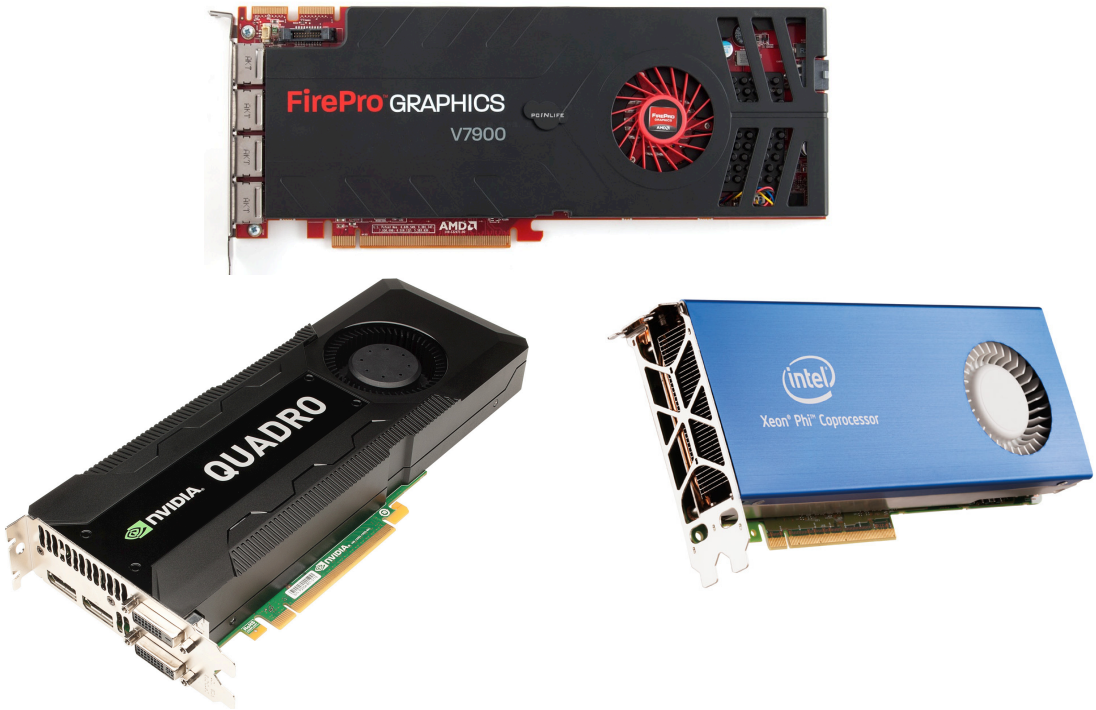
Une aide à la programmation sur GPU
pour le calcul scientifique

La librairie GMlib



Loïc MARÉCHAL / INRIA, Projet Gamma

Avril 2020

Document v2.01 Librairie v3.19

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 développement 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 tire non seulement parti des GPU des trois principaux constructeurs, AMD, Intel et Nvidia, mais aussi des 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 coeurs 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 des programmeurs. Il est très important de noter qu'en matière de code sur GPU, c'est le stockage, transfert et 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 le plus libre possible sur la partie code 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 : les 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 (OpenGL, Direct 3D), puis de fournir un code assez succins, appelé shader, à appliquer sur chaque entité graphique sélectionnée. Il est par exemple courant de fournir un ensemble de sommets, de triangles, de vecteurs normaux aux sommets, 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 et la position à l'écran de chaque pixel du triangle.

Une telle approche a été reprise pour la `GMlib`, mais ici les textures, projections, et effets graphiques sont remplacés par des tétraèdres, arrêtes, champs de solutions physiques, et tous les accès indirects possibles entre ces entités. Les types de données de maillage actuellement proposées sont : sommets, arrêtes, triangles, quadrilatères, tétraèdres, pyramides, prismes et hexaèdres ainsi que les liens implicitent qui les relient entre eux (boules, coquille, voisin) et qui sont automatiquement construits par la librairie. De plus, des types de données libres (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 décompose en quatre étapes : initialisations, renseignements des données, compilation des kernels 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 le kernel 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, c'est une phase de description des données.

Renseignement des données : L'utilisateur doit boucler sur chaque sommet, élément, ou autre donné précédemment déclarés pour en transmettre le contenue à la `GMlib`, qui elle-même se chargera de les transmettre sur la carte graphique au moment et sous la 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, puis le code source du code 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, ce qui lui permet de fonctionner sur l'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 lancer l'exécution de chaque kernel, autant de fois que nécessaire pour converger un calcul et en ne modifiant que quelques paramètres globaux afin de contrôler ledit calcul (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`, 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

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. Logiquement, 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, liens topologiques),

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

Toutes ces phases sont exécutées par le CPU hôte et seule les kernels tournent sur GPU et s'en trouveront donc accélérées. Dire d'un code qu'il "tourne sur GPU" est abus de langage, car les entrées-sorties, initialisation, compilation et post-traitement sont exécutés sur CPU et l'on doit parler de logiciel hybride CPU/GPU. Il est donc important d'optimiser toutes les phases d'initialisation, 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 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.

Ensuite il faut 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 utiliser ses propres données en mémoire CPU (tableaux, structures, objets, etc.), mais doit obligatoirement les déclarer et le 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 les processus, on va déclarer et renseigner les sommets d'un maillage que l'on vient de lire et de ranger 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 et 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 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 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 + vitesse * direction;
```

Ce code, on ne peux plus succin, amène de nombreuses remarques :
— il n'y a pas de boucle itérant de 1 à NmbVer,
— il ne comporte aucune définition préalable des variables utilisée,
— il n'y a pas d'accès aux tableaux des sommets et données associées, mais seulement des scalaires,
— 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. Nous avions demandé l'accès aux coordonnées des sommets (un vecteur 3-D), à la vitesse (scalaire) et au vecteur de déplacement (vecteur 3-D) et la librairie a donc défini d'elle-même les variables du type adéquat pour recevoir toutes les données demandées de 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), 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 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 fait remplir 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 il va effectuer les calculs.

Stockage du résultat en mémoire : de la même manière qu'avec les lectures, toutes les variables étant définie 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 étant 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 (8,16,32 ou 64 bits) et réelles (16,32 ou 64 bits) et utilise la technique du surtypage d'opérateur pour adapter des 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.

Les types de données

Kernels a accès directs

Kernels a accès indirects

1.3 Programmation en OpenCL

2 Utilisation

2.1 Installation et compilation

2.2 Initialisation

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

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

3 Liste des commandes

3.1 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 vouent 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.

3.2 GmlCompileKernel

3.3 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()

Commentaires

3.4 GmlDebugOn

Active le mode de débogage qui permet d'afficher l'intégralité des sources compilés 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

Commentaires

3.5 GmlExtractEdges

Non implémenté.

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	

Commentaires

3.6 GmlExtractFaces

Non implémenté.

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	

Commentaires

3.7 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.

3.8 GmlGetDataLine

Permets 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.

3.9 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	

3.10 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	

3.11 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` !

3.12 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.

3.13 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 ! La procédure est bloquante et ne rends la main qu'a la fin de l'exécution, mais cette une limitation imposée par la version actuelle de la `GMlib` qui sera levée par la suite afin de permettre des exécutions en *pipeline*.

Syntaxe

```
temps = 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
temps	double	si ¡ 0 : code d'erreur, si ¿=0 : temps d'exécution en secondes

3.14 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.

3.15 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.

3.16 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.

3.17 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

3.18 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 sera automatiquement synchronisée, avant chaque exécution de kernel, la structure sera recopiée de la mémoire CPU vers le GPU afin qu'il puisse y lire des paramètres et à la fin, la copie en mémoire GPU sera rapatriée sur le CPU afin d'y lire d'éventuels résultats de calculs ou codes d'erreurs.

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, comme elle sera 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.

3.19 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 choisi 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.

3.20 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

```
temps = 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
temps	double	temps d'exécution en secondes

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.

3.21 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.

3.22 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.

3.23 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).

3.24 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 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 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).

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. Son 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.*