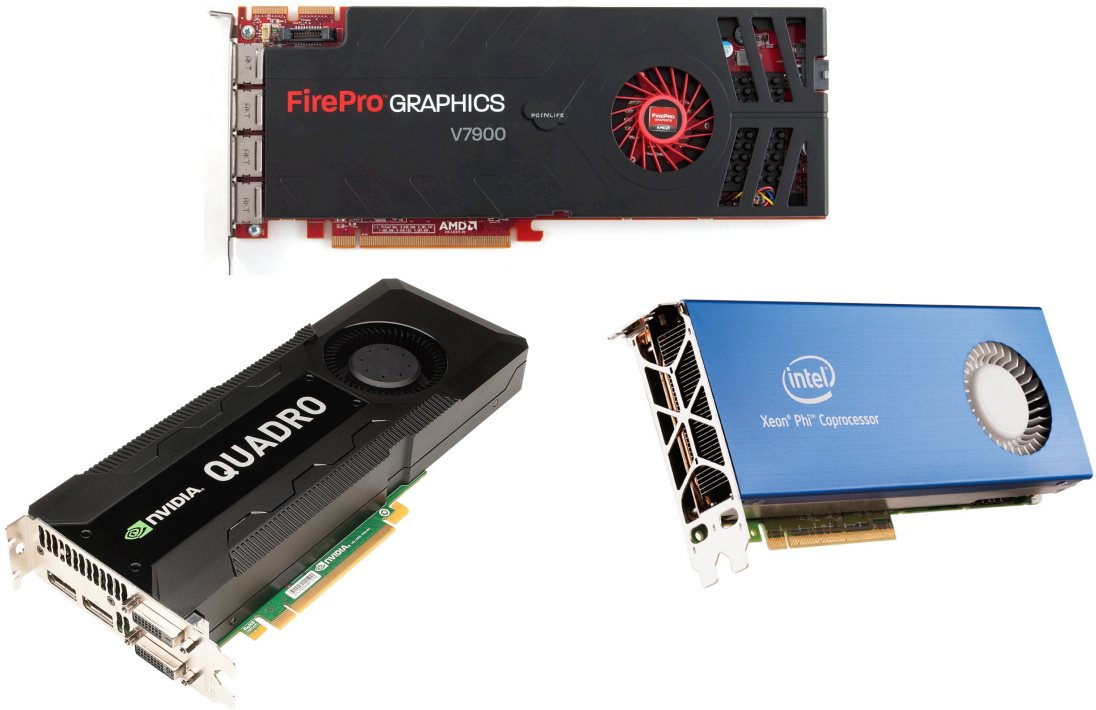
Une aide à la programmation sur GPU
pour le calcul scientifique

La librairie GMlib



Loïc MARÉCHAL / INRIA, Projet Gamma

Janvier 2014

Document v1.1

Table des matières

Couverture : différentes cartes graphiques ou accélératrices capables d'exécuter du code OpenCL.

1 Introduction

Toutes les différentes architectures rencontrées dans le milieu du HPC souffrent aujourd'hui du même problème : comment relier efficacement d'énormes quantités de mémoires d'un côté et de non moins considérables capacités de calculs de l'autre ?

Alors que, jusqu'aux années 1990, les principaux problèmes étaient d'augmenter la mémoire et la capacité de traitement, ces deux ressources sont désormais limitées par les bus ou réseaux qui les relient. Par conséquent, un système performant n'est plus celui qui offre le plus de téra-octets ou de téra-flops, mais celui qui propose la bande passante entre le(s) processeur(s) et la mémoire, la plus élevée.

Augmenter cette bande passante est rendue très complexe par la nécessité d'accès aléatoire à la mémoire. En effet, on suppose qu'un processeur ou cœur puisse accéder librement à n'importe quel emplacement en mémoire ce qui physiquement nécessite une matrice d'interconnexion (crossbar ou switch) dont la complexité croit avec le carré du nombre d'éléments connectés.

C'est pour contourner cette difficulté que les premiers calculateurs vectoriels, tels les IBM 3090 ou Cray 1, où chaque unité de calcul ne peut accéder efficacement qu'à certains emplacements mémoires. Concrètement parlant, si une machine vectorielle possède 64 additionneurs et doit réaliser l'addition de deux vecteurs `u[ 1->256 ]` et `v[ 1->256 ]`, l'unité $n°1$ ne pourra accéder qu'aux cases mémoires contenant `u[1]` , `u[65]`, `u[129]` et `u[193]`, pareillement, l'unité $n°2$ accédera rapidement aux cases `u[2]` , `u[66]`, `u[130]` et `u[194]`. Ceci impose une grande contrainte au programmeur, mais simplifie significativement l'architecture du système : elle croît en effet linéairement avec le nombre d'unités de calcul.

Les GPU actuels empruntent donc cette caractéristique à leurs lointains ancêtres, mais en l'implémentant d'une manière considérablement plus souple. En effet, la programmation sur GPU est basée sur des boucles, appelées kernels, qui sont exécutées en parallèle par les nombreuses unités de calcul. De ce fait, le portage de codes sur ces architectures tient plus de la programmation multi threads que de la vectorisation à l'ancienne. Celle-ci intervient néanmoins sous deux aspects :

-Macro : les performances maximales sont obtenues lorsque les accès mémoires se font via l'indice principal de boucle, c'est-à-dire quand le code balaye linéairement la mémoire. Ceci introduit le concept le plus important de la programmation sur GPU, à savoir, la prévisibilité des accès mémoires. Le processeur étant le plus efficace lorsqu'il peut prédire l'adresse exacte de chaque lecture et écriture en mémoire de toutes les itérations de la boucle. Ainsi, la boucle suivante est prévisible : $\forall_i\ u(i) = v(i) * w(i)$ alors que celle-ci ne l'est pas : $\forall_i\ u(i) = v(w(i))$. Bien que l'on sache que $w(i)$ va parcourir la mémoire linéairement, c'est la valeur lue qui donnera l'indice d'accès au tableau $v()$ et celle-ci est a priori inconnue, entraînant alors une grande latence mémoire. Un tel accès imprévisible est typiquement 10 fois plus lent.

-Micro : chaque unité de calcul ou cœur de ces GPU est elle-même une unité vectorielle qui traite les données sur des vecteurs courts, typiquement de quatre flottants, très utiles dans les calculs géométriques. On préférera définir un jeu de coordonnées dans l'espace

comme un vecteur de quatre flottants (`float4 coord`) plutôt que comme un tableau de quatre scalaires (`float coord[4]`). Dans le premier cas, additionner deux jeux de coordonnées ne prendra qu'une seule instruction machine traitant un super-mot de 128 bits d'un seul coup, alors qu'il faudra quatre opérations scalaires de 32 bits dans le second cas.

1.1 Motivation

Les deux principales motivations à la programmation sur GPU sont leur rapport performance/prix, bien plus intéressant que celui des CPU standards, ainsi que leur capacité d'évolution future, elle aussi bien plus grande que celle des CPU multicœurs qui semble être limitée à moyen terme.

Il faut néanmoins être réaliste, le prix d'une carte graphique haut de gamme est très élevé et souvent assez proche de celui d'un serveur de calcul. Quant aux performances annoncées par les constructeurs ou publiées dans de nombreux articles scientifiques surfant sur l'effet de mode des GPU, ils ne sont que théoriques. De nombreux codes d'exemples servant à démontrer la puissance des GPU n'utilisent que des algorithmes et des données simplifiées non représentatifs de la réalité industrielle de la simulation numérique. De plus, ils recourent à l'astuce de comparer un code sur GPU avec son équivalant séquentiel sur CPU... Les serveurs actuels possédant 12 ou 16 cœurs, la comparaison GPU / CPU multicœurs est nettement moins favorable.

Dans la pratique, le portage d'un code industriel travaillant sur des données réelles sur GPU ne saurait être plus de deux à quatre fois plus rapide qu'un bon serveur de calcul multi-cœurs. Ceci est déjà intéressant non seulement d'un point de vue prix, mais aussi d'encombrement (une carte d'extension contre plusieurs unités de racks) ou de consommation électrique.

Le gain espéré n'étant pas si extraordinaire, il est donc important pour un développeur de ne pas investir trop de temps dans une tentative de portage. Et c'est justement le problème de la programmation sur GPU : elle est très fastidieuse et consommatrice de temps !

Deux postes sont notamment lourds : la mise en forme et le transfert des structures de données de et vers le GPU, telles qu'elles soient assimilables et efficaces pour celui-ci, et le portage, l'optimisation et le débogage des algorithmes proprement dits dans le langage du GPU. Il a donc paru important de proposer aux développeurs une librairie simplifiant ces deux aspects dans le cadre d'applications traitant des maillages, données de base de la plupart des codes de simulations numériques. La librairie proposée, la *GMlib*, fourni donc au développeur des structures de données de maillages simples à définir et transférer ainsi qu'une panoplie de codes sources de base en OpenCL permettant d'y accéder efficacement.

Le mode opératoire est donc le suivant :

-choisir les types de données parmi ceux proposés qui vous permettront de stocker vos données, -partir d'un des codes de base proposés accédant à ce type de données et y insérer vos propres calculs sur ces données.

Ceci nous amène donc à parler de l'interfaçage entre vos données et la librairie, autrement dit, l'API.

1.2 API

Les types des données

L'élément de base de l'API est le DataType, celui-ci peut être prédéfini par la *GMlib* (Vertices, Edges, Triangles, Quadrialerals, Tetrahedra et Hexahedra) ou bien librement défini par l'utilisateur (RawData). L'allocation, le stockage et le transfert des données sont entièrement pris en charge, l'utilisateur ne dialoguant avec la librairie qu'à travers des requêtes. Comme un bon exemple vaut mieux qu'un long discours, voici comment initialiser une liste des coordonnées d'un maillage de 100 sommets préalablement lu en mémoire principale :

```
VerIdx = GmlNewData(GmlVertices, 100, 0, GmlInput);
for(i=0;i<100;i++)
  GmlSetVertex(VerIdx, i, coords[i][0], coords[i][1], coords[i][2]);
GmlUploadData(VerIdx);
```

La fonction `GmlNewData` permet d'allouer 100 sommets et retourne une étiquette (un simple entier) qui servira d'identificateur pour toutes les commandes qui auront à travailler sur cette entité. De fait, deux tableaux seront alloués, l'un en mémoire principale et l'autre en mémoire graphique, sans qu'aucun ne soit directement visible par le programmeur.

Le remplissage du tableau en mémoire principale se fait tout d'abord à l'aide d'une boucle sur les sommets et la commande `GmlSetVertex` qui permet de transférer les coordonnées d'un sommet de la structure interne de votre programme vers celle de la *GMlib*.

Après cette seconde étape, la librairie possède bien une copie de vos données, mais celles-ci sont toujours dans la mémoire principale, il faut encore les transférer dans la mémoire graphique, ce qui est fait d'un seul bloc grâce à la commande `GmlUploadData`. La section 2 va présenter plus en détail l'utilisation concrète de l'API.

Les kernels

L'autre entité clef est le kernel, une procédure écrite en OpenCL, qui sera exécutée par le GPU. L'intégration d'un source en OpenCL se fait selon le mode opératoire suivant :

1. transformation du fichier source mon_code.cl et en fichier header min_code.h lisible par le C grâce à l'utilitaire cl2h fourni.
2. inclusion du fichier header dans votre programme en C chargé de lancer les procédures sur le GPU par un simple `#include "mon_code.h";`
3. compilation du source à la volée au début de l'exécution de votre programme C à l'aide de la *GMlib* : `KrnIdx = GmlNewKernel(mon_code, "nom_procedure");`. Cette commande va compiler le source, envoyer le binaire exécutable sur la carte graphique et retourner une étiquette qui servira de référent pour lancer ce code par la suite.

4. lancement du kernel sur le GPU, pour travailler sur les coordonnées précédemment définies par exemple, via `LaunchKernel(KrnIdx, 100, 1, VerIdx)`. Les quatre paramètres sont dans l'ordre, l'étiquette de la procédure OpenCL à lancer, le nombre d'itérations de la boucle (ici le nombre de sommets du maillage), le nombre de DataTypes à fournir en paramètres d'entrée à cette procédure GPU, puis suivent des étiquettes de ces données, ici, seul l'identifiant du tableau de coordonnées en fourni.

La programmation en OpenCL et les aller-retour entre CPU et GPU étant assez déroutants au début, une collection de dix codes d'exemples couvrants (presque) exhaustivement tous les types de données et modes d'accès sont fournis avec la *GMlib*. La méthode la plus efficace pour faire ses premières armes sur GPU est de partir d'un de ces codes fonctionnels et de l'adapter à ses besoins.

1.3 Kernels types

Deux situations

La plupart des algorithmes bouclant sur des entités de maillages se regroupent en deux principales catégories du point de vue des écritures en mémoire :

— Les boucles à écriture directe, c'est-à-dire que toutes les données écrites accèdent aux tableaux via l'indice de la boucle principale. En cas d'exécution concurrente de plusieurs sous-parties de la boucle en simultanée, il n'y a aucun conflit d'accès à la mémoire, les sous-parties écrivant à des emplacements disjoints.

— Les boucles à écriture indirecte, c'est le cas lorsqu'on boucle sur les éléments d'un maillage, mais que l'on écrit des données relatives aux sommets des éléments de ce maillage. Lors d'une exécution concurrente, il est possible que deux éléments aient à écrire simultanément à la même adresse mémoire, car ils partagent un sommet commun. Le résultat étant alors indéterminé (bien que le résultat "Nan" soit quasi certain).

Les sections 2.3 et 2.4 illustrent ces deux types de situations.

Kernels à accès directs

Des codes complets (les parties en C et en OpenCL) sont fournis pour les cinq types d'éléments.

Chaque code propose de lire un maillage composé de sommets et d'un type d'élément, d'initialiser la *GMlib*, d'allouer et transférer le maillage sur le GPU, de charger puis lancer un kernel OpenCL bouclant sur les éléments et lisant les coordonnées des leurs sommets pour en calculer le barycentre. Le résultat est stocké dans un tableau qui est ensuite rapatrié en mémoire principale.

Les boucles sur les éléments avec accès en lecture aux sommets ou à des données qui y sont associées (champs de solutions) sont classiques en éléments finis et les cinq exemples couvrent les arêtes, triangles, quadrilatères, tétraèdres et hexaèdres.

Kernels à accès indirects

Gérer des écritures concurrentes est plus complexe et nécessite de découper le processus en deux boucles dites de "scatter" et de "gather".

Admettons que l'on veuille boucler sur des triangles, calculer trois valeurs d'une solution physique différentes pour chacun des sommets et ajouter cette valeur aux champs de solutions des sommets en question. Si on réalise cet algorithme en utilisant un kernel à accès direct comme dans la section précédente, le résultat final sera faux, car plusieurs triangles auront écrit simultanément des valeurs à un même sommet.

Pour contourner la difficulté, on va d'abord lancer un kernel simple qui va calculer les valeurs aux trois sommets, mais va les stocker dans un tableau local à chaque triangle. Ensuite, une seconde boucle, sur les sommets cette fois-ci, va additionner toutes les valeurs stockées aux triangles de sa "boule", c'est-à-dire partageant ce même sommet. Ainsi, le premier kernel boucle et écrit sur les éléments et le second boucle et écrit sur les sommets, les problèmes d'écritures simultanées sont ainsi évités.

Pareillement, les cinq codes types réalisant des écritures indirectes sur tous les types d'éléments sont fournis.

1.4 Programmation en OpenCL

Une description exhaustive ou même une simple présentation du langage OpenCL dépasserait le cadre de ce simple document technique. Le document de référence en la matière est la documentation officielle du Khronos Group [2], un regroupement de fabricants de matériels et d'éditeurs de logiciels supportant cette plate-forme ouverte.

Il est aussi intéressant de lire quelques documents parlant d'optimisation des codes en OpenCL ([3] et [5]) ou d'expérience de portage de codes préexistants sur GPU ([4]).

2 Utilisation

2.1 Installation et compilation

La librairie étant très compacte, elle se compose d'un fichier .c et de deux headers .h, le plus simple est donc d'en inclure une copie dans votre code et de la recompiler en même temps.

Vos sourcessouhai appelant la librairie devront seulement inclure "gmlib2.h" et il faudra fournir le chemin d'accès aux includes d'OpenCL sur votre système. Pour linker l'exécutable, il faudra aussi préciser le chemin de la librairie OpenCL.

Sous MacOS X il suffit de préciser "–framework=OpenCL" à la compilation pour ajouter les includes et la librairie. L'environnement OpenCL faisant partie du système depuis la version 10.6, il est inutile d'installer quoi que ce soit.

Sous Linux, il faut tout d'abord installer le kit de développement fourni par le constructeur du matériel, AMD dans le cas de cartes graphiques Radeon ou FirePro, ainsi que pour

exploiter leur CPUs (Athlon, Opteron) via OpenCL, NVIDIA pour les cartes GeForce, Quadro et Tesla et Intel pour ses CPUs. Il faut ensuite ajouter "-I /usr/local/SDK_CUDA" (si vous utilisez une carte NVIDIA) à la compilation et "-lOpenCL" comme option de link.

La situation sous Windows (à partir de XP) est identique à celle de Linux.

2.2 Initialisation

Le processus est encore un peu primitif. La fonction GmlInit ne listant qu'un seul paramètre qui peut prendre la valeur GmlCpu ou GmlGpu selon le type de matériel sur lequel vous souhaitez calculer.

Si vous précisez GmlCpu et que vous n'avez pas installé le kit de développement OpenCL d'Intel ou d'AMD, la librairie retournera une erreur et n'essayera pas d'initialiser un autre périphérique de calcul.

Si vous précisez GmlGpu, la librairie tentera d'initialiser la première carte graphique rencontrée (pas moyen d'adresser plusieurs cartes pour le moment). Si la machine ne possède pas de telle carte, c'est à vous de tenter de réinitialiser la librairie en mode CPU.

2.3 Exemple 1 : calcul d'une valeur moyenne pour chaque triangle d'un maillage

2.3.1 Problème

On dispose d'un maillage composé de NV sommets et de NT triangles et on veut calculer le barycentre de chaque triangle sur le GPU et récupérer les coordonnées de ces centres dans un tableau en mémoire principale.

Le mode opératoire est le suivant :

1. initialiser la librairie
2. allouer une donnée de type sommet
3. renseigner la structure de sommets
4. transférer ces sommets vers la mémoire graphique
5. allouer une donnée de type triangle
6. renseigner la structure de triangles
7. transférer ces triangles vers la mémoire graphique
8. allouer un type de donné libre pour stocker les barycentres
9. compiler le source du kernel de calcul des barycentres
10. lancer ce kernel de calcul en lui donnant comme paramètres les trois données allouées
11. transférer les coordonnées des barycentres dans la mémoire principale
12. récupérer chaque barycentre

2.3.2 Code CPU

On suppose que le maillage a déjà été initialisé et que les sommets et les triangles sont notés de 0 à n-1.

```
#include "CalMidKernel.h"
int triangles[nt][3];
float coords[nv][3], centres[nt][3];

GmlInit(GmlGpu);

VerIdx = GmlNewData(GmlVertices, nv, 0, GmlInput);
for(i=0;i<nv;i++)
  GmlSetVertex(VerIdx, i, coords[i][0], coords[i][1], coords[i][2]);
GmlUploadData(VerIdx);

TriIdx = GmlNewData(GmlTriangles, nt, 0, GmlInput);
for(i=0;i<nt;i++)
  GmlSetTriangle(TriIdx, i, triangles[i][0], triangles[i][1], triangles[i][2]);
GmlUploadData(TriIdx);

MidIdx = GmlNewData(GmlRawData, nt, sizeof(cl_float3), GmlOutput);
CalMid = GmlNewKernel(calmidkernel, "CalMid");
GmlLaunchKernel(CalMid, nt, 3, VerIdx, TriIdx, MidIdx);

DownloadData(MidIdx);
for(i=0;i<nt;i++)
  GetRawData(MidIdx, i, &centres[i][0], &centres[i][1], &centres[i][2]);
```

2.3.3 Code GPU

On commence par lire le numéro de triangle à traiter par cette instance du kernel. Puis on lit les indices des sommets qui forment ce triangle et sont stockés dans un vecteur de trois entiers ce qui permet de lire tous les indices d'un seul coup dans une variable locale `idx` de type `int3`. On pourra ensuite accéder à chaque indice à l'aide des suffixes `idx.s0`, `idx.s1` et `idx.s2`.

De même, toutes les coordonnées des sommets et des centres sont des vecteurs de trois flottants (`float3`) et peuvent être traitées vectoriellement. `coords[ idx.s0 ]` déclenche donc la lecture des trois coordonnées du premier sommet du triangle et l'ajout de `coords[ idx.s1 ]` réalisera l'addition des vecteurs terme à terme.

```
CalMid(float3 *coords, int3 *triangles, float3 *centres)
{
  int i = get_global_id(0);
```

```
  int3 idx = triangles[i];
  centres[i] = (coords[ idx.s0 ] + coords[ idx.s1 ] + coords[ idx.s2 ]) / 3;
}
```

2.4 Exemple 2 : boucle à accès mémoire indirect présentant des dépendances mémoires

2.4.1 Problème

On dispose d'un maillage composé de NV sommets et de NT triangles et on voudrait boucler sur les triangles tout en modifiant les coordonnées des leurs sommets, afin d'optimiser la forme du triangle par exemple. Une telle boucle présente une écriture indirecte et doit donc être découpée en deux boucles à écriture directe, l'une sur les triangles et l'autre sur les sommets.

Le mode opératoire est le suivant :

1. initialiser la librairie
2. allouer une donnée de type sommet
3. renseigner la structure de sommets
4. transférer ces sommets vers la mémoire graphique
5. allouer une donnée de type triangle
6. renseigner la structure de triangles
7. transférer ces triangles vers la mémoire graphique
8. allouer un type de donné libre pour stocker les coordonnées temporaires (tableau de scatter)
9. créer la boule des triangles incidents à chaque sommet
10. transférer ces boules vers la mémoire graphique
11. compiler le source du kernel de calcul des positions temporaires aux triangles
12. lancer ce kernel de calcul en lui donnant comme paramètres les trois données allouées
13. compiler le source du premier kernel de calcul des positions moyennes aux sommets
14. compiler le source du second kernel de calcul des positions moyennes aux sommets
15. lancer ces kernel de calculs en leur donnant comme paramètres les trois données allouées
16. transférer les coordonnées des sommets dans la mémoire principale
17. récupérer chaque coordonnée

2.4.2 Boules de sommets vectorisées

Bien que cet exemple ressemble beaucoup au premier, il y a néanmoins deux différences d'importance.

La première est la création des boules, c'est-à-dire la liste de tous les éléments d'un certain type qui partagent un même sommet. Le problème de ce genre d'information est leur taille variable, en effet, si le nombre de sommets dans un triangle est par définition constant (trois), le nombre de triangles de la boule d'un sommet lui ne l'est pas. Il est en moyenne de six, mais peut tout à fait être très élevé pour certains sommets du maillage.

Afin de stocker ce type d'information, on a en général recours à deux tables : une première concaténant les boules de tous les sommets les unes derrière les autres, et une seconde table donnant pour chaque sommet l'adresse de sa boule dans la première table. Ce système, compact et efficace sur CPU, n'est pas adapté aux GPU, car il présente une double indirection mémoire : on lit l'adresse de la boule pour lire les numéros de triangles pour lire les données associées à ces triangles. Les emplacements mémoires à lire sont donc totalement imprévisibles pour le GPU. L'idéal serait d'avoir des tailles de boules constantes afin de les coder en un seul tableau et d'éviter ainsi une des indirections. C'est le cas avec des maillages structurés de type grille, très souvent utilisés dans les exemples de codes portés sur GPU pour démontrer leur efficacité... Ce n'est bien évidemment pas le cas avec des maillages non structurés.

Néanmoins, il est possible de s'approcher des performances des grilles structurées grâce à une technique de vectorisation de boules. Celle-ci consiste à considérer que le degré des sommets est constant, par exemple huit dans notre exemple d'un maillage triangulaire. On va donc stocker directement pour chaque sommet la liste des huit triangles incidents. Si un sommet est pointé par moins de huit triangles, les dernières valeurs de la boule sont initialisées à -1. Si en revanche le degré du sommet est plus élevé, il ne tiendra donc pas dans la boule de base, est les triangles au-delà de huit seront stockés dans une seconde table de boules, dite, boules d'extensions. Cette dernière est une table classique du même type que celles utilisées sur CPU, à la différence qu'elle ne contient des entrées que pour les sommets dont le degré dépasse la taille de la boule de base (huit dans notre exemple). Seuls les triangles surnuméraires y sont stockés ce qui en réduit considérablement la taille.

La table suivante donne pour chaque type d'éléments, la taille de vecteur de boules optimale ainsi que les pourcentages de la table des boules globale contenus dans chacune des deux tables (de base et d'extension) et la mémoire supplémentaire requise par rapport à une table de boules conventionnelle.

éléments	vecteur	table de base	table d'extension	surcoût mémoire
arêtes	16	98,72 %	1,28 %	34,60 %
triangles	8	99,98 %	0,02 %	33,35 %
quadrilatères	4	99,99 %	0,01 %	0,02 %
tétraèdres	32	99,71 %	0,29 %	46,75 %
hexaèdres	8	96,87 %	3,13 %	9,79 %

2.4.3 Code CPU

On suppose que le maillage a déjà été initialisé et que les sommets et triangles sont notés de 0 à n-1.

```
#include "sources_opencl.h"
int triangles[nt][3];
float coords[nv][3], centres[nt][3];

GmlInit(GmlGpu);

VerIdx = GmlNewData(GmlVertices, nv, 0, GmlInout);
for(i=0;i<nv;i++)
  GmlSetVertex(VerIdx, i, coords[i][0], coords[i][1], coords[i][2]);
GmlUploadData(VerIdx);

TriIdx = GmlNewData(GmlTriangles, nt, 0, GmlInput);
for(i=0;i<nt;i++)
  GmlSetTriangle(TriIdx, i, triangles[i][0], triangles[i][1], triangles[i][2]);
GmlUploadData(TriIdx);

MidIdx = GmlNewData(GmlRawData, nt, sizeof(cl_float3), GmlInternal);
BalIdx = GmlNewBall(VerIdx, TriIdx);
PosTri = GmlNewKernel(sources_opencl, "scatter_tri");
VerBal1 = GmlNewKernel(sources_opencl, "gather1");
VerBal2 = GmlNewKernel(sources_opencl, "gather2");
GmlLaunchKernel(PosTri, nt, 3, VerIdx, TriIdx, MidIdx);
GmlLaunchBallKernel(VerBal1, VerBal2, BalIdx, 3, VerIdx, TriIdx, MidIdx);

DownloadData(MidVerIdx);
for(i=0;i<nv;i++)
  GmlSetVertex(VerIdx, i, &coords[i][0], &coords[i][1], &coords[i][2]);
```

2.4.4 Code GPU

Cette boucle avec dépendances en écritures qui aurait dû être retranscrite dans un kernel OpenCL, donnera donc deux, mais en fait trois kernels !

En effet, on va non seulement séparer la boucle en une paire de boucles scatter-gather pour lever la dépendance mémoire, mais cette dernière sera elle-même composée de deux boucles de gather sur les boules des sommets. L'une sur la table de base des boules vectorisées et l'autre sur le reliquat des boules d'extensions.

Voici donc le premier kernel de scatter qui boucle sur les triangles, calcule des valeurs différentes à ajouter aux coordonnées des sommets, mais les stocke dans un tableau temporaire (scatter) associé à chaque triangle.

```
scatter_tri(float3 *coords, int3 *triangles, float3 (*scatter)[3])
{
  int i = get_global_id(0);
  int3 idx = triangles[i];
  float3 v0, v1, v2;

  v0 = coords[ idx.s0 ];
  v1 = coords[ idx.s1 ];
  v2 = coords[ idx.s2 ];

  scatter[i][0] = (2*v0 + v1 + v2) / 4;
  scatter[i][1] = (2*v1 + v0 + v2) / 4;
  scatter[i][2] = (2*v2 + v1 + v0) / 4;
}
```

Le second kernel boucle sur les sommets, lit les indices des triangles incidents et va chercher leurs contributions stockées dans les tableaux de gather pour les sommer et écrire le nouveau jeu de coordonnées de chaque sommet.

```
gather1(char *degres, *boules, float3 *coords, int3 *triangles, float3 (*scatter)[3])
{
  int i, deg;
  int8 code, TriIdx, VerIdx;
  float4 NewCrd = (float4){0,0,0,0}, NulCrd = (float4){0,0,0,0};

  deg = degres[i];
  code = boules[i];
  TriIdx = code >> 3;
  VerIdx = code & (int8){7,7,7,7,7,7,7,7};

  NewCrd += (deg >  0) ? scatter[ TriIdx.s0 ][ VerIdx.s0 ] : NulCrd;
  NewCrd += (deg >  1) ? scatter[ TriIdx.s1 ][ VerIdx.s1 ] : NulCrd;
  NewCrd += (deg >  2) ? scatter[ TriIdx.s2 ][ VerIdx.s2 ] : NulCrd;
  NewCrd += (deg >  3) ? scatter[ TriIdx.s3 ][ VerIdx.s3 ] : NulCrd;
  NewCrd += (deg >  4) ? scatter[ TriIdx.s4 ][ VerIdx.s4 ] : NulCrd;
  NewCrd += (deg >  5) ? scatter[ TriIdx.s5 ][ VerIdx.s5 ] : NulCrd;
  NewCrd += (deg >  6) ? scatter[ TriIdx.s6 ][ VerIdx.s6 ] : NulCrd;
  NewCrd += (deg >  7) ? scatter[ TriIdx.s7 ][ VerIdx.s7 ] : NulCrd;

  coords[i] = NewCrd / (float)deg;
}
```

Enfin le dernier kernel,et le plus étrange, va boucler seulement sur certains sommets dont le degré trop élevé ne permettait pas à la boule d'être contenue dans la table de

base. Il va donc reprendre le calcul là où l'avait laissé la boucle précédente et ajouter les contributions des triangles restants. Attention, l'indice principal de boucle n'est pas le numéro du sommet dans la numérotation globale du maillage, mais seulement l'indice parmi les sommets de degrés élevés. Son index réel est donné par la table `infos[nv][3]`.

```
gather2(int3 *infos, int *boules, float3 *coords, float3 (*scatter)[3])
{
  int i, j, deg, VerIdx, code, BalAdr;
  float4 NewCrd;

  VerIdx = infos[i].s0;
  BalAdr = infos[i].s1;
  deg = infos[i].s2;
  NewCrd = VerCrd[ VerIdx ] * (float4){8,8,8,0};

  for(j=BalAdr; j<BalAdr + deg; j++)
  {
    code = boules[j];
    NewCrd += TriPos[ code >> 3 ][ code & 7 ];
  }

  coords[ VerIdx ] = NewCrd / (float)(8 + deg);
}
```

3 Liste des commandes

3.1 GmlListGPU

Syntaxe

```
GmlListGPU();
```

Commentaires

Affiche à l'écran la liste des matériels capables d'exécuter du code OpenCL.

Sur chaque ligne est d'abord affiché le numéro du périphérique de calcul à fournir à l'initialisation de la librairie, puis suit une brève description de celui-ci.

Exemple de sortie sur un MacBook Pro modèle 2013 :

```
0       : Intel(R) Core(TM) i7-3740QM CPU @ 2.70GHz
1       : GeForce GT 650M
2       : HD Graphics 4000
```

Le premier périphérique est le processeur principal à quatre cœurs accédant aux 16 GO de mémoire centrale, le second est la carte graphique discrète GeForce qui dispose de 384 unités de calculs et de 1 GO de mémoire dédiée et enfin le troisième est le mini-GPU intégré au processeur principal qui à 64 unités de calcul et partage 1 GO de mémoire avec celle du processeur.

3.2 GmlInit

Syntaxe

```
LibIndex = GmlInit(NuméroProcesseur);
```

Commentaires

Initialisation de la librairie nécessaire préalablement à tout lancement de commandes.

L'unique paramètre à fournir est le numéro du processeur sur lequel tous les kernels seront exécutés. La liste des processeurs disponibles est fournie par la commande `GmlListGPU`, ceux-ci pouvant êtres indifféremment des CPU, auquel cas tous les cœurs du processeur principal de la machine hôte seront utilisés, ou bien des GPU, et dans ce cas c'est la carte graphique qui exécutera les codes.

La valeur de retour est un pointeur sur une structure permettant de passer des arguments de et vers la carte graphique. La définition de cette structure est laissée à l'utilisateur qui peut y ranger tout ce qu'il souhaite pour dialoguer entre le CPU et le GPU.

3.3 GmlStop

Syntaxe

```
GmlStop();
```

Commentaires

Libère tous les types de données alloués, tous les kernels et ferme la session OpenCL.

3.4 GmlNewData

Syntaxe

```
index = GmlNewData(type, nombre, taille, accès);
```

Paramètres

Paramètre	type	description
type	int	étiquette précisant s'il s'agit d'un type de donnée prédéfini par la librairie (GmlVertices, GmlEdges, GmlTriangles, GmlQuads, GmlTetrahedra ou GmlHexahedra) ou bien d'un type librement défini par l'utilisateur (GmlRawData)
nombre	int	combien d'entités de ce type allouer
taille	int	la taille en octets de l'entité est à fournir seulement dans le cas libre (GmlRawData)
accès	int	mode de transfert possible entre les données de cette entité stockées en mémoire principale et leur image sur la carte graphique. Copie du CPU vers GPU : GmlInput. Copie du GPU vers le CPU : GmlOutput. Transfert possible dans les deux sens : GmlInout. Buffer interne au GPU ne permettant aucun transfert : GmlInternal.

Retour	type	description
index	int	étiquette de cette entité à fournir à tout kernel ayant besoin d'y accéder.

Commentaires

La fonction va allouer deux buffers de tailles identiques, l'un dans la mémoire principale et l'autre dans la mémoire graphique. Ce premier sera initialisé élément par élément via des appels successifs à la commande GmlSet, puis l'ensemble du tableau sera transféré dans son équivalent sur GPU avec un appel à UploadData.

Le procédé inverse sera utilisé pour récupérer les résultats après un calcul : DownloadData va recopier les données de mémoire graphique vers la mémoire principale. Par la suite, des appels à GmlGet permettront de lire chaque ligne du tableau.

3.5 GmlFreeData

Syntaxe

```
erreur GmlFreeData(data);
```

Commentaires

Libère l'entité d'index "data" et ses deux buffers en mémoire principale et graphique. Notez que cet index de donnée pourra être éventuellement réutilisé par un appel à NewData subséquent.

3.6 GmlSetRawData

Syntaxe

```
erreur GmlSetRawData(data, ligne, tableau);
```

Paramètres

Paramètre	type	description
data	int	étiquette de la donnée à modifier
ligne	int	numéro de la ligne à modifier
tableau	void *	pointeur sur un tableau contenant la ligne de donnée à copier dans la mémoire principale

Retour	type	description
erreur	int	0 pour un succès ou 1 sinon

Commentaires

L'idéal pour remplir un tableau de type RawData est de définir une structure contenant une seule ligne de ce type de donnée et de la réutiliser à chaque appel de GmlSetRawData comme l'illustre l'exemple ci-dessous :

```
struct
{
  int type;
  float quality;
  float normal[3];
}my_data;

for(i=0;i<NmbTriangles;i++)
{
  my_data.type = 1;
  my_data.qualite = qualities[i];
  my_data.normal[0] = normals[i][0];
  my_data.normal[1] = normals[i][1];
  my_data.normal[2] = normals[i][2];
  GmlSetRawData(IdxData, i, &my_data, sizeof(my_data));
}
```

3.7 GmlGetRawData

Syntaxe

```
erreur GmlGetRawData(data, ligne, tableau);
```

Paramètres

Paramètre	type	description
data	int	étiquette de la donnée à récupérer
ligne	int	numéro de la ligne à récupérer
tableau	void *	pointeur sur un tableau dans lequel la ligne de donnée de la mémoire principale sera copiée

Retour	type	description
erreur	int	0 pour un succès ou 1 sinon

Commentaires

Voir GmlSetRawData ci-dessus.

3.8 GmlSetVertex

Syntaxe

```
erreur GmlSetVertex(data, ligne, x, y, z);
```

Paramètres

Paramètre	type	description
data	int	étiquette de l'entité de type vertex à modifier
ligne	int	numéro du vertex à modifier
x,y,z	float	les coordonnées du vertex

Retour	type	description
erreur	int	0 pour un succès ou 1 sinon

Commentaires

Attention, les coordonnées ne sont stockées qu'en simple précision pour l'instant.

3.9 GmlGetVertex

Syntaxe

```
erreur GmlGetVertex(data, ligne, px, py, pz);
```

Paramètres

Paramètre	type	description
data	int	étiquette de l'entité de type vertex à récupérer
ligne	int	numéro du vertex à récupérer
px,py,pz	float*	pointeurs sur trois flottants qui recevront les coordonnées du vertex

Retour	type	description
erreur	int	0 pour un succès ou 1 sinon

Commentaires

Faire attention à fournir des pointeurs sur des flottants et non des doubles qui seront supportés dans une version ultérieure de la librairie.

3.10 GmlSetEdge

Syntaxe

```
erreur GmlSetEdge(data, ligne, i1, i2);
```

Paramètres

Paramètre	type	description
data	int	étiquette de l'entité de type arête à modifier
ligne	int	numéro de l'arête à modifier
i1,i2	int	numéros des deux sommets de l'arête

Retour	type	description
erreur	int	0 pour un succès ou 1 sinon

Commentaires

Ne pas oublier que les indices des sommets vont de 0 à n-1 et non de 1 à n.

3.11 GmlSetTriangle

Syntaxe

```
erreur GmlSetTriangle(data, ligne, i1, i2, i3);
```

Paramètres

Paramètre	type	description
data	int	étiquette de l'entité de type triangle à modifier
ligne	int	numéro du triangle à modifier
i1,i2,i3	int	numéros des trois sommets du triangle

Retour	type	description
erreur	int	0 pour un succès ou 1 sinon

Commentaires

Ne pas oublier que les indices des sommets vont de 0 à n-1 et non de 1 à n.

3.12 GmlSetQuadrilateral

Syntaxe

```
erreur GmlSetQuadrilateral(data, ligne, i1, i2, i3, i4);
```

Paramètres

Paramètre	type	description
data	int	étiquette de l'entité de type quadrilatère à modifier
ligne	int	numéro du quadrilatère à modifier
i1,...,i4	int	numéros des quatre sommets du quadrilatère

Retour	type	description
erreur	int	0 pour un succès ou 1 sinon

Commentaires

Ne pas oublier que les indices des sommets vont de 0 à n-1 et non de 1 à n.

3.13 GmlSetTetrahedron

Syntaxe

```
erreur GmlSetTetrahedron(data, ligne, i1, i2, i3, i4);
```

Paramètres

Paramètre	type	description
data	int	étiquette de l'entité de type tétraèdre à modifier
ligne	int	numéro du tétraèdre à modifier
i1,...,i4	int	numéros des quatre sommets du tétraèdre

Retour	type	description
erreur	int	0 pour un succès ou 1 sinon

Commentaires

Ne pas oublier que les indices des sommets vont de 0 à n-1 et non de 1 à n.

3.14 GmlSetHexahedron

Syntaxe

```
erreur GmlSetHexahedron(data, ligne, i1, i2, i3, i4, i5, i6, i7, i8);
```

Paramètres

Paramètre	type	description
data	int	étiquette de l'entité de type hexaèdre à modifier
ligne	int	numéro de l' hexaèdre à modifier
i1,...,i8	int	numéros des huit sommets de l'hexaèdre

Retour	type	description
erreur	int	0 pour un succès ou 1 sinon

Commentaires

Ne pas oublier que les indices des sommets vont de 0 à n-1 et non de 1 à n.

3.15 GmlUploadData

Syntaxe

```
erreur GmlUploadData(data);
```

Commentaires

Déclenche la recopie des données de cette entité de la mémoire principale vers la mémoire graphique. Cette opération est extrêmement lente, car le bus de la carte graphique est au moins 10 fois plus lent que la mémoire. Des débits de 2 à 6 GO/s sont à attendre alors que la mémoire graphique est capable de 50 à 200 GO/s. C'est pourquoi il faut limiter ces transferts au strict minimum, car ils peuvent réduire à néant tous les gains de temps apportés par le GPU.

3.16 GmlDownloadData

Syntaxe

```
erreur GmlDownloadData(data);
```

Commentaires

Voir ci-dessus.

3.17 GmlNewBall

Syntaxe

```
boule GmlNewBall(sommets, éléments);
```

Paramètres

Paramètre	type	description
sommets	int	étiquette d'une entité de type sommet dont on souhaite créer la boule
éléments	int	étiquette d'une entité d'un type d'éléments incidents aux sommets

Retour	type	description
boule	int	étiquette d'un nouveau type de boule

Commentaires

Création automatique d'une boule de sommets vectorielle.

Deux tableaux de boules seront crée : un premier vectoriel contenant un nombre fixe d'éléments incidents pour chaque sommet, et un second, beaucoup plus petit, donnant les éléments qui n'ont pu tenir dans la première table faute de place.

Le nombre d'éléments donnés pour chaque sommet dans le premier tableau, appelé vecteur de boule, est déterminé automatiquement en fonction du degré moyen des sommets. Il vaut typiquement 4 dans le cas des quadrilatères, 8 pour les triangles et hexaèdres, 16 pour les arêtes et 32 pour les tétraèdres.

Un type de donnée "boule" encapsule en fait quatre autres données :

— degrés[iVertex] : un tableau contenant le degré partiel pour chaque sommet, c'est-à-dire borné par la taille du vecteur de boule

— boules_base[iVertex][iElement] : tableau donnant les premiers éléments incidents pour chaque sommet

— infos[xVertex][3] : tableau donnant pour chaque extra sommet (uniquement ceux dont le degré dépasse la taille du vecteur de boule), l'indice du sommet dans la numérotation globale, la position de ces extra éléments dans le tableau d'extension de boule et le nombre de ces éléments (extra degré)

— boules_ext[xElement] : tableau donnant un élément d'une boule d'extension

3.18 GmlFreeBall

Syntaxe

```
erreur GmlFreeBall(boule);
```

Commentaires

Libère une donnée de type boule ainsi que ses quatre sous tableaux.

3.19 GmlUploadBall

Syntaxe

```
erreur GmlUploadBall(boule);
```

Commentaires

Transfère vers la carte graphique des quatre sous tableaux d'une boule de sommets vectorisée.

3.20 GmlNewKernel

Syntaxe

```
kernel GmlNewKernel(source, procédure);
```

Paramètres

Paramètre	type	description
source	char *	pointeur sur une chaîne de caractères contenant le source OpenCL à compiler
procédure	char *	pointeur sur une chaîne de caractères contenant le nom de la procédure particulière à compiler à l'intérieur du code source qui peut en contenir plusieurs

Retour	type	description
kernel	int	étiquette du kernel compilé

Commentaires

L'utilitaire *cl2h* est fourni avec la librairie afin de faciliter l'insertion des sources OpenCL dans le code. Cette commande transforme un source *toto.cl* en un fichier header du C *toto.h* qui contient une unique chaîne de caractères initialisée avec la totalité sur source OpenCL.

3.21 GmlLaunchKernel

Syntaxe

```
temps GmlLaunchKernel(kernel, lignes, arguments, ...);
```

Paramètres

Paramètre	type	description
kernel	int	étiquette du kernel OpenCL à lancer sur le GPU
lignes	int	nombre de lignes de la boucle principale du kernel, c'est la valeur maximale retournée par `get_global_index(0)` du côté OpenCL
arguments	int	nombre de données à passer en paramètres au kernel
...	int	liste d'étiquettes des données séparées par des virgules

Retour	type	description
temps	double	temps d'exécution du kernel en secondes ou un code d'erreur si la valeur est négative

Commentaires

L'ordre dans lequel les paramètres seront vus par le kernel est exactement celui des étiquettes données en arguments. De plus, deux arguments seront automatiquement ajoutés à la fin de la liste : la structure GmlParametres et un entier contenant le nombre de lignes sur lequel effectuer la boucle.

3.22 GmlLaunchBallKernel

Syntaxe

```
temps GmlLaunchBallKernel(kernel1, kernel2, boule, arguments, ...);
```

Paramètres

Paramètre	type	description
kernel1	int	étiquette du premier kernel dit "base scatter" à lancer sur le GPU
kernel2	int	étiquette du second kernel dit "extentions gather" à lancer sur le GPU
boule	int	étiquette d'une entité de type boule sur laquelle sera effectuée la paire de kernel base-extension
arguments	int	nombre de données à passer en paramètres aux deux kernels
...	int	liste d'étiquettes des données séparées par des virgules

Retour	type	description
temps	double	temps d'exécution du kernel en secondes ou un code d'erreur si la valeur est négative

Commentaires

Une donnée de type boule ne peut être passée qu'en premier paramètre de donnée de la commande GmlLaunchBallKernel et ne peut figurer dans la liste des paramètres supplémentaires. De même, une boule ne peut être passée à un simple GmlLaunchKernel.

3.23 GmlReduceVector

Syntaxe

```
temps GmlReduceVector(data, opération, résidu);
```

Paramètres

Paramètre	type	description
data	int	étiquette d'une donnée de type RawData contenant un simple flottant par ligne
opération	int	GmlMin = calcul de la valeur minimale du tableau, GmlSum = somme globale, GmlMax = valeur maximale
résidu	double*	pointeur sur un double qui recevra la valeur finale du résidu

Retour	type	description
temps	double	temps d'exécution du kernel en secondes ou un code d'erreur si la valeur est négative

Commentaires

Si les données que vous souhaitez réduire sont plus complexes qu'un vecteur (tableau bidimensionnel ou tableau de structures), il faut alors passer par une étape intermédiaire. Créez un kernel qui lit votre donnée complexe, applique une fonction de normalisation et stocke le résultat dans un vecteur simple. Vous pouvez ensuite réduire ce dernier à l'aide de la fonction GmlReduceVector.

Exemple

On veut calculer le résidu d'un tableau de vecteurs tridimensionnels ($tab[n][3]$) comme étant la plus petite norme des n vecteurs. On va donc allouer un nouveau tableau $res[n]$ de type libre, lancer un kernel bouclant sur n qui va lire chaque vecteur $tab[n]$, calculer sa norme et l'écrire dans $res[n]$. Ensuite, on appellera GmlReduceVector sur ce tableau $res[n]$ avec l'opération GmlMin.

Partie du code en C :

```
VecIdx = NewData(GmlRawData, n, 3*sizeof(float), GmlInternal);
ResIdx = NewData(GmlRawData, n, sizeof(float), GmlOutput);
LaunchKernel(CalculLongueures, n, 2, VecIdx, ResIdx);
GmlReduceVector(ResIdx, GmlMin, &longueur_min);
```

Partie du code en OpenCL :

```
__kernel void CalculLongueures(__global float3 *vec, __global float *res)
{
  int i = get_global_id(0);
  res[i] = length(vec[i]);
}
```

3.24 GmlGetMemoryUsage

Syntaxe

```
taille GmlGetMemoryUsage();
```

Commentaires

Retourne simplement la taille en octets de la mémoire graphique totale allouée sur le GPU.

3.25 GmlGetMemoryTransfer

Syntaxe

```
taille GmlGetMemoryTransfer();
```

Commentaires

Retourne le nombre d'octets qui ont été transférés de, ou, vers la mémoire graphique depuis l'initialisation de la *GMlib*.

3.26 GmlGetParameters

Syntaxe

```
paramètres GmlGetParameters();
```

Commentaires

Retourne le pointeur sur la structure de paramètres de l'utilisateur (celle retournée à l'ouverture le la librairie).

4 Glossaire

4.1 API

Application programming interface, ou interface de programmation, c'est la liste des arguments et leur format à fournir afin d'appeler une procédure d'un programme ou librairie.

4.2 Compute Unit

Une puce GPU contient un grand nombre d'unités de calcul fonctionnant en parallèle de la même manière qu'un CPU possède plusieurs cœurs. Les constructeurs jouent un peu sur les mots en comptant chaque unité vectorielle capable de traiter quatre scalaires à la fois comme étant quatre unités de calculs distinctes, ce qui n'est en pratique pas tout à fait le cas. À ce compte, un CPU comme l'Intel core i7-3770 qui possède quatre cœurs, chacun possédant une unité vectorielle capable de traiter huit flottants simultanément, serait considéré comme une puce à 32 unités de calculs selon cette terminologie.

Il est à noter que les unités de GPU sont beaucoup plus simples que celles des CPU et leur puissance de calcul par gigahertz est typiquement trois fois plus faible. Une comparaison du nombre de cœurs-gigahertz entre les deux architectures n'est pas non plus réaliste.

4.3 GPU

Graphic Processing Unit, aussi appelé GPGPU (pour General Purpose), c'est une puce adaptée aux calculs géométriques et vectoriels qui, comme sont nom ne l'indique pas, est tout sauf "general purpose" ! Ces puces se trouvent dans les cartes graphiques des deux constructeurs bien connus, AMD-ATI Radeon et NVIDIA GeForce, mais il y aussi des GPU dans la plupart de puces pour smartphones et tablettes (ARM Mali et Imagination Technologies SGX) et les consoles des jeux (IBM Cell).

4.4 Kernel

Nom donné à une boucle exécutée par un GPU. Celui-ci gère lui-même l'indice de boucle qu'il distribue à ses unités de calculs, l'utilisateur ne fournissant que le noyau de calcul à l'intérieur de la boucle, d'où le nom kernel.

4.5 OpenCL

Open Compute Langage, c'est un langage dérivé du C, avec quelques emprunts au C++, et qui a été conçu pour être indépendant de toute architecture matérielle. Il peut donc être exécuté efficacement sur des GPU, des CPU multicœurs, des FPGA ou tout

type de matériel calculant de manière concurrente. Sont objectif est d'exposer clairement l'indépendance entre les calculs afin de mieux les répartir sur les différentes unités.

4.6 Workgroup

C'est une portion de boucle qui est exécutée simultanément sur les différentes unités de calculs du GPU. C'est pourquoi on ne peut jamais supposer que l'itération i d'une boucle sera exécutée avant l'itération $i + 1$, car elles peuvent tout à fait être traitées en même temps par deux unités différentes. Ces tailles vont de 16 pour les smartphones jusqu'à 1024 pour les cartes dédiées au calcul GPU.

Références

[1] SAGAN, Space-Filling Curves, *Springer Verlag, New York, 1994.*

[2] Aaftab Munshi, The OpenCL Specification, *Khronos Group, http ://www.khronos.org/opencl/, 2012.*

[3] NVIDIA Corporation, OpenCL Optimization, *NVIDIA Corporation, NVIDIA_GPU_Computing_Webinars_Best_Practises_For_OpenCL_Programming.pdf, 2009.*

[4] Andrew Corrigan & Rainald Löhner, Porting of FEFLO to Multi-GPU Clusters, *49th AIAA Aerospace Sciences Meeting, Orlando, January 2011.*

[5] Apple Corp, OpenCL Programming Guide for Mac, *http ://developer.apple.com, 2012.*