float>(rows, cols);
stream_f_b = brook::stream::create<float>(rows, cols);
```

```
stream_f_a.read(infloatbuffer);
BRblur3x3(stream_f_a,stream_f_b);
stream_f_b.write(outfloatbuffer);
```

Utilisation d'OpenGL ou DirectX9 : Il suffit de créer une variable d'environnement BRT_RUNTIME. Elle prend la valeur 'ogl' pour utiliser OpenGL et 'dx9' pour utiliser DirectX9. Il faut toutefois noter que l'utilisation d'OpenGL avec brookGPU peut occasionner des performances médiocres, la maintenance pour cette librairie n'étant plus assurée.