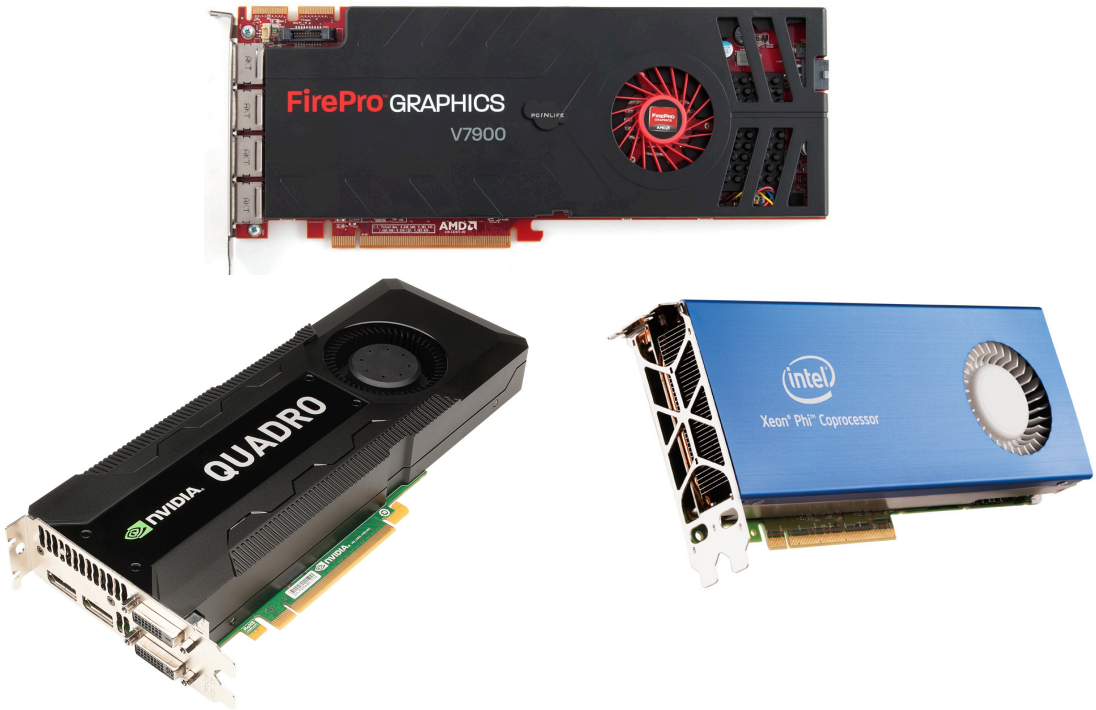
Une aide à la programmation sur GPU
pour le calcul scientifique

La librairie GMlib



Loïc MARÉCHAL / INRIA, Projet Gamma

Mai 2020

Document v2.05 Librairie v3.21

Table des matières

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

1 Introduction

La `GMlib` v3 est une librairie visant à grandement faciliter le portage ou le développement sur GPU de logiciels de calculs scientifiques utilisant des maillages non structurés comme principale donnée.

Elle est basée sur le standard ouvert OpenCL, qui a pour avantage de fonctionner sur la plupart des matériels et systèmes actuels. OpenCL fonctionne non seulement sur les GPU des trois principaux constructeurs, AMD, Intel et Nvidia, mais aussi sur les GPU intégrés des smartphones et tablettes (ARM, Imagination Technologies, etc.) et en l'absence de GPU dans un appareil, il est parfaitement à même d'utiliser efficacement tous les cœurs et capacités vectorielles des CPU, ce qui en fait une technologie très générique.

Mais le développement d'applications performantes et industrielles, c'est-à-dire, autre chose que la résolution d'un laplacien sur une grille structurée et carrée, requiert un investissement et des compétences conséquentes de la part du programmeur. Il est donc important de préciser qu'en matière de programmation sur GPU, c'est le stockage, le transfert et l'accès efficace aux données qui sont le plus problématiques et non les algorithmes, qui ne nécessitent pas tant d'adaptation par rapport aux plateformes CPU parallèles conventionnelles.

C'est pourquoi l'idée sous-jacente de la `GMlib` est d'abstraire autant que possible la gestion des données et de laisser le programmeur se concentrer le plus librement possible sur la partie algorithmique proprement dite. Les GPU étant à l'origine conçus pour l'affichage graphique, il est naturel d'en utiliser le paradigme de programmation proposé par les principaux acteurs, connus sous le terme de *shaders*.

L'approche du graphisme sur GPU consiste à demander au programmeur de transmettre l'ensemble des données nécessaires à l'affichage à la librairie graphique (OpenGL, Direct 3D ou Vulkan), puis de fournir un code assez succinct, appelé shader, à appliquer sur chacune des ces données graphiques. Il est par exemple courant de fournir un ensemble de sommets, de vecteurs normaux associés et de triangles, et par la suite, de demander à la librairie graphique de boucler sur les triangles tout en accédant aux coordonnées et normales de leurs sommets afin d'exécuter un code (shader) calculant la couleur, l'ombrage et la position à l'écran de chaque pixel du triangle.

Une telle approche a été reprise par la `GMlib`, mais ici les textures, projections, et effets graphiques sont remplacés par des tétraèdres, arêtes, champs de solutions physiques, et tous les accès indirects possibles entre ces entités. Les types de données de maillage actuellement proposées sont : sommets, arêtes, triangles, quadrilatères, tétraèdres, pyramides, prismes et hexaèdres ainsi que les liens implicites qui les relient entre eux (boules, coquilles, voisins) et qui sont automatiquement construits par la librairie.

De plus, des types de données librement définissables (entier, flottant, scalaires, tableaux, vecteur) peuvent être associés à chaque entité de maillage afin d'y stocker les données nécessaires aux calculs.

Un code basé sur la `GMlib` se déroule en quatre étapes : initialisations, renseignements des données, compilation des *kernels* (nom donné aux *shaders* dans le monde de la simulation numérique) et exécution de ces derniers sur GPU.

Initialisation : on doit tout d'abord initialiser la librairie en lui fournissant le numéro du matériel qui va exécuter les kernels en OpenCL. Puis on dimensionne chaque type de donnée de maillage et de solution associée simplement en fournissant le nombre de chaque entité et la taille des données.

Renseignement des données : l'utilisateur doit boucler sur chaque sommet, élément, ou autres données précédemment déclarés pour en transmettre le contenu à la `GMlib`, qui elle-même se chargera de les transmettre sur la carte graphique au bon moment et sous une forme adéquate.

Compilation des kernels : l'utilisateur décrit les boucles qu'il souhaite exécuter sur GPU en précisant pour chacune, la liste des entités de maillage et solutions qui lui seront nécessaires, ainsi que leur mode d'accès, lecture et/ou écriture, et enfin le code source du kernel à exécuter sur ces données et qui sera au préalable compilé par la librairie. Cette compilation à chaud permet de générer un code binaire exécutable au moment même de l'exécution du logiciel et donc de fonctionner sur n'importe quel type de matériel (AMD, Intel, NVIDIA, etc.)

Exécution des kernels : une fois toutes les données définies et transmises et les codes compilés et optimisés sur l'architecture choisie, il ne reste plus qu'a déclencher l'exécution de chaque kernel, autant de fois que nécessaire pour converger le calcul et en ne modifiant que quelques paramètres globaux afin de contrôler le processus numérique (coefficients, résidus, flags, etc.)

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 équivalent 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`, fournit 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 Installation et compilation

1.3 Initialisation

2 Utilisation

2.1 API

L'interaction avec la librairie repose sur deux éléments fondamentaux : les types de données et les codes (*i. e.* kernels) qui vont opérer sur celles-ci. Par conséquent, le déroulement d'un logiciel utilisant un accélérateur GPU va se dérouler en deux phases :

— l'allocation et la définition des données, maillages, champs de solutions et liens topologiques,

— la déclaration, la compilation et l'exécution des kernels en OpenCL.

Ces deux phases sont exécutées par le CPU hôte et seule les kernels tournent sur le GPU proprement dit et s'en trouveront donc accélérées. Dire d'un code qu'il "tourne sur GPU" est abus de langage, car les entrées-sorties, initialisations, compilations et post-traitements sont exécutés sur CPU et l'on devrait plutôt parler de logiciel hybride CPU/GPU ou bien accélérés par GPU. Il est donc important d'optimiser toutes les phases initiales sur CPU, car celle-ci représente le *coefficient d'Amdahl* de tout code hybride CPU/GPU.

L'interfaçage avec la `Gmlib` commence donc par son initialisation en lui fournissant le numéro du périphérique de calcul sur lequel tous les kernels seront exécutés : `LibIdx = GmlInit(2);`

Sur la machine de l'auteur de ces lignes, une telle commande initialise la librairie avec la carte graphique *ATI Radeon Pro 460* et l'étiquette retournée servira à communiquer avec toutes les autres commandes de la librairie par la suite. Notez que plusieurs instances de la librairie peuvent être ouvertes simultanément, avec le même périphérique, ou bien un autre.

Remarque : une instance ne peut utiliser qu'un seul périphérique de calcul, composé lui-même de plusieurs *compute units* et une seule zone mémoire, il s'agit donc de parallélisme à mémoire partagée.

Si l'utilisateur souhaite combiner la puissance de plusieurs GPU, il devra implémenter la distribution des calculs entre les cartes lui-même.

Il faut ensuite définir et renseigner les types de données qui seront utilisés par les calculs effectués sur le GPU. Cela introduit un aspect très important : l'utilisateur ne peut pas utiliser ses propres données en mémoire CPU (tableaux, structures, objets, etc.), mais doit obligatoirement les déclarer et les transférer à la librairie. C'est une contrainte très forte, car la nature des données offertes est assez limitée et le portage d'un code sur GPU va surtout consister à plier ses propres structures de données dans le canevas proposé par la `GMlib`.

Pour illustrer le processus, on va déclarer et renseigner les sommets d'un maillage que l'on vient de lire, dans le tableau `crd[n][3]` et assigner à chacun une vitesse et un vecteur de déplacement aléatoire.

```
VerIdx = GmlNewMeshData(LibIdx, GmlVertices, NmbVer);
for(i=0;i<NmbVer;i++)
   GmlSetDataLine(LibIdx, VerIdx, i, crd[i][0], crd[i][1], crd[i][2], 0);

VitIdx = GmlNewSolutionData(LibIdx, GmlVertices, 1, GmlFlt, "Vitesse");
for(i=0;i<NmbVer;i++)
{
   vitesse = rand();
   GmlSetDataLine(LibIdx, MasIdx, i, &vitesse);
}

DirIdx = GmlNewSolutionData(LibIdx, GmlVertices, 1, GmlFlt4, "Direction");
for(i=0;i<NmbVer;i++)
{
   vecteur[0] = rand();
   vecteur[1] = rand();
   vecteur[2] = rand();
   vecteur[3] = 0;
   GmlSetDataLine(LibIdx, MasIdx, i, &vecteur);
```

```
}
```

On souhaite calculer sur GPU le déplacement itératif de ces sommets selon leur direction et vitesse puis récupérer les coordonnées résultantes après 1000 itérations.

Pour cela, nous allons tout d'abord écrire le code de déplacement d'une itération en OpenCL : il s'agit d'un kernel et il doit être placé dans un fichier indépendant, que nous appellerons *iteration.cl*.

Puis on va le compiler en lui passant en paramètres les données préalablement définies : les sommets qu'on souhaite envoyer sur le GPU et récupérer à la fin (*Read & Write*) et la vitesse et le déplacement qu'on souhaite seulement envoyer, mais pas récupérer (*Read*).

```
KrnIdx = GmlCompileKernel( LibIdx,  iteration, "iteration", GmlVertices, 3,
                           VerIdx, GmlReadMode | GmlWriteMode, NULL,
                           VitIdx, GmlReadMode               , NULL,
                           DirIdx, GmlReadMode               , NULL );
```

Et voici le fameux kernel nommé *iteration.cl* :

```
VerCrd = VerCrd + VerVitesse * VerDirection;
```

Ce code, on ne peut plus succinct, amène de nombreuses remarques :
- il n'y a pas de boucle itérant de 1 à NmbVer,
- il ne comporte aucune définition préalable des variables utilisées,
- il n'y a pas d'accès aux tableaux des sommets et données associées, mais seulement des scalaires préremplis,
- on ne sait pas comment sera stocké le résultat,
- les coordonnées et le champ de déplacement sont des vecteurs en 3-D.

Boucle implicite : il y a bien un compteur de boucle qui est incrémenté de 1 à NmbVer, mais la boucle est effectué par le GPU et l'utilisateur n'a pas moyen de contrôler l'ordre de traitement des sommets. Il ne fournit que le noyau de la boucle (d'où le nom *kernel*), sans itérateur, et ce morceau de code sera instancié et exécuté autant de fois qu'il y a de sommets dans le maillage, mais dans un ordre non spécifié.

Variables implicites : les déclarations de variables locales sont effectuées par la librairie en fonction des paramètres définis lors de la compilation du kernel. Comme nous avons demandé l'accès aux coordonnées des sommets (un vecteur 3-D), à la vitesse (un scalaire) et au vecteur de déplacement (un vecteur 3-D), la librairie a donc défini d'elle-même les variables du type adéquat pour recevoir toutes les données demandées associées à l'unique sommet en cours de traitement.

Les noms des données de type maillage sont imposés par la librairie et connus (VerCrd pour les coordonnées, TriVer pour les numéros des trois sommets d'un triangle, etc.), quant aux données librement définies par l'utilisateur, elles portent le nom qu'il leur a assigné au moment de l'initialisation.

Dans notre exemple, `vitesse` est un scalaire et `direction` est un vecteur. Le code OpenCL des définitions est en fait créé par la librairie et ajouté juste avant le kernel fourni par l'utilisateur.

Accès aux tableaux globaux : il n'y a pas qu'un seul sommet dans le maillage et pourtant la librairie n'a défini qu'un seul vecteur `VerCrd` contentant les coordonnées d'un seul sommet. Là encore, la librairie a bien défini le tableau des coordonnées (`VerCrdTab[ NmbVer ]`), mais celui-ci est caché à l'utilisateur et les coordonnées du sommet en cours de traitement sont automatiquement lues avant de passer la main au kernel.

De fait, lorsque l'unique ligne du kernel d'exemple commence à être exécutée, la librairie a déjà rempli les variables locales avec les données du sommet courant i : `VerCrd = VerCrdTab[i]`, `vitesse = vitesseTab[i]`, `direction = directionTab[i]` et il n'a pas à se soucier du transfert des variables globales vers les locales sur lesquelles les calculs seront effectués.

Stockage du résultat en mémoire : de la même manière qu'avec les lectures, toutes les variables étant définies comme *Write* au moment de la compilation seront stockées dans les tableaux globaux après l'exécution du kernel utilisateur. Cette écriture est cachée et transparente. Dans l'exemple, seules les coordonnées sont accédées en écriture et la librairie rajoutera d'elle-même la ligne suivante : `VerCrdTab[i] = VerCrd` afin de stocker le résultat du calcul sur ce sommet dans la mémoire du GPU, d'éventuels calculs intermédiaires seront perdus !

Calcul vectoriel : le langage OpenCL offre des variantes scalaires et vectorielles de tailles 2, 4, 8 ou 16 pour chaque type de données entières (codées sur 8, 16, 32 ou 64 bits) et réelles (16, 32 ou 64 bits) et utilise la technique du surtypage d'opérateur pour adapter les opérations aux types des opérandes.

Dans notre kernel, `VerCrd` et `direction` sont des vecteurs 3-D et effectuer l'opération `VerCrd = VerCrd + direction` va déclencher une addition vectorielle traitant les trois coordonnées simultanément.

Que se passe-t-il alors lorsqu'on multiplie le déplacement par le scalaire contenant la vitesse ?

Le compilateur détecte la différence de type et choisit l'opération appropriée, dans le cas présent, il va utiliser la fonction multipliant tous les termes d'un vecteur par un scalaire.

2.2 Les types de données

L'utilisateur étant contraint à utiliser uniquement les types de données proposés, il est donc très important de bien en comprendre le fonctionnement et tous les usages possibles. Ces types se regroupent en trois catégories : les données des maillages (sommets, arêtes, triangles, etc.), les champs de solutions libres (des tableaux, scalaires ou vecteurs associés

à chaque entité d'un type de maillage) et enfin les liens topologiques, de loin les plus complexes, qui permettent des accès indirects efficaces entres tous les différents types (boules des points, voisinages par face, coquille d'arêtes, etc.).

Types de données de maillage : ils permettent de représenter tout maillage hybride à l'ordre un, les ordres deux et trois étant prévus pour une version ultérieure. Sont donc proposés les types classiques, Vertices, Edges, Triangles, Quadrilaterals, Tetrahedra, Pyramids, Prisms et Hexahedra.

Les sommets (Vertices) sont composés de trois flottants et d'un entier pour stocker les coordonnées et la référence physique, et tous les autres éléments sont composés de 2 à 8 entiers pour stocker les numéros de sommets ainsi qu'un entier supplémentaire pour la référence.

Remarque : tous les indices de tableaux vont de 0 à $n-1$.

Leur allocation se fait avec la commande `GmlNewMeshData`, en spécifiant simplement le nombre d'entités souhaitées dans le maillage.

Notez qu'une instance de la librairie ne peut comporter qu'un seul tableau d'un même type de maillage, il n'est donc pas possible d'allouer deux tableaux de triangles par exemple.

Leur renseignement se fait ensuite avec la commande `GmlSetDataLine`, en précisant le type de la donnée, le numéro de l'entité (de 0 à n-1), puis un nombre variable d'arguments correspondant aux données composant l'élément en question. L'exemple suivant alloue et renseigne trois sommets :

```
VerIdx = GmlNewMeshData(LibIdx, GmlVertices, 3);
GmlSetDataLine(LibIdx, VerIdx, 0, 1.5, 2.3, 1.0, 8);
GmlSetDataLine(LibIdx, VerIdx, 1, 9.2, 8.3, 1.0, 6);
GmlSetDataLine(LibIdx, VerIdx, 2, 6.3, 1.6, 2.3, 0);
```

Les champs associés : ils permettent d'associer des données librement à toutes les entités d'un type de maillage donné. Il ne peut y avoir qu'un seul type de donnée OpenCL dans un champ, mais ce type peut être vectoriel (taille de 2 à 16) et peut même être composé d'un tableau de vecteurs. Les types scalaires proposés sont : *byte*, *int*, *float* et *double* ainsi que leurs déclinaisons vectorielles. Chaque type de solution est associé à un type de maillage précis, mais il est possible de déclarer autant de champs associés pour chaque type de donnée de maillage que nécessaire ce qui permet de simuler l'équivalent d'une structure hybride, chaque champ stockant un type OpenCL différent.

L'allocation se fait avec la commande `GmlNewSolutionData`, en spécifiant le type de maillage auquel ce champ de solution est associé, la taille du tableau local associé à une entité de maillage (= 1 pour un scalaire), ainsi que le type de la donnée (GmlByte, GmlInt16, GmlFloat4, etc.) et le nom sous lequel apparaitra la variable déclarée automatiquement. Les données sont renseignées avec la commande `GmlSetDataLine`, mais contrairement aux éléments, on ne passe pas toutes les valeurs des champs explicitement, mais seulement un pointeur sur un tableau contenant la totalité du champ associé à cette entité et qui sera recopié dans la structure interne de la librairie.

L'exemple suivant alloue et renseigne un tableau de deux flottants pour chacun des trois sommets alloués précédemment :

```
float vec[2];
VecIdx = GmlNewSolutionData(LibIdx, GmlVertices, 2, GmlFlt, "masse_vitesse");
vec = {1.2, -6.3};
GmlSetDataLine(LibIdx, VecIdx, 0, &vec);
vec = {0.3,  5.2};
GmlSetDataLine(LibIdx, VecIdx, 1, &vec);
vec = {3.4, -0.1};
GmlSetDataLine(LibIdx, VecIdx, 2, &vec);
```

Les liens topologiques : c'est l'entité la plus complexe à appréhender et manipuler, mais aussi la plus puissante, grâce à elle, et un peu d'imagination, on peut contourner l'essentiel des limitations imposées par le calcul sur GPU. Un lien topologique permet de lier un type de maillage source et un type de destination, par exemple lier les triangles vers les tétraèdres, ce qui consiste à fournir les indices des deux tétraèdres voisins pour chaque triangle. Un tel tableau serait associé aux triangles (la source) et serait composé de deux entiers par triangles, car un maximum de deux tétraèdres peut partager un même triangle.

Les liens ne sont pas commutatifs, car inverser les triangles et les tétraèdres revient à donner pour chaque tétraèdre (la source), la liste des quatre triangles qui constituent ses faces. Ces liens topologiques vont être utilisés dans les kernels à accès mémoire indirect, voir section 2.4, et permettront d'accéder aux données associées aux tétraèdres dans une boucle sur les triangles, pour reprendre notre premier exemple.

Un des points forts de la `GMlib` est qu'elle va automatiquement construire les liens standards entre tous les types de données, et ce au moment de la compilation d'un kernel.

Par exemple, si on demande une boucle sur les arêtes, et que des données à lires sont associées à des triangles, la coquille des triangles pour chaque arête sera construite. L'utilisateur ne sera amené à créer ses propres liens topologiques que pour effectuer des accès mémoires indirects spécifiques à son algorithme, par exemple, donner les deux triangles *upwind* et *downwind* pour chaque arête dans un solveur fluide.

Dans un tel cas en 2-D, si l'utilisateur demandait une boucle sur les arêtes et voulait lire des données de calculs associées aux triangles, la librairie construirait le lien topologique naturel partant des arêtes pour pointer sur les triangles : il s'agirait donc des deux triangles partageant une même arête, et ce n'est pas ce dont le solveur a besoin. Dans ce cas, il sera de la responsabilité de l'utilisateur d'allouer un type de donnée topologique avec la commande `GmlNewLinkData` en précisant les étiquettes des types source et destination ainsi que le nombre de liens nécessaires pour chaque source. Vous pourriez, pourquoi pas, avoir envie de créer une table donnant pour chaque sommet les indices des 180 pyramides les plus proches !

L'allocation et le remplissage sont similaires aux champs associés :

```
int pyramides[ NmbVer ][ 180 ];
```

```
PyrIdx = GmlNewLinkData(LibIdx, GmlVertices, GmlPyramides, 180, "Gizeh");
for(i=0;i<NmbVer;i++)
   GmlSetDataLine(LibIdx, PyrIdx, pyramides[i]);
```

Remarque : le lien inverse de sommet vers pyramides, soit une pyramide vers ses cinq sommets, n'est rien d'autre que le tableau des éléments pyramidaux lui-même. De fait, tous les éléments du maillage ne sont que des liens entre de un numéro d'élément et plusieurs numéros de sommets.

2.3 Kernels à accès directs

C'est le type de Kernel le plus simple et comme tout kernel il ne boucle que sur les éléments d'un seul type, et n'accède en lecture ou en écriture qu'à des données relatives au type d'élément de la boucle.

Si on boucle sur les sommets et accède à un champ de solution qui leur est associé, tous les tableaux seront accédés via l'indice de boucle principal : il n'y a donc aucune indirection mémoire. C'est aussi le type de Kernel le plus efficace, car il utilise à 100% la bande passante de la mémoire du GPU étant donné que les tableaux sont lus linéairement.

Dans l'exemple introductif de la section 2.1, les deux types de solutions alloués sont relatifs aux sommets et le kernel d'une ligne boucle sur ces sommets et accède aux coordonnées en lecture et écriture et aux solutions en lecture seulement.

```
VerCrd = VerCrd + VerVitesse * VerDirection;
```

Dans ce kernel, à chaque itération de la boucle de 0 à NmbVer, les coordonnées, la vitesse et le vecteur de direction sont tous lus de manière cachée par la librairie qui y accède via l'indice de boucle principal i. Dans les faits, un GPU transférant tout d'abord la mémoire par blocs de 32 ou 64 entiers consécutifs vers la mémoire cache et accédant ensuite à cette dernière, l'accès à la case `vitesse[0]` va déclencher la lecture de `vitesse[0..31]`, puis les accès suivants, de 1 à 31, ne couteront que le prix d'un accès au cache.

Dans cette configuration, le nommage des variables automatique est évident, elles portent le nom court du type de maillage de la boucle principale auquel est concaténé le nom de la variable accédée. Dans notre exemple, la boucle portant sur les `GmlVertices` (nom court `Ver`), et on accède aux coordonnées (`Crd`), à la vitesse (`Vitesse`) et au vecteur de direction (`Direction`), ce qui donne les noms automatiques suivants : `VerCrd`, `VerVitesse` et `VerDirection`, élémentaire !

Remarque : les accès directs aux données peuvent être fait en mode lecture, ou écriture, ou bien les deux en même temps.

2.4 Kernels a accès indirects

C'est ici que les choses deviennent un peu plus sérieuses. Un kernel est dit à accès indirect lorsqu'il est amené à lire des données associées à un type de maillage différent de

celui de la boucle principale. Dans un tel cas, les accès mémoires prennent la forme `a = u[v[i]]` et non plus `a = u[i]` comme c'est le cas dans les accès directs.

Ces accès indirects sont nettement plus lents pour deux raisons.

Tout d'abord il faut effectuer deux accès mémoire pour obtenir une seule valeur utile au final. Le GPU commence par déclencher une première lecture `tmp = v[i]`, puis lit `a = u[ tmp ]`.

Enfin et surtout, les accès successifs à `a = u[ tmp ]` ne sont pas consécutifs en mémoire lorsqu'on incrémente le compteur, i car les valeurs contenues dans `v[i]` sont totalement inconnues au moment de la compilation et dépendent uniquement de la numérotation des éléments et sommets du maillage.

Afin de limiter le problème, on recourt à deux méthodes, la renumérotation du maillage par une courbe de Hilbert (ou toute autre SFC) afin d'améliorer l'efficacité de la mémoire cache, ainsi qu'a la vectorisation des liens montants (boules de points, coquilles d'arêtes), par nature irréguliers, pour en améliorer la compacité d'accès mémoire (cf xxx). Le déroulement de ces kernels est totalement transparent pour l'utilisateur, car la librairie va se charger de lire le tableau d'indirection, puis les données ainsi accédées. Dans le cas d'une boucle sur les triangles (`Tri`), si on souhaite accéder aux coordonnées des sommets (`Crd`), la librairie va d'elle-même définir et remplir un tableau (`TriVer[3]`) de trois vecteurs contenant les coordonnées des trois sommets du triangle.

À titre d'illustration, voici un code bouclant sur les triangles et calculant leur barycentre :

```
KrnIdx = GmlCompileKernel( LibIdx,  barycentre, "barycentre", GmlTriangles, 3,
                           VerIdx, GmlReadMode , NULL,
                           TriIdx, GmlReadMode , NULL,
                           BarIdx, GmlWriteMode, NULL );
```

Et voici le kernel indirect associé :

```
Bar = (TriCrd[0] + TriCrd[1] + TriCrd[3]) / 3.;
```

La variable `Bar` est un unique vecteur de coordonnées, car il est relatif au triangle qui est l'élément de la boucle principal, alors que la variable `TriCrd` est un tableau de taille 3, car elle est relative aux sommets d'un triangle qui en compte trois.

Comme vous pouvez le constater, il n'est nul besoin d'accéder explicitement aux numéros des trois sommets du triangle, ceci étant effectué par la librairie juste avant de passer la main au kernel de l'utilisateur. Un lien topologique est donc une structure complètement muette et transparente. Ce qui peut être perturbant, c'est que différentes commandes de compilation d'un même kernel mais avec `GmlEdges` à la place de `GmlTriangles` par exemple, va produire des codes différents ! En effet, la variable contenant les coordonnées s'appellera alors `EdgCrd[2]` et sa taille sera de deux au lieu de trois dans le cas d'une boucle principale sur les triangles. On parle alors de variable et de programmation contextuelle.

Une fois maitrisé ce concept, il est alors possible d'accéder indirectement (en lecture seulement) à n'importe quel type de données (maillage ou solution) associées à des types de

maillage complètement différents du type de la boucle principale, car tous les cas possibles de liens indirects sont gérés par automatiquement par la `GMlib`.

Liens topologiques descendants, transverses et montants : un lien est descendant si le type de la boucle est de dimension strictement supérieure au type accédé indirectement (exemple : triangle vers arêtes), transverse si les deux types sont de même dimension (triangle vers les triangles ou quadrilatères voisins) et montant si la dimension du type indirect est strictement supérieure au type de la boucle (boucle sur les sommets et accès aux triangles de leurs boules).

Lien descendant : ce premier type est trivial, il s'agit le plus souvent de boucler sur un élément et d'accéder aux données associées à ses sommets. Dans ce cas, la table d'indirection est la liste des noeuds composant chaque élément et la librairie n'a qu'à la transmettre telle quelle au GPU.

Lien transverse : le second type, présente une petite finesse, car par défaut, si on boucle sur les triangles et qu'on accède à des données relatives à chaque triangle, il s'agit tout simplement d'un accès direct comme vu à la section 2.3. Mais il est aussi possible de fournir un tableau de liens topologiques spécifiques au moment de la compilation du kernel, comme un tableau des voisins par arêtes. Dans ce cas, on pourra boucler sur les triangles et accéder à des données relatives au triangle en cours de traitement ainsi qu'à ses trois voisins. Le code suivant va récupérer les barycentres des triangles, calculés à l'exemple précédent, en même temps que ceux de leurs trois voisins, afin de déterminer la distance maximum entre le centre d'un triangle et celui de ses voisins.

```
NgbIdx = GmlSetNeighbours(LibIdx, GmlTriangles);
DisIdx = GmlNewSolutionData(LibIdx, GmlTriangles, 1, GmlFlt, "distance");
KrnIdx = GmlCompileKernel( LibIdx,  distance, "distance", GmlTriangles, 2,
                           BarIdx, GmlReadMode , NgbIdx,
                           DisIdx, GmlWriteMode, NULL );
```

La première ligne demande la création d'un lien topologique liant les triangles entre eux par voisinage d'arêtes et ce lien, stocké dans la variable `NgbIdx,` est passé à la compilation en paramètre de lien indirect lors de l'accès au tableau des barycentres. De ce fait, le kernel va lire le barycentre associé au triangle en cours ainsi que ceux de ses trois voisins s'ils existent. Si la valeur 0, avait été passée à la place de `NgbIdx`, le kernel se serait contenté de lire le seul barycentre du triangle en cours.

Et voici kernel correspondant :

```
float d1, d2, d3;
d1 = distance(TriBar[0], TriBar[1]);
d2 = distance(TriBar[0], TriBar[2]);
d3 = distance(TriBar[0], TriBar[3]);
```

```
distance = min(d1, d2);
distance = min(distance, d3);
```

Ici, le tableau `TriBar[4]` est nommé de la sorte, car on boucle sur les triangles (`Tri`) et on accède à leurs barycentres (`Bar`), mais il y a quatre barycentres lus pour chaque triangle : le premier est celui du triangle pointé par l'indice de boucle et les trois suivants sont ceux de ses voisins. Notez que si un des voisins est nul (arête frontière), le barycentre associé sera initialisé avec le vecteur $0, 0, 0$. Ici encore, il s'agit de variables vectorielles et une seule ligne de ces calculs traite les trois composantes des coordonnées.

Lien montant : enfin le dernier type est aussi le plus complexe, car les liens montants présentent l'inconvénient majeur d'être de taille variable, chose que détestent les GPU, qui ne sont efficaces qu'avec des tailles régulières et de puissance de 2. En effet, si le nombre de sommets d'un triangle est toujours égal à trois, en revanche, le nombre de triangles partageant un même sommet est variable et dépend de la topologie du maillage. Afin de traiter ces cas efficacement, un système complexe est mis en place par la `GMlib`. Tout d'abord la commande de renumérotation de Hilbert, nécessaire pour obtenir de bonnes performances d'accès mémoire, propose un mode spécifique au calcul sur GPU et va ordonner les sommets du maillage selon leur degré croissant. Ainsi, tous les premiers sommets ayant un degré inférieur ou égal à 8 auront leur boule de triangle stockée dans un vecteur de cette taille (avec d'éventuelles valeurs nulles), puis les boules des sommets suivants seront stockées sur des vecteurs de taille 16 et ainsi de suite. Tout ceci est transparent pour l'utilisateur qui pourra connaitre le degré du sommet en cours avec la variable automatique `VerTriDeg`, soit Vertex triangles degree.

Remarque : une variable accédée indirectement via un lien montant pourra être nulle du fait que les tailles des vecteurs sont alignées sur des puissances de deux et que le degré réel peut donc être inférieur. Les données inutilisées sont toujours initialisées à 0, ce qui peut perturber certains calculs.

Pour rester dans la même thématique, nous allons cette fois-ci boucler sur les sommets, accéder aux barycentres des triangles de la boule, et relaxer les coordonnées du sommet vers la moyenne des barycentres.

```
CrdIdx = GmlNewSolutionData(LibIdx, GmlVertices, 1, GmlFlt4, "relaxed");
KrnIdx = GmlCompileKernel( LibIdx,  smoothing, "smoothing", GmlVertices, 2,
                           BarIdx, GmlReadMode , NULL,
                           CrdIdx, GmlWriteMode, NULL );
```

Le lien montant des sommets vers les triangles (*i. e.* leur boule), est créé automatiquement, car c'est le lien naturel dans le domaine du maillage. Notez qu'il n'est nul besoin de spécifier l'accès aux types de sommets (`VerIdx`) ni de triangles (`TriIdx`), mais seulement aux types de solutions contenant les barycentres et les coordonnées relaxées car la librairie sait que le premier se rapporte aux triangles et le second aux sommets, et en déduit le lien adéquat.

Voici le kernel correspondant :

```
int i;
float4 NewCrd = {0};
for(i=0;i<VerTriDegMax;i++)
  NewCrd += VerTriBar[i];
relaxed = 0.8 * relaxed + 0.2 * NewCrd / VerTriDeg;
```

Dans le cas de kernel montant, une nouvelle variable automatique apparait : il s'agit du degré de l'entité de boucle en cours de traitement, soit dans le cas présent, le degré de la boule du point. Cette valeur n'étant pas constante dans un maillage non structuré, la variable `VerTriDeg` est renseignée pour chaque sommet afin d'effectuer le bon nombre d'itérations de la boucle interne additionnant les barycentres des triangles et effectuant la moyenne par une divisant finale.

De fait, deux variables contenant le degré du sommet sont définies : `VerTriDeg`, qui contient le degré exact, et `VerTriDegMax` qui contient la taille du vecteur dans lequel sont stockés les barycentres de la boule. Ce dernier est toujours dimensionné sur la puissance de 2 par valeur supérieure. Comme on peut le constater dans le kernel, on effectue la boucle sur `VerTriDegMax`, qui vaut 8 dans le cas présent, quitte à additionner inutilement des barycentres dont la valeur est nulle, ce qui ne perturbe pas le calcul. À la fin de la boucle, on divise cette fois-ci par la vraie valeur du degré contenue dans `VerTriDeg`.

Ces calculs inutiles sont en fait plus performants sur GPU, car les itérations d'une boucle dont la taille est connue au moment de la compilation sont effectuées en parallèle par un groupe de *compute units* (64 le plus souvent), alors qu'une boucle dont la taille est connue dynamiquement au moment de l'exécution est sérialisée, divisant ainsi sa vitesse d'un ou deux ordres de grandeur !

2.5 Paire de Kernels scatter / gather

Une des problématiques les plus fréquentes des codes accédant à des maillages non structurés de manière concurrente, qu'ils soient multithread, parallèles ou vectoriels, est l'accès en écriture via des indirections mémoires. C'est le cas d'une boucle sur les triangles durant laquelle on écrit des données associées à leurs sommets. On comprend bien que deux triangles traités simultanément peuvent alors écrire sur un même sommet qu'ils auraient en commun. Pour contourner le problème, il existe de nombreuses techniques, comme la découpe en sous-blocs de triangles n'ayant aucun sommet en commun (cf ref xxx), les micro synchronisations afin d'éviter les conflits d'écritures simultanées (cf. ref xxx) et enfin la duplication des données : c'est la méthode scatter-gather, idéale pour les GPU.

Il est en effet toujours possible de casser une boucle à écriture indirecte en deux boucles à lecture indirectes, mais écriture directe, qui ne présentent donc plus de conflit d'écriture. Pour illustrer le propos, prenons l'exemple d'une boucle calculant une valeur pour chaque triangle et additionnant celle-ci à un champ associé à leurs sommets. Il s'agit d'une boucle à écriture indirecte puisqu'on parcourt les triangles et qu'on écrit aux sommets.

On va donc la scinder en deux sous-boucles. La première parcourt les triangles, effectue le calcul et stocke le résultat dans un champ temporaire associé à ces triangles : on boucle

et on écrit sur les triangles, il s'agit donc d'une boucle sans conflit d'écriture mémoire. La seconde parcourt les sommets et additionne les contributions de chaque triangle stockées dans le champ temporaire et enfin écrit le résultat final dans le champ associé à ces sommets : on boucle et on écrit sur les sommets et il n'y a donc plus de conflit d'écriture là non plus. Ces boucles ne sont pas réellement indépendantes, car elles effectuent de fait une partie d'un même calcul qui aurait pu être réuni dans une seule boucle à écriture indirecte. La première est habituellement qualifiée de *scatter*, car elle ventile les résultats d'un calcul dans des variables locales et temporaires, et la seconde est appelée *gather*, car elle n'effectue pas de calcul utile proprement dit, mais ne fait que réunir les contributions partielles précédemment stockées.

C'est pourquoi on qualifie cette technique de *paire scatter-gather*.

Compilation du kernel de *scatter* :

```
TmpIdx = GmlNewSolutionData(LibIdx, GmlTriangles, 1, GmlFlt4, "Tmp");
KrnIdx = GmlCompileKernel( LibIdx,  scatter, "scatter", GmlTriangles, 3,
                           VerIdx, GmlReadMode , NULL,
                           TmpIdx, GmlWriteMode, NULL );
```

Code OpenCL du kernel de *scatter* :

```
Tmp = (TriCrd[0] + TriCrd[1] + TriCrd[3]) / 3.;
```

Compilation du kernel de *gather* :

```
SolIdx = GmlNewSolutionData(LibIdx, GmlVertices, 1, GmlFlt4, "VerSol");
KrnIdx = GmlCompileKernel( LibIdx,  gather, "gather", GmlVertices, 2,
                           TmpIdx, GmlReadMode , NULL,
                           SolIdx, GmlWriteMode, NULL );
```

Code OpenCL du kernel de *gather* :

```
int i;
float4 NewSol = {0};
for(i=0;i<VerTriDegMax;i++)
  NewSol += VerTriTmp[i];
VerSol = NewSol / VerTriDeg;
```

2.6 Programmation en OpenCL

3 Exemples

3.1 Calcul d'une valeur moyenne pour chaque triangle d'un maillage

3.2 Boucle à accès mémoire indirect présentant des dépendances mémoires

4 Liste des commandes

4.1 GmlCheckFP64

Permets de tester la présence ou non d'unités de calcul flottant en double précision. La norme OpenCL rend obligatoire la présence d'unités 32-bits, mais les capacités 16-bits (demi-précision) et 64-bits sont optionnelles.

Syntaxe

```
flag = GmlCheckFP64(LibIdx);
```

Paramètres

Paramètre	type	description
LibIdx	size_t	l'index de l'instance tel que retourné par GmlInit()

Retour	type	description
flag	int	0 : pas de capacité de calcul flottant 64-bit, 1 : FP64 disponible

Commentaires

Bien que des unités de calculs double précision soient présentent sur la plupart des GPU de jeux vidéos, leur nombre est souvent 1/4 à 1/32 du nombre d'unités 32-bits, réduisant d'autant la vitesse de calcul. Cette dernière est alors inférieure à celle du CPU principal ! La capacité FP64 du GPU est alors un faux ami.

4.2 GmlCompileKernel

Syntaxe

```
KrnIdx = GmlCompileKernel(LibIdx, OclSrc, PrcNam, MshTyp, NmbDat, ...);
```

Voici certainement la plus complexe des procédures de la librairie !

Son rôle est de créer un exécutable compatible avec la carte graphique sélectionnée à partir d'un source OpenCL en ASCII, d'un type d'éléments pour la boucle principale et d'une liste de données à accéder en lecture ou écriture, directement ou indirectement.

Les paramètres essentiels sont le type de la donnée de maillage sur laquelle boucler et qui va déterminer le mode d'accès aux autres données de maillage ou de solution qui vont suivre. Le paramètre suivant étant le nombre de données utilisateurs auquel le kernel pourra accéder et pour chacune d'entre elles devront être précisés les modes d'accès à la mémoire et d'éventuelles options dont voici le détail.

Pour chaque type de donnée, on fournit le numéro de l'étiquette. Le mode d'accès de la donnée, direct ou indirect via une table topologique, est déterminé en fonction du type de donnée de la boucle et celui de la donnée accédée.

Dans le cas où la donnée est de dimension égale, il s'agit d'un accès direct et la donnée utilisateur sera composée d'une seule entrée : par exemple, si on boucle sur les triangles et qu'on accède à un champ de solution contenant la normale de chaque triangle, la donnée générée correspondante sera du type `float4 TriNrm`, car on traite un seul triangle par itération de la boucle et un triangle ne possède qu'une seule normale stockée dans un `float4`.

Dans le cas où la donnée accédée est de dimension inférieure, la donnée sera alors présentée sous forme d'un tableau dont la taille est égale au nombre d'entités du type accédé qui composent le type de maillage de la boucle principale. Par exemple, si on boucle sur les triangles et qu'on accède à un champ de solution stockant la normale à chaque sommet, comme il y a trois sommets par triangle, le type sera défini comme ceci : `float4 TriVerNrm[3]` car il y a trois normales stockées chacune sur un vecteur de type `float4`.

Dans le cas où la donnée accédée est de dimension supérieure, il s'agit d'un kernel montant (boule ou coquille), et là aussi la donnée sera présentée sous forme d'un tableau dont la dimension sera variable et définie séparément pour chaque entité de la boucle. Si cette fois-ci on boucle sur les sommets et accède aux normales associées aux triangles, le champ sera défini comme ceci : `VerTriNrm[ VerTriDeg ]`, et la variable `VerTriDeg` sera définie à la volée pour chaque sommet.

De plus, il est possible de court-circuiter le système de lien topologique par défaut et de fournir son propre tableau de lien ad hoc, comme le tableau des voisins par exemple. Si l'om souhaite boucler sur les tétraèdres et accéder à un champ de solution associé aux tétraèdres de l'itération de la boucle, mais aussi de ses quatre voisins potentiels, on peut générer cette table avec la commande `GmlSetNeighbours()` et la passer en paramètre de lien topologique à la création du kernel.

L'exemple suivant va illustrer la différence entre tous ces types d'accès en partant du même type de donnée. Imaginons que nous partons d'un maillage composé de sommets, de triangles, de tétraèdres et d'un champ associé à chaque triangle et contenant son aire. Nous allons accéder en lecture aux aires des triangles, mais en bouclant sur les sommets dans un premier kernel, puis sur les tétraèdres dans un deuxième, sur les triangles dans un troisième et enfin sur les triangles et leurs voisins dans le dernier. Vous constaterez que le type de donnée contenant l'aire, portant défini comme un simple scalaire, change

complètement selon les kernels !

Paramètres

Paramètre	type	description
LibIdx	size_t	l'index de l'instance tel que retourné par GmlInit()
OclSrc	char *	pointeur sur une chaine de caractères contenant le source OpenCL du kernel à compiler
PrcNam	char *	pointeur sur une chaine contenant le nom de la procédure telle qu'elle apparaitra dans les kernel OpenCL générés
MshTyp	int	type de donnée sur lequel le kernel va boucler : cette information est capitale, car elle va déterminer les types, dimensionnements et modes d'accès mémoires de toutes les données utilisateurs qui seront accédées par le kernel
NmbTyp	int	nombre de types de données qui seront lues et/ou écrites par le kernel, pour chaque donnée on attend le jeu de trois paramètres suivants :
DatIdx	int	étiquette du type de donnée à accéder
flags	int	liste de flags concaténés à l'aide de l'opérateur logique *ou*, représenté par une barre (`|` en C, tel que `GmlReadMode`, `GmlWriteMode`, `GmlRefFlag` et `GmlVoyeurs`
LnkIdx	int	index d'un type de lien topologique spécifique à suivre pour accéder aux données, si cette valeur est nulle, la librairie passera par un lien construit par défaut.

Retour	type	description
DatIdx	int	numéro de la nouvelle étiquette de kernel

Commentaires

4.3 GmlDebugOff

Désactive le mode de débogage (qui désactivé par défaut).

Syntaxe

```
GmlDebugOff(LibIdx);
```

Paramètres

Paramètre	type	description
LibIdx	size_t	l'index de l'instance tel que retourné par GmlInit()

4.4 GmlDebugOn

Active le mode de débogage qui permet d'afficher l'intégralité des sources compilées avec notamment les parties de lectures et écriture mémoire ajoutées par la librairie.

Syntaxe

```
GmlDebugOn(LibIdx);
```

Paramètres

Paramètre	type	description
LibIdx	size_t	l'index de l'instance tel que retourné par GmlInit()

Retour	type	description

4.5 GmlDownloadParameters

Cette fonction copie le contenu de la structure de paramètres, définie avec `GmlNewParameters`, de la mémoire du GPU vers celle du CPU hôte afin d'en exploiter le contenu après le lancement d'un kernel qui y aurait écrit certaines informations.

Syntaxe

```
GmlDownloadParameters(LibIdx);
```

4.6 GmlExtractEdges

Cette procédure construit la table des arêtes uniques présentes dans tout le volume. Pour ce faire, elle analyse la totalité des éléments de dimension 1 (arêtes), 2 (faces) et 3 (volumes) afin d'en extraire les arêtes. Si une table d'arêtes avait déjà été renseignée, mais n'était pas complète, ne comportant que certaines arêtes caractéristiques du maillage par exemple, celle-ci est conservée et les nouvelles arêtes sont ajoutées à la suite de la table. Dans ce cas, l'étiquette du type de donnée d'arêtes aura changé, car l'ancienne table est libérée et recopiée dans la nouvelle. Cette nouvelle étiquette est la valeur de retour de la procédure (`NewEdgIdx = GmlExtractEdges(LibIdx)`).

Syntaxe

```
DatIdx = GmlExtractEdges(LibIdx);
```

Paramètres

Paramètre	type	description
LibIdx	size_t	l'index de l'instance tel que retourné par GmlInit()

Retour	type	description
DatIdx	int	numéro de la nouvelle étiquette de donnée d'arête

Commentaires

La numérotation des arêtes ainsi générée peut ne pas être optimale, car celles-ci ayant été générées en bouclant consécutivement sur les éléments, la numérotation des arêtes suit globalement celles des autres éléments volumiques, mais n'est pas aussi optimale qu'une vraie renumérotation via une SFC comme Hilbert. De plus, ces arêtes n'ont pas été triées selon leur degré de coquille ce qui a pour conséquence de consommer plus de mémoire et de temps de lecture dans le cas d'un kernel montant arête vers tétraèdres par exemple. Les arêtes seront toutes stockées dans la zone à haut degré, soit 32, dans le cas des coquilles de tétraèdres, alors que la moyenne est à 6. Dans une telle situation, il faut envisager de renuméroter les arêtes selon leur degré et une courbe de Hilbert afin d'optimiser l'exécution d'un kernel d'arêtes vers tétraèdres.

4.7 GmlExtractFaces

Cette procédure construit la table des faces uniques présentes dans tout le volume, triangulaire ou quadrilatéral. Pour ce faire, elle analyse la totalité des éléments de dimension 2 (faces) et 3 (volumes) afin d'en extraire les faces. Si une table de face avait déjà été renseignée, mais n'était pas complète, ne comportant que des faces frontières par exemple, celle-ci sont conservées et les nouvelles faces sont ajoutées à la suite de la table. Dans ce cas, les étiquettes des types de données triangles et quadrilatère auront changé, car les anciennes tables auront été libérées et recopiées dans les nouvelles. Contrairement à l'extraction d'arêtes, la valeur de retour de la procédure est la somme des triangles et quadrilatères du maillage. Afin de récupérer les nouvelles étiquettes des types de données `GmlTriangles` et `GmlQuadriaterals`, il faut passer par la commande `GmlMeshInfo()`.

Syntaxe

```
DatIdx = GmlExtractFaces(LibIdx);
```

paramètres

Paramètre	type	description
LibIdx	size_t	l'index de l'instance tel que retourné par GmlInit()

Retour	type	description
DatIdx	int	somme des triangles et quadrilatères du maillage

Commentaires

Comme pour les arêtes, la numérotation des faces ne sera pas optimale et seulement dérivée de celle des éléments de volume. Néanmoins, il n'y a pas de problème de degré des liens montant vers les éléments de volume, car celui-ci est fixé à deux voisins de part et d'autre de chaque face.

4.8 GmlFreeData

Syntaxe

Libère la mémoire occupée sur le CPU et le GPU par cette donnée.

```
flag = GmlFreeData(LibIdx, DatIdx);
```

Paramètres

Paramètre	type	description
LibIdx	size_t	l'index de l'instance tel que retourné par GmlInit()
DatIdx	int	

Retour	type	description
flag	int	

Commentaires

L'étiquette sera réutilisée par de futures créations de données, donc faire attention à ne pas confondre l'ancienne et la nouvelle dans ses listes de types.

4.9 GmlGetDataLine

Permet de récupérer les données du GPU et de les stocker dans les variables de l'utilisateur. Cette procédure concerne tous les types de données, maillage, solutions et liens topologiques. Le nombre d'arguments est variable et l'utilisateur doit fournir un pointeur de type adéquat pour chaque scalaire composant une ligne de ce type.

Syntaxe

```
flag = GmlGetDataLine(LibIdx, DatIdx, LinNmb, ...);
```

Paramètres

Paramètre	type	description
LibIdx	size_t	l'index de l'instance tel que retourné par GmlInit()
DatIdx	int	index de la donnée à renseigner
LinNmb	int	numéro de la ligne dans cette donnée
...	int * / float *	pointeur sur un entier ou un flottant qui

Retour	type	description
flag	int	0 : échec, 1 : succès

Commentaires

Après l'exécution d'un kernel sur le GPU, les données indiquées comme GMLWriteMode ne sont pas immédiatement transférées du GPU vers le CPU. C'est l'appel à GmlGetDataLine() de la première ligne de donnée qui déclenche le transfert en bloc de la totalité des lignes du champ. Il en découle que si vous commencez par lire les données sans commencer par la première ligne, les données retournées seront vides.

4.10 GmlGetKernelRunTime

Syntaxe

Permets de connaitre le temps d'exécution total de ce kernel au moment de l'appel de la fonction. Le temps retourné est basé sur les compteurs de profilage internes d'OpenCL et peut être sensiblement inférieur au temps physique écoulé qui est calculé avec la fonction `GmlGetWallClock`.

```
temps = GmlGetKernelRunTime(LibIdx, KrnIdx);
```

Paramètres

Paramètre	type	description
LibIdx	size_t	l'index de l'instance tel que retourné par GmlInit()
KrnIdx	int	numéro du kernel à évaluer

Retour	type	description
temps	double	temps en secondes

4.11 GmlGetMemoryTransfer

Syntaxe

Permet de connaitre à tout instant la quantité de mémoire en octets transférée de ou vers le GPU.

```
taille = GmlGetMemoryUsage(LibIdx);
```

Paramètres

Paramètre	type	description
LibIdx	size_t	l'index de l'instance tel que retourné par GmlInit()

Retour	type	description
taille	size_t	

4.12 GmlGetMemoryUsage

Permets de connaitre la quantité de mémoire en octets actuellement utilisée par la `GMlib` sur le GPU.

Syntaxe

```
taille = GmlGetMemoryUsage(LibIdx);
```

Paramètres

Paramètre	type	description
LibIdx	size_t	l'index de l'instance tel que retourné par GmlInit()

Retour	type	description
taille	size_t	

4.13 GmlGetReduceRunTime

Permets d'obtenir le temps total d'exécution d'un des kernels de réduction de la librairie. Le fonctionnement est identique à `GmlGetKernelRunTime` sauf l'étiquette de kernel qui est remplacée par le code de l'opération de réduction dont on veut récupérer le chronomètre (GmlMin, GmlMax, Fmlsum).

4.14 GmlGetWallClock

Syntaxe

Retourne simplement le temps physique en secondes. Cette fonction est très utile pour s'affranchir des informations de profilages, toujours incomplètes, et des perturbations des fonctions système d'horloges telles que `clock()` liées aux environnements multithreads.

```
temps = GmlGetWallClock();
```

Retour	type	description
temps	double	temps absolu exprimé en secondes

4.15 GmlImportMesh

Cette routine est une aide (*helper*) qui permet en une seule commande de lire un maillage, d'initialiser les types de données qu'il contient sur le GPU et de les transférer. Elle n'est présente dans la `GMlib` qui si elle a été compilée avec le flag `-DWITH_LIBMESHB`. L'idée est de donner un nom de fichier mesh et la liste des mots clefs que vous souhaitez importer, s'ils sont présents dans le fichier. La procédure retourne simplement le nombre de ces mots-clefs trouvés. Afin d'en connaitre les index pour les manipuler par la suite, il fait appeler pour chacun d'eux la commande GetMeshInfo().

Syntaxe

```
NmbKwd = GmlImportMesh(LibIdx, FilNam, ...);
```

Paramètres

Paramètre	type	description
LibIdx	size_t	l'index de l'instance tel que retourné par GmlInit()
FilNam	char *	chemin d'accès complet et noms du fichier avec son extension
...	int	liste des mots-clefs de maillages ou solution tels que définis par la `libMeshb`

Retour	type	description
NmbKwd	int	nombre de mots-clefs effectivement trouvés et lus

Commentaires

Attention à passer en arguments les étiquettes de mots-clefs définis par la `libMeshb` et non par la `GMlib`!

4.16 GmlInit

Permets d'initialiser une instance de la librairie avec une unité de calcul capable d'exécuter du code OpenCL. La liste de ces unités est disponible via la commande *GmlListGPU*. Notez qu'il est possible d'ouvrir plusieurs instances de la `GMlib` afin de bénéficier de la capacité de calcul des différentes unités et même d'ouvrir plusieurs instances sur la même unité, dont le temps machine sera alors partagé.

Syntaxe

```
LibIdx = GmlInit(DevIdx);
```

Paramètres

Paramètre	type	description
DevIdx	int	numéro de l'unité de calcul sur laquelle exécuter tous les kernels de cette instance

Retour	type	description
LibIdx	size_t	identifiant unique de l'instance de la librairie

Commentaires

Par défaut, les systèmes rangent les unités de type CPU en premier et les GPU en dernier. De fait, l'unité 0 étant presque toujours un CPU, il est pertinent d'exécuter un `GmlInit(0)` par défaut en l'absence d'indication de la part de l'utilisateur. Un tel choix est garanti de fonctionner sur tout système.

4.17 GmlLaunchKernel

Lancement de l'exécution d'un kernel déjà compilé. Cette procédure prend très peu de paramètres, car tout a été défini au moment de la compilation : nombre d'itération de la boucle, les données à lire ou écrire, code à compiler ainsi que le GPU ou CPU exécutant. Il ne reste alors qu'à donner un index d'instance de librairie et un numéro de kernel et ça démarre !

Syntaxe

```
flag = GmlLaunchKernel(LibIdx, KrnIdx);
```

Paramètres

Paramètre	type	description
LibIdx	size_t	l'index de l'instance tel que retourné par GmlInit()
KrnIdx	int	numéro du kernel à exécuter

Retour	type	description
flag	int	si < 0 : code d'erreur, 1 = succès

4.18 GmlListGPU

Cette procédure se contente d'afficher à l'écran les noms complets des unités de calculs compatibles OpenCL sur le système. En cas de lancement d'un logiciel en ligne de commande sans entrer d'unité de calcul GPU, il est utile d'appeler `GmlListGPU()` comme message d'aide par défaut.

Syntaxe

```
GmlListGPU();
```

Commentaires

Les systèmes rangent en général des unités de type CPU dans les premiers indices et les GPU en dernier.

4.19 GetMeshInfo

Permets de retrouver le numéro de datatype et le nombre de lignes à partir du simple type de donnée de maillage. Cela est utile après l'appel de `GmlImportMesh` afin de connaitre les informations sur tous les mots-clefs importés du fichier de maillage ou solutions.

Syntaxe

```
flag = GetMeshInfo(LibIdx, MshTyp, NmbLin, DatIdx);
```

Paramètres

Paramètre	type	description
LibIdx	size_t	l'index de l'instance tel que retourné par GmlInit()
MshTyp	int	
NmbLin	int	
DatIdx	int *	

Retour	type	description
flag	int	

Commentaires

Si un des pointeurs NmbLin ou DatIdx est nul, il ne sera tout simplement pas renseigné.

4.20 GmlNewLinkData

Cette commande permet de créer des liens topologiques arbitraires entre deux types de données de maillage. Elle fonctionne de la même manière que les autres créations de données, `GmlNewMeshData` et `GmlNewSolutionData`, à savoir qu'on en précise le nombre de champs (des entiers) et qu'on boucle sur chaque ligne pour les renseigner. Mais c'est un outil extrêmement puissant, car il permet de contourner les limitations d'accès mémoire indirectes des GPU en passant par des tables d'indirection astucieusement conçues.

Syntaxe

```
DatIdx = GmlNewLinkData(LibIdx, MshTyp, LnkTyp, NmbDat, LnkNam);
```

Paramètres

Paramètre	type	description
LibIdx	size_t	l'index de l'instance tel que retourné par GmlInit()
MshTyp	int	étiquette de l'entité de maillage source
LnkTyp	int	étiquette de l'entité de maillage qui est pointée par la source
NmbDat	int	nombre d'entités de type LnkTyp pointées par chaque entité de type MshTyp
LnkNam	char *	nom donné au tableau contenant le lien tel qu'il sera transmis au code OpenCL

Retour	type	description
DatIdx	int	étiquette de la donnée résultante

Commentaires

Ce lien topologique arbitraire est destiné à se substituer au lien par défaut généré automatiquement par la librairie. Il faut alors en passer l'étiquette à la place de la valeur 0 par défaut pour chaque type de donné concerné dans l'appel de compilation d'un kernel.

4.21 GmlNewMeshData

Création d'un type de donné de maillage, pour le moment seul l'ordre un, mais mixte est proposé. Le type `GmlVertices` est composé des coordonnées et d'une référence et tous les autres éléments sont composés des numéros de leurs sommets et d'une référence. Notez qu'il ne peut y avoir qu'un seul tableau d'un type d'entité par instance de la librairie. Cette limitation à un maillage par instance sera levée dans les prochaines versions.

Syntaxe

```
DatIdx = GmlNewMeshData(LibIdx, MshTyp, NmbLin);
```

Paramètres

Paramètre	type	description
LibIdx	size_t	l'index de l'instance tel que retourné par GmlInit()
MshTyp	int	type d'entité de maillage tel que défini dans le header
NmbLin	int	nombre de ligne de ce tableau d'entités

Retour	type	description
DatIdx	int	étiquette du datatype alloué et à passer en argument aux autres routines

4.22 GmlNewParameters

Permets d'allouer une structure librement définie par l'utilisateur et contenant toute sorte de paramètres nécessaires à la communication entre le CPU et le GPU. Une image miroir du côté CPU et du côté GPU pourra être manuellement synchronisée, avant l'exécution d'un kernel en appelant la commande `GmlUploadParameters`, la structure sera alors recopiée de la mémoire CPU vers le GPU afin qu'il puisse y lire des paramètres et à la fin. Enfin, la copie en mémoire GPU sera rapatriée sur le CPU afin d'y lire d'éventuels résultats de calculs ou codes d'erreurs après appel à la commande `GmlDownloadParameters`.

Syntaxe

```
ParDat = GmlNewParameters(LibIdx, taille, source);
```

Paramètres

Paramètre	type	description
LibIdx	size_t	l'index de l'instance tel que retourné par GmlInit()
taille	int	c'est le sizeof(my_struct) en octets
source	char *	source du code contenant la définition de la structure afin de la rendre visible côté OpenCL

Retour	type	description
ParDat	void *	pointeur sur la structure allouée en mémoire CPU

Commentaires

Afin de garantir que la structure est identique du côté CPU et GPU, il est habile d'inclure sa définition dans un header coté CPU et de passer ce header comme code OpenCL à compiler via la commande `GmlNewParameters`. Bien que la taille de la structure ne soit pas limitée, si elle est transférée deux fois à chaque lancement de kernel (upload / download), cela pénalisera le temps d'exécution si elle est utilisée pour transférer de gros tableaux. Ceci est en revanche très utile pour déboguer un code en recopiant la totalité d'un tableau problématique dans la structure afin de l'analyser du côté CPU.

4.23 GmlNewSolutionData

Création d'un type de donné libre associé à une entité de maillage. Bien qu'il y ait "solution" dans le nom, ce type est fait pour y stocker ce que l'on veut. On choisit le type OpenCL (entier, flottant, scalaire, vecteur) et la taille du tableau local, attention, il s'agit du nombre de variables associé à chaque entité de maillage, pas du nombre total de lignes de ce champ qui est par définition égal au nombre de lignes de l'entité de maillage auquel il fait référence. Il n'est pas possible de stocker des types différents, à la manière d'une structure en C, mais seulement un tableau homogène. Si vous avez besoin de mélanger des deux entiers et des quatre flottants associés à chaque triangle par exemple, il est tout à fait

possible d'allouer deux `SolutionData` associées aux triangles, l'une composé d'un `GmlInt2` et l'autre d'un
GmlFloat4. Notez qu'il est possible de demander un tableau de `2 x GmlInt` et un autre de
4 x GmlFloat, ce qui revient au même.

Syntaxe

```
DatIdx = GmlNewSolutionData(LibIdx, MshTyp, NmbDat, ItmTyp, DatNam);
```

Paramètres

Paramètre	type	description
LibIdx	size_t	l'index de l'instance tel que retourné par GmlInit()
MshTyp	int	étiquette du type de maillage auquel ce champ est associé
NmbDat	int	taille du tableau local
ItmTyp	int	type de chaque entrée du tableau tel que défini dans le header
DatNam	char *	nom du champ tel qu'il apparaitra du côté OpenCL

Retour	type	description
DatIdx	int	étiquette du datatype à passer aux autres routines

Commentaires

Seuls les entiers (8 et 32 bits) et les flottants (32 et 64 bits) sont disponibles pour le moment. Tout type est disponible sous la forme d'un scalaire ou d'un vecteur de puissance 2 allant jusqu'à 16.

4.24 GmlReduceVector

Réduction d'un vecteur de flottant à une seule valeur scalaire (norme). Le format d'entrée est imposé, il s'agit d'un tableau contenant un seul scalaire flottant simple précision par ligne. Les normes de vecteur sont pré codées et il n'est pas possible de définir les siennes pour l'instant, mais il sera possible ultérieurement de proposer des kernels de réduction. Afin de contourner ces limitations, il faut passer par un kernel intermédiaire qui va réaliser le calcul souhaiter et en stocker le résultat dans un tableau de flottant qui sera réduit par `GmlReduceVector`. Les opérations proposées sont la somme des termes, la valeur minimum et la maximum.

Syntaxe

```
flag = GmlReduceVector(LibIdx, DatIdx, OppCod, norme);
```

Paramètres

Paramètre	type	description
LibIdx	size_t	l'index de l'instance tel que retourné par GmlInit()
DatIdx	int	étiquette du datatype contentant le vecteur à réduire
OppCod	int	code de l'opération de réduction à effectuer telle que défini dans le header
norme	double *	valeur résiduelle

Retour	type	description
flag	int	si < 0 : code d'erreur, 1 : succès

Commentaires

Les codes OpenCL des trois opérations proposées se trouvent dans le fichier reduce.cl et il est très simple de s'en inspirer afin d'y ajouter son propre calcul de norme. Les opérations proposées par défaut sont le calcul de la somme des termes du vecteur, de la valeur minimum ou maximum, ou bien des normes L_0, L_1, L_2 et L_{inf}.

4.25 GmlSetDataBlock

Cette procédure est similaire à `GmlSetDataLine` mais transfert la totalité des lignes d'un datatype d'un seul coup au lieu d'une ligne à la fois. Cela permet d'une part de meilleures performances, mais aussi d'écrire des routines de transfert génériques à tous types de données étant donné que les paramètres ne font appel qu'à des pointeurs en mémoire et non aux valeurs, forcément différente, contenue dans chaque type de donnée. On indique donc, le numéro de la ligne de départ, celle de fin, et les pointeurs sur les données utilisateur de la première ligne et de la dernière à transférer.

Syntaxe

```
flag = GmlSetDataBlock(LibIdx, TypIdx, BegIdx, EndIdx, DatBeg, DatEnd, RefBeg,
RefEnd);
```

Paramètres

Paramètre	type	description
LibIdx	size_t	l'index de l'instance tel que retourné par GmlInit()
TypIdx	int	étiquette du datatype à renseigner
BegIdx	int	numéro de la première ligne à envoyer
EndIdx	int	numéro de la dernière ligne
DatBeg	void *	pointeur sur les données utilisateur de la première ligne
DatEnd	void *	idem sur la dernière
RefBeg	int *	pointeur sur la référence associé à la première ligne ou NULL s'il ne s'agit pas d'un type de donnée de maillage (solution)
RefEnd	int *	pointeur sur la dernière référence

Retour	type	description
flag	int	0 : échec, 1 : succès

Commentaires

La procédure est pour l'instant bloquante, mais elle sera lancée en parallèle dans un thread à l'avenir afin de réaliser les initialisations de données en pipeline. De plus, il est prévu que la routine `GmlImportMesh` utilise `GmfReadBlock` et `GmlSetDataBlock` en pipeline afin de lire, préparer et transférer les données en même temps.

4.26 GmlSetDataLine

Renseigne une ligne d'un type de donné, de maillage ou autre, afin que la librairie puisse la stocker sur CPU et GPU. Le nombre d'arguments dépend bien évidemment du type de la donnée. Cette routine permet de renseigner les données de maillage, de solutions ou bien des liens arbitraires entre types. Le nombre d'arguments et leur type est variable et correspond à la définition de l'entité de maillage ou du lien topologique tel que défini par l'utilisateur. Attention, dans le cas des données de solution, on passe un seul et unique pointeur vers la table les contenant de manière consécutive en mémoire, ceci afin de rendre génériques les routines de transfert, car la nature et la taille de ces champs varient selon le contexte du solveur.

Syntaxe

```
flag = GmlSetDataLine(LibIdx, DatIdx, LinNmb, ...);
```

Paramètres

Paramètre	type	description
LibIdx	size_t	l'index de l'instance tel que retourné par GmlInit()
DatIdx	int	étiquette du datatype à renseigner
LinNmb	int	numéro de la ligne à renseigner
...	int/float	liste variable d'arguments contenant les données de ce type de maillage ou de lien topologique ou bien un pointeur sur le tableau contenant la solution associée

Retour	type	description
flag	int	0 : échec, 1 : succès

Commentaires

Faire très attention aux paramètres de la section variable, car le compilateur C n'a aucun moyen d'en vérifier le nombre et le type requis.

4.27 GmlSetNeighbours

Création d'une donnée de type lien topologique et contenant la liste des voisins entre entités du même type. Deux entités de dimension D sont dites voisines si elles partagent une même entité géométrique de dimension D-1 (par exemple deux tétraèdres sont voisins s'ils partagent un triangle). Cette liste pourrait tout à fait être créée manuellement par l'utilisateur avec un appel à `GmlNewLinkData` puis en créant lui-même le tableau des voisinages et en la renseignant avec `GmlSetDataBlock`. La procédure `GmlSetNeighbours` est proposée comme *helper* afin d'en faciliter la tâche et de réduire la taille des codes.

Syntaxe

```
DatIdx = GmlSetNeighbours(LibIdx, EleIdx);
```

Paramètres

Paramètre	type	description
LibIdx	size_t	l'index de l'instance tel que retourné par GmlInit()
EleIdx	int	étiquette du type de donnée de maillage dont on veut construire les voisins

Retour	type	description
DatIdx	int	étiquette sur une donnée de type lien topologique

Commentaires

Le lien topologique ainsi créé pourra être fourni à la compilation d'un kernel à la place du lien par défaut (valeur 0 du paramètre).

4.28 GmlStop

Libère tous les contextes et structure OpenCL, la mémoire sur le CPU et le GPU, et termine l'instance, mais pas la librairie qui peut continuer à fonctionner avec les éventuelles autres instances allouées.

Syntaxe

```
GmlStop(LibIdx);
```

Paramètres

Paramètre	type	description
LibIdx	size_t	l'index de l'instance tel que retourné par GmlInit()

4.29 GmlUploadParameters

Cette fonction copie le contenu de la structure de paramètres, définie avec `GmlNewParameters`, de la mémoire du CPU hôte vers celle du GPU afin que celui-ci puisse utiliser les paramètres définis par l'utilisateur avant le lancement du kernel.

Syntaxe

```
GmlUploadParameters(LibIdx);
```

5 Glossaire

5.1 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.

5.2 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.

5.3 GPU

Graphic Processing Unit, aussi appelé GPGPU (pour General Purpose), c'est une puce adaptée aux calculs géométriques et vectoriels qui, comme son 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).

5.4 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.

5.5 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. Son objectif est d'exposer clairement l'indépendance entre les calculs afin de mieux les répartir sur les différentes unités.

5.6 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.*