12 avril 2011

OPENCL

Notes d'exploration

- Patrice Bonneau -

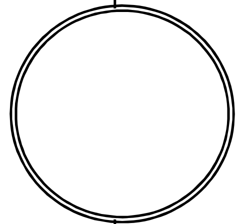

Table des matières

1. Présentation d'OpenCL 1

2. Systèmes d'exploitation supportés 2
   a. Windows 2
   b. Linux 2

3. Compilateurs supportés 2
   a. Windows 2
   b. Linux 2

4. Cartes graphiques supportées 2
   a. ATI 2
   b. nVidia 3

5. Installation d'OpenCL avec un GPU ATI 4
   a. Environnement Windows 7 4
   b. Environnement Linux 4
   c. Environnement Mac 5

6. Installation d'OpenCL avec un GPU nVidia 6
   a. Environnement Windows 7 6
   b. Environnement Linux 6
   c. Environnement Mac 6

7. Configuration de Code::Blocks 6

8. Compilation avec gcc/g++ 6

9. Utilisation d'OpenCL 7
   a. Les bases 7
   b. Les conditions particulières 7
   c. Documentation et références OpenCL 7

   Les espaces d'adresses 8
   Les types de variables 8
   Les conversions de types 9
   Les opérateurs 10
   Les fonctions incorporées 10
   Les restrictions 10
   À faire attention 10

10. Exemple d'utilisation d'OpenCL 11
   a. Affichage des ressources de l'environnement de calcul 11
   b. Mettre un vecteur au carré 11
   c. Paralléliser une triple boucle imbriquée 12

Annexe A – Les sources des exemples ... 14
Environnement de calcul ... 14
Vecteur au carré ... 15
Triple boucle imbriquée ... 19

1. Présentation d'OpenCL [1]

OpenCL (Open Computing Language) est la combinaison d'un API et d'un langage de programmation dérivé du C, proposé comme un standard ouvert par le Khronos Group. OpenCL est conçu pour programmer des systèmes parallèles hétérogènes comprenant par exemple à la fois un CPU multicoeur et un GPU. OpenCL distingue le processeur hôte (processeur central faisant office de chef d'orchestre) des devices (CPU, GPU, ou autre) dont la mission est d'exécuter des noyaux de calcul intensifs. OpenCL distingue donc l'application (écrite en C) tournant sur le processeur hôte et qui va appeler l'API OpenCL, des kernels qui sont programmés en OpenCL-C et dont la vocation est d'être exécuté sur les devices. OpenCL permet d'exprimer du parallélisme de tâche et du parallélisme de donnée (SPMD - Single Program Multiple Data) de manière hiérarchique. Un graphe de tâche peut être créé dynamiquement via l'API OpenCL. Chaque tâche peut être soit une unique instance d'un kernel (appelée task), soit de multiples instances d'un même kernel (appelé NDRange). Les NDRanges peuvent être de 1, 2 ou 3 dimensions. Chaque instance de kernel appartenant à un NDRange est appelée work-item. Le NDrange peut lui même être structuré en work-groups, ce qui permet aux work-items à l'intérieur des work-groups de partager des données et de se synchroniser via des barrières.

Si parmi certains de ses objectifs techniques, OpenCL semble se rapprocher de C pour CUDA, modèle de programmation propriétaire de la société Nvidia, OpenCL a des objectifs plus larges car n'étant pas uniquement dédié aux GPU. Dans le monde du calcul performance ou du jeu, OpenCL permettra de tirer parti de la puissance des processeurs graphiques, des CPU multicœurs ou d'autres systèmes de calcul intensifs tel le CELL d'IBM, qui équipe notamment la Playstation 3 de Sony. Dans le monde des systèmes embarqués sur puce (SoC), tel qu'on les trouve dans les smartphones, OpenCL permettra l'accès, via une infrastructure de programmation unique, au processeur central, ainsi qu'aux différents sous-systèmes multimédias embarqués (GPU, DSP, computing array ou autres).

OpenCL a été initialement conçu par Apple (qui en possède les droits d'auteur), et affiné dans le cadre d'une collaboration avec AMD, Intel et Nvidia. Apple soumet d'abord sa proposition initiale au Khronos Group. Le 16 juin 2008, le Khronos Compute Working Group est formé, comprenant des représentants des fabricants de matériel informatique et de logiciels.

OpenCL est intégré dans Mac OS X 10.6 (Snow Leopard). AMD décide de supporter OpenCL et DirectX 11 plutôt que Close to Metal dans son framework Stream SDK. RapidMind annonce l'adoption de OpenCL sous sa plate-forme de développement, afin de supporter les processeurs graphiques de plusieurs fabricants avec une interface unique. Nvidia confirme également le 9 décembre 2008 le support complet de la spécification 1.0 dans son GPU Computing Toolkit.

[1] Source : WIKIPEDIA. OpenCL [En ligne] (Consulté le 08 avril 2011)

2. Systèmes d’exploitation supportés

a. Windows

- Windows XP SP3 (32 bits), SP2 (64 bits)
- Windows Vista SP1 (32/64 bits)
- Windows 7 (32/64 bits)

b. Linux

(Pour les versions ci-dessous et ultérieures)

- openSUSE 11.1 (32/64 bits)
- Ubuntu 9.10 (32/64 bits)
- Red Hat Enterprise Linux 5.3 (32/64 bits)

3. Compilateurs supportés

a. Windows

- Intel® C Compiler (ICC) 11.x
- Microsoft Visual Studio 2010 Professional Edition
- Microsoft Visual Studio 2008 Professional Edition
- Minimalist GNU for Windows (MinGW) [GCC 4.4]

b. Linux

- GNU Compiler Collection (GCC) 4.1 or later
- Intel® C Compiler (ICC) 11.x

4. Cartes graphiques supportées

Voici la liste des cartes graphiques supportées par OpenCL. Cette liste, divisée par constructeur, a été mise à jour au printemps 2011. À noter que si vous ne possédez pas une de ces cartes, ça ne pose pas de problème. OpenCL peut fonctionner en utilisant uniquement les cœurs de votre processeur. Cependant, même si elle n’est pas supportée, il faut télécharger OpenCL via le constructeur de votre carte graphique.

a. ATI

AMD Radeon™ HD Graphics	6900 Series (6970, 6950) 6800 Series (6870, 6850) 6600 Series (6670) 6500 Series (6570) 6400 Series (6450)

ATI Radeon™ HD Graphics	5900 Series (59702) 5800 Series (5870, 5850, 5830) 5700 Series (5770, 5750) 5600 Series (5670) 5500 Series (5570) 5400 Series (5450) 4800 Series (4850, 4870)	4850 Utilisée pour mes tests.
ATI FirePro™ Graphics	V8800 V7800 V5800 V4800 V3800	
ATI Mobility Radeon™ HD	5800 Series (5870, 5850, 5830) 5700 Series (5770, 5750, 5730) 5600 Series (5650) 5400 Series (5470, 5450, 5430)	5750 Utilisée pour mes tests.
ATI Mobility FirePro™	M7820 M5800	

b. nVidia

GeForce GTX	400 Series (480, 470) 200 Series (295, 285, 280, 260) 200M Series (285M, 280M, 260M)
GeForce 9800	GX2, GTX, GTX+, GT
GeForce 9800M	GTX, GT, GTS
GeForce 9700M	GT
GeForce 9600	GSO, GT
GeForce 9600M	GT, GS
GeForce 9650M	GS
GeForce 9500	GT, GS
GeForce 9500M	G
GeForce 9400	mGPU
GeForce 9300	mGPU
GeForce 9300M	G
GeForce 8800	GTX, GTS
GeForce 8800M	Ultra, GTX, GTS 512, GT, GS, GTS
GeForce 8700M	GT
GeForce 8600	GTS, GT
GeForce 8600M	GT, GS
GeForce 8500	GT
GeForce 8400	GS
GeForce 8400M	GT, GS

GeForce 8300	mGPU
GeForce 8200	mGPU
GeForce 8100	mGPU
GeForce GTS	300M Series (360M, 350M) 200 Series (250) 200M Series (260M, 250M) 100 Series (150)
GeForce GT	300M Series (335M, 330M, 325M) 200 Series (240, 220) 200M Series (240M, 230M) 100 Series (130, 120)
GeForce	210, 310M, 305, G100, G210M, G110M

5. Installation d'OpenCL avec un GPU ATI

a. Environnement Windows 7

(testé avec un processeur Intel quad core 64 bits et AMD phenom II (4 cœurs) 64 bits)

1. Télécharger puis exécuter "**Catalyst Software Suite**" à l'adresse : http://support.amd.com/fr/gpudownload/Pages/index.aspx.

   Ceci installera les derniers pilotes de même que le SDK qui contient OpenCL. L'installation créera les répertoires "C:\Program Files (x86)\ATI Stream\..."

b. Environnement Linux[2]

(testé avec un processeur AMD 64 bits sous Ubuntu 10.10)

1. Télécharger puis exécuter "**AMD Catalyst™ 11.3 Proprietary Linux x86 Display Driver**" à l'adresse : http://support.amd.com/fr/gpudownload/Pages/index.aspx.

   Ceci installera les derniers pilotes uniquement. À noter qu'il est possible que l'on doive d'abord lui donner les droits d'exécution avec "chmod +x …".

2. Télécharger le SDK (incluant OpenCL) à l'adresse : http://developer.amd.com/gpu/AMDAPPSDK/downloads/Pages/default.aspx.

   L'enregistrer et le décompresser sur le bureau. Le dossier "**ati-stream-sdk-v2.3-lnx64**" sera ainsi créé. Ici c'est la version 64 bits qui est utilisée.

3. Créer le dossier "**/usr/local/opencl**".

[2] Les étapes d'installation ATI/Linux ont été recueillies en grande partie sur http://crack-wifi.com/.

4. Copier le <u>contenu</u> du dossier “**ati-stream-sdk-v2.3-lnx64**” dans le dossier créé ci-dessus.

5. Ajouter à votre fichier “ ~/**.bashrc** ” les lignes suivantes :
   a. `export ATISTREAMSDKROOT=/usr/local/opencl/`
   b. `export ATISTREAMSDKSAMPLESROOT=/usr/local/opencl/`
   c. `export LD_LIBRARY_PATH=$LD_LIBRARY_PATH :$ATISTREAMSDKROOT/lib/x86_64/`
   d. `export LIBRARY_PATH=$LIBRARY_PATH:$ATISTREAMSDKROOT/lib/x86_64/`
   e. `export C_INCLUDE_PATH= $C_INCLUDE_PATH: $ATISTREAMSDKROOT/include/`
   f. `export CPLUS_INCLUDE_PATH=$CPLUS_INCLUDE_PATH: $ATISTREAMSDKROOT/include/`

   À noter que pour un système de 32 bits, il faut remplacer les répertoires “x86_64” par “x86”.

6. Exécuter votre fichier **.bashrc** afin d’activer les nouveaux ajouts.

7. Créer les répertoires “**/usr/lib/OpenCL**” et “**/usr/lib/OpenCL/vendors**”.

8. Créer un lien dans le dossier “**vendors**” vers le fichier “**libatiocl64.so**”.
   a. `sudo ln -sf /usr/local/opencl/lib/x86/libatiocl32.so /usr/lib/OpenCL/vendors/libatiocl32.so`

9. Aller dans le répertoire “**/usr/local/opencl**” et exécuter un “**make”.**

10. Créer un lien dans le dossier “**/usr/lib**” vers le fichier “**libOpenCL.so.so**”.
    a. `sudo ln -sf /usr/local/opencl/lib/x86/libOpenCL.so /usr/lib/libOpenCL.so`

11. Copier le fichier “**/usr/local/opencl/icd-registration.tgz**” dans le répertoire racine (/), puis le décompresser.
    a. `tar xfz icd-registration.tgz`

    Ceci ajoutera deux fichiers dans le répertoire “/etc/OpenCL/vendors”.

12. Tester l’installation
    a. `cd /usr/local/opencl/samples/opencl/`
    b. `sudo make`
    c. `cd bin/x86_64/`
    d. `./HelloCL`

c. Environnement Mac
(non testé)

Aucune installation n’est nécessaire pour Mac. OpenCL est intégré dans Mac OS X 10.6 (Snow Leopard).

6. Installation d'OpenCL avec un GPU nVidia

(aucune installation avec nVidia n'a été testée)

a. Environnement Windows 7

(testé avec **...**)

À faire...

b. Environnement Linux

(testé avec **...**)

À faire...

c. Environnement Mac

(non testé)

Aucune installation n'est nécessaire pour Mac. OpenCL est intégré dans Mac OS X 10.6 (Snow Leopard).

7. Configuration de Code::Blocks

1) Configurer votre IDE pour lier les librairies. Voici la procédure pour Code::Blocks.
   a. Créer un projet.
   b. Dans le menu, cliquer sur : **Projet -> Options de construction…**
   c. Dans la nouvelle fenêtre, cliquer sur l'onglet : **Linker settings**
   d. Ajouter : **C:\Program Files (x86)\ATI Stream\lib\x86\OpenCL.lib**
   e. Cliquer sur l'onglet : **Search directories**
   f. Ajouter : **C:\Program Files (x86)\ATI Stream\include**

8. Compilation avec gcc/g++

```
g++ source.cpp -o nomSortie –lOpenCL
```

9. *Utilisation d'OpenCL*

a. Les bases

À première vue, le fonctionnement d'OpenCL peut paraitre fastidieux. Mais lorsqu'on y regarde de plus prêt, on se rend compte qu'il n'en est rien. Pour paralléliser une partie d'un code séquentiel, il suffit de la retirer du code source et la remplacer par du code OpenCL. Le travail le plus difficile sera le portage du code séquentiel vers le code parallèle, soit l'étape 5 des 10 étapes que voici.

1) Création d'une plateforme de travail OpenCL qui contiendra les objets suivants.
2) Définition des périphériques disponibles (CPU/GPU).
3) Création d'un contexte de travail.
4) Création d'une file d'exécution.
5) Construction et compilation du programme contenant le code à paralléliser.
6) Création de tampons qui serviront de liens entre le code séquentiel et le code parallèle.
7) Création du noyau qui contiendra le programme construit à l'étape 5.
8) Mise en file d'exécution du noyau créé ci-dessus (lancement instantané des calculs).
9) Récupération des résultats depuis les tampons d'échange.
10) Libération des ressources.

Le code à exécuter en parallèle est généralement fourni à OpenCL de deux manières. Soit à l'intérieur même du code source, contenu dans un « const char* » qui sera lu lors de l'étape 5. Soit dans un fichier séparé, qui sera lu lors de cette même étape.

b. Les conditions particulières

Certains codes sont parallélisable seulement sur les CPU. En effet, les cœurs d'un CPU sont génériques, c'est-à-dire qu'ils fonctionnent comme un processeur à part entière. Par contre, les GPU ne sont pas adaptés, par exemple, pour gérer des chaines de caractères. Ils sont spécialisés pour faire des opérations graphiques de bas niveau, incluant les calculs mathématiques.

Autre point important, la distinction entre code séquentiel et code parallèle. Pour être bref, les deux ne se mélangent pas. Lorsque le code séquentiel s'exécute et arrive sur du code à paralléliser, il s'arrête et ne reprend son exécution que lorsque le code parallèle a terminé son exécution. De plus, le code parallèle doit contenir tout ce dont il a besoin pour son exécution. En d'autres mots, il n'accède pas aux variables et classes du code séquentiel.

c. Documentation et références OpenCL

OpenCL offre de bonnes références sur son site web à l'adresse : http://www.khronos.org/registry/cl/sdk/1.0/docs/man/xhtml/, dont nous apportions ici quelques notions préliminaires.

Les espaces d'adresses

Les espaces d'adresses, comme dans tout système parallèle, peuvent être définis comme étant globaux ou locaux. OpenCL implémente quatre types d'espaces d'adresses. On peut en voir un exemple dans le code « Vecteur au carré » à l'Annexe A – Les sources des exemples, lors de la conception de la fonction « `maFonctionAuCarre` » dans la section de déclaration des variables en début de code. Les espaces d'adresses débutent par deux barres de soulignement (underscore), et sont les suivantes :

- __global
- __local
- __constant
- __private

Les types de variables

Les types de variables dans OpenCL sont essentiellement les mêmes que pour le langage C. Cependant, pour des raisons de portabilité, OpenCL a redéfini ses propres types. Généralement, il suffit de faire précéder les types du langage C par le préfixe « `cl_` ». Il faut prendre note que le type booléen n'existe pas pour OpenCL et que les types du langage C sont supportés partout, sauf à l'intérieur des fonctions du « __kernel ». Voici quelques exemples :

char / unsigned char	→	cl_char / cl_uchar
short / unsigned short	→	cl_int / cl_uint
Int / unsigned int	→	cl_int / cl_uint
long / unsigned long	→	cl_long / cl_ulong
float	→	cl_float
double	→	cl_double

Il faut également noter qu'OpenCL possède quatre autres types de variables qui lui sont propres, dont certaines ont été conçu pour le traitement d'image. Les voici :

image2d_t & image3d_t	→	Utilisé par les fonctions de lectures et d'écritures des images. Voir : [http://www.khronos.org/ ... /imageFunctions.html](http://www.khronos.org/). Ils doivent être précédés par « read_only » ou « write_only ». On ne peut lire ET écrire sur une même image.
sampler_t	→	Un type d'échantillonnage.
event_t	→	Peut être utilisé pour identifier des copies asynchrones entre une mémoire globale et locale. N'est pas admis pour les arguments d'un noyau (__kernel).

La dimension d'un tableau est définie en ajoutant au type un suffixe correspondant à la dimension désirée. Par exemple, un tableau de 4 entiers se déclare : « int4 nomVariable ».

Les conversions de types

Les conversions explicites se font en utilisant la fonction « convert_*T*() », où « *T* » prend la valeur du type désiré. Voici un exemple :

```
Char toto = 'z';
int tata = convert_int(toto);
```

Il est possible de rencontrer des problèmes lorsque l'on veut convertir une variable dont la valeur contenue dans le type source n'entre pas dans les limites du type cible, par exemple une variable de type « `unsigned int` » qui contient 3 milliards qui doit être converti dans un « `int` ». Pour un tel exemple, on peut utiliser le mode de conversion saturée. Dans ce cas, la valeur converti prendra la valeur la plus proche permise dans le type cible. Pour ce faire, on utilise le suffixe « _sat », comme dans l'exemple ci-dessous.

```
Unsigned int toto = 3 000 000 000;
int tata = convert_int_sat(toto);
```

Dans le cas où l'on voudrait convertir un nombre réel en un nombre entier, il est possible de définir le mode d'arrondissement en ajoutant un suffixe à la fonction « convert_T() ». Le tableau ci-dessous montre les quatre modes possibles, suivi d'un exemple.

_rte	Round to nearest even → Arrondi au plus proche entier (3,5 = 4)
_rtz **(Par défaut)**	Round towards zero → Arrondi vers zéro (3,5 = 3 & -3,5 = -3)
_rtp	Round toward positive infinity → Arrondi vers l'infini positif
_rtn	Round toward negative infinity → Arrondi vers l'infini négatif

```
float toto = 0.1234f;
int tata = convert_int_rte(toto);
```

ou

```
float toto = 3 000 000 000.1234f;
int tata = convert_int_sat_rte(toto);
```

Il existe une autre façon d'effectuer des conversions. Il s'agit de la fonction « as_*T*() », où « *T* » prend la valeur du type désiré. Voici quelques exemples.

```
float toto = 12345.3f;
unsigned int tata = as_uint(toto);
```

```
float4 toto = (float4)(1.1f, 2.2f, 3.3f, 4.4f);
int4 tata = as_int4(toto);
```

Les opérateurs

Les opérateurs, qu'ils soient logiques ou arithmétiques, sont les mêmes que pour le langage C.

Les fonctions incorporées

OpenCL inclut une grande gamme de fonctions. On y retrouve des fonctions de mathématiques (*sin, cos, log, sqrt, pow*, …), de géométrie (*cross, distance, length, normalize*, …), de traitement d'images (*read_nnn, write_nnn, get_nnn*, …), de relation (*isequal, isgreater, isnan, isordered*, …) de même que des fonctions communes (*min, max, degrees, radians, sign*, …). Ainsi, nous n'avons pas besoin d'inclure des bibliothèques telles que « math.h » dans notre code source. Ceci est très utile dans les fonctions parallélisées qui sont passées dans un noyau (__kernel).

Les restrictions

- Un pointeur déclaré avec un qualificatif __constant, __local ou __global ne peut être assigné qu'à un espace mémoire de même type.
- Les pointeurs de fonctions ne sont pas permis.
- Les variables de type image2d_t ou image3d_t ne peuvent être spécifiées que comme argument d'une fonction du noyau (__kernel).
- Les entêtes (headers) du standard C99, tels que « assert.h », « stdio.h », « stdlib.h », « string.h », etc. ne peuvent pas être utilisés à l'intérieur du code des fonctions parallélisées. Exemple, voir « `maFonctionAuCarre` » en annexe.
- La récursivité n'est pas supportée.

À faire attention

La fonction « `clCreateProgramWithSource()` » a pour troisième paramètre le nom de la variable « char » qui contient la fonction à paralléliser, alors que la fonction « `clCreateKernel()` » a pour second paramètre le nom de la fonction écrite à l'intérieur de la variable « char ». Voir l'exemple ci-dessous.

```
const char *maFonctionTripleBoucle = {
    "__kernel void tripleBoucle(__global int *output)\n"
    "{\n"
    "    // Le code ici...\n"
    "}\n"
};
```

```
clCreateProgramWithSource (…, (const char**)&maFonctionTripleBoucle, …);
```

```
clCreateKernel (…, "tripleBoucle", …);
```

10. Exemple d'utilisation d'OpenCL

Dans le but de simplifier les exemples, aucune gestion d'erreurs ne sera faite. Pour faire la gestion d'erreur, il suffit de faire une vérification avec la constante « `CL_SUCCESS` », telle que : « `if (codeErreur != CL_SUCCESS) { … }` ».

a. Affichage des ressources de l'environnement de calcul

Dans cet exemple, seules les étapes 1 et 2 des étapes définies à la section 9 (Utilisation d'OpenCL) sont utilisées. Le code source de cet exemple est disponible à l'Annexe A – Les sources des exemples, sous la rubrique "Environnement de calcul". L'algorithme est le suivant :

- Création de la plateforme de travail → clGetPlatformIDs (…)
- Rechercher tous les PÉRIPHÉRIQUES → clGetDeviceIDs (…, CL_DEVICE_TYPE_ALL, …)
- Afficher le nombre de périphériques → printf()
- Rechercher les CPU dans le système → clGetDeviceIDs (…, CL_DEVICE_TYPE_CPU, …)
- Afficher le nombre de CPU → printf()
- Rechercher les GPU dans le système → clGetDeviceIDs (…, CL_DEVICE_TYPE_ GPU, …)
- Si aucun GPU n'est trouvé, alors…
  - Afficher "Aucun GPU" → printf()
- Sinon…
  - Afficher le nombre de GPU → printf()

b. Mettre un vecteur au carré

Cet exemple est complet. Bien qu'il travaille sur un petit vecteur, toutes les étapes de construction du code OpenCL y sont représentées. Le code source de cet exemple est disponible à l'Annexe A – Les sources des exemples, sous la rubrique "Vecteur au carré". L'algorithme est le suivant :

- Création de la plateforme de travail → clGetPlatformIDs (…)
- Rechercher tous les PÉRIPHÉRIQUES → clGetDeviceIDs (…, CL_DEVICE_TYPE_ALL, …)
- Création du contexte de travail → clCreateContext (…, &TOUS_peripheriqueID, …)
- Création de la file d'exécution → clCreateCommandQueue (contexte, …)
- Construire le programme avec la fonction auCarre → clCreateProgramWithSource (…)
- Compiler le programme "auCarre" → clBuildProgram(programme, …)
- S'il y a des erreurs de compilation, alors…
  - Afficher le journal (log) de compilation → clGetProgramBuildInfo (…)
- Association des variables de données avec le tampon d'échange → clCreateBuffer(…, varData, …)
- Construire le noyau → clCreateKernel(programme, …)
- Associer le tampon d'échanges avec les arguments de la fonction → clSetKernelArg(…, &buffer)
- Mettre le noyau dans la file d'exécution → clEnqueueNDRangeKernel(file_execution, noyau, …)
- Récupération des résultats dans le tampon → clEnqueueReadBuffer(…,buffer, …, varData)
- Affichage des résultats → printf(…, varData);
- Libération des ressources → clReleaseMemObject(object)

c. Paralléliser une triple boucle imbriquée

Cet exemple est très similaire au précédent, à l'exception qu'il montre comment construire sa fonction à paralléliser de manière à ce qu'une triple boucle puisse être divisée dans les différentes unités de calcul. Le code source de cet exemple est disponible à l'Annexe A – Les sources des exemples, sous la rubrique "Triple boucle imbriquée". L'algorithme étant identique à celui du « vecteur au carré », il ne sera pas présenté ici.

```
int output = 0;
for (int i = 0; i < 4; i++) {
    for (int j = 0; j < 8; j++) {
        for (int k = 0; k < 3; k++) {
            output += i * j * k;
        }
    }
}
```

L'exemple ci-dessus montre une triple boucle imbriquée telle que programmée en langage C++, où chaque boucle est exécutée respectivement 4, 8 et 3 fois. Mais si on veut la paralléliser, comment faire pour que chaque unité de calcul sache où nous en sommes dans l'exécution des boucles? En d'autre mots, comment suivre l'incrémentation des variables « i, j et k » dans chacune des unités de calcul? Rien de plus facile!

Pour qu'OpenCL connaisse le nombre de niveaux de notre boucle, il faut l'aviser au moment de mettre le noyau dans la file d'exécution avec la fonction « `clEnqueueNDRangeKernel()` » en définissant le « NDRange » via le cinquième paramètre. L'exemple ci-dessous, qui correspond à une boucle simple, montre la mise en file d'un noyau comportant « QTE_DONNEE » dans le premier niveau, et aucune donnée dans les second et troisième niveaux.

```
size_t dimensions_globales[] = { QTE_DONNEES, 0, 0 };

codeErreur = clEnqueueNDRangeKernel(file_execution,
                                    noyau,
                                    1,
                                    NULL,
                                    dimensions_globales,
                                    NULL,
                                    0,
                                    NULL,
                                    NULL);
```

Une fois que la file d'exécution connait le nombre de niveaux, il suffit de récupérer les numéros d'identification de chaque niveau pour un calcul donné et de les mettre dans les variables « i, j et k » dès le début de l'exécution de la fonction à paralléliser. Pour ce faire, on utilise la fonction « `get_global_id()` ». L'exemple ci-dessous montre comment la triple boucle présentée précédemment sera parallélisée. À noter que les niveaux sont numérotés de 0 à 2.

```
size_t dimensions_globales[] = { 4, 8, 3 };

codeErreur = clEnqueueNDRangeKernel(file_execution,
                                    noyau,
                                    3,
                                    NULL,
                                    dimensions_globales,
                                    NULL,
                                    0,
                                    NULL,
                                    NULL);
```

```
const char *maFonctionTripleBoucle = {
    "__kernel void tripleBoucle(__global int *output)\n"
    "{\n"
    "    int i = get_global_id(0);\n"
    "    int j = get_global_id(1);\n"
    "    int k = get_global_id(2);\n"
    "\n"
    "    output += i * j * k;"
    "}\n"
};
```

Annexe A – Les sources des exemples

Environnement de calcul

```
#include <stdlib.h>
#include <stdio.h>
#include "CL/opencl.h"

int main (int argc, const char * argv[])
{
    // Variables pour la Plateforme de travail
    cl_platform_id          plateformeID;
    cl_uint                 qtePlateformes;
    cl_int                  codeErreur;

    // Variables pour les périphériques contenant des unités de calcul
    cl_device_id            TOUS_peripheriqueID;
    cl_uint                 TOUS_qtePeripheriques;
    cl_device_id            CPU_peripheriqueID;
    cl_uint                 CPU_qtePeripheriques;
    cl_device_id            GPU_peripheriqueID;
    cl_uint                 GPU_qtePeripheriques;

    // Création de la plateforme de travail
    codeErreur = clGetPlatformIDs(1, &plateformeID, &qtePlateformes);

    // Rechercher tous les PÉRIPHÉRIQUES dans le système.
    codeErreur = clGetDeviceIDs(plateformeID,
                                CL_DEVICE_TYPE_ALL,
                                1,
                                &TOUS_peripheriqueID,
                                &TOUS_qtePeripheriques);
    printf("Il y a au total %d peripherique(s) ", TOUS_qtePeripheriques);
    printf("de calcul dans cet ordinateur.  Soit :\n");

    // Rechercher les CPU dans le système.
    codeErreur = clGetDeviceIDs(plateformeID,
                                CL_DEVICE_TYPE_CPU,
                                1,
                                &CPU_peripheriqueID,
                                &CPU_qtePeripheriques);
    printf("    - %d CPU(s)\n", CPU_qtePeripheriques);

    // Rechercher les GPU dans le système.
    codeErreur = clGetDeviceIDs(plateformeID,
                                CL_DEVICE_TYPE_GPU,
                                1,
                                &GPU_peripheriqueID,
                                &GPU_qtePeripheriques);

    if (codeErreur == CL_DEVICE_NOT_FOUND) {
        printf("    - Aucun GPU\n");
    } else {
        printf("    - %d GPU(s)\n", GPU_qtePeripheriques);
    }

    Return 0;
}
```

Vecteur au carré

```
#include <stdlib.h>
#include <stdio.h>
#include "CL/opencl.h"

#define QTE_DONNEES     20

int main (int argc, const char * argv[])
{
    // Variables pour la Plateforme de travail
    cl_platform_id              plateformeID;
    cl_uint                     qtePlateformes;
    cl_int                      codeErreur;

    // Variables pour les périphériques contenant des unités de calcul
    cl_device_id                TOUS_peripheriqueID;
    cl_uint                     TOUS_qtePeripheriques;

    // Variables pour les contextes de travail
    cl_context                  contexte;
    cl_context_properties       proprietes[3];

    // Variable pour les files d'exécution des commandes
    cl_command_queue            file_execution;

    // Variable pour la fonction à paralléliser
    cl_program                  programme;

    const char                  *maFonctionAuCarre = {
        "__kernel void auCarre(__global int *input, __global int *output)\n"
        "{\n"
        "    int id = get_global_id(0);\n"
        "    output[id] = input[id] * input[id];"
        "}\n"
    };

    // Variable pour le noyau qui exécutera
    // le programme contenant la fonction parallèle.
    cl_kernel                   noyau;

    // Variables qui contiendront les données.
    int*                        inputData;
    int*                        outputData;
    cl_mem                      input_buffer;
    cl_mem                      output_buffer;


    // Initialisation des variables de données
    inputData = (int*)malloc(QTE_DONNEES * sizeof(int));
    outputData = (int*)malloc(QTE_DONNEES * sizeof(int));

    for (int i = 0; i < QTE_DONNEES; i++) {
        inputData[i] = i;
        outputData[i] = 0;
    }
    // ---------------------------------------------------------------------
    // ----------------- Fin de la section des variables ------------------
    // ---------------------------------------------------------------------
```

```
// Création de la plateforme de travail
codeErreur = clGetPlatformIDs(1, &plateformeID, &qtePlateformes);


// Rechercher les PÉRIPHÉRIQUES dans le système, tout type confondu.
codeErreur = clGetDeviceIDs(plateformeID,
                            CL_DEVICE_TYPE_ALL,
                            1,
                            &TOUS_peripheriqueID,
                            &TOUS_qtePeripheriques);


// Création du contexte de travail
proprietes[0] = (cl_context_properties) CL_CONTEXT_PLATFORM;
proprietes[1] = (cl_context_properties) plateformeID;
proprietes[2] = 0;
contexte = clCreateContext(proprietes,
                           TOUS_qtePeripheriques,
                           &TOUS_peripheriqueID,
                           NULL,
                           NULL,
                           &codeErreur);


// Création de la file d'exécution pour le CPU
file_execution = clCreateCommandQueue(contexte,
                                      TOUS_peripheriqueID,
                                      0,
                                      &codeErreur);


// Construire le programme avec la fonction auCarre pour le CPU
programme = clCreateProgramWithSource(contexte,
                                      1,
                                      (const char**)&maFonctionAuCarre,
                                      NULL,
                                      &codeErreur);
// Compilation du programme
codeErreur = clBuildProgram(programme,
                            TOUS_qtePeripheriques,
                            &TOUS_peripheriqueID,
                            NULL,
                            NULL,
                            NULL);
if (codeErreur != CL_SUCCESS) {
   // Si erreur de compilation, alors... Affichage du LOG de compilation
    char logErreurs[4096];
    size_t longueur_Log;

    clGetProgramBuildInfo (programme,
                           TOUS_peripheriqueID,
                           CL_PROGRAM_BUILD_LOG,
                           sizeof(logErreurs),
                           logErreurs,
                           &longueur_Log);
    printf("\n[IMPRESSION DU JOURNAL DE COMPILATION]\n\n");
    printf("%s\n\n", logErreurs);
    printf("[FIN DU JOURNAL]\n\n");
```

```
    return 1;
}


// Association des variables de données avec le tampon d'échange
input_buffer = clCreateBuffer(contexte,
                              CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                              sizeof(int) * QTE_DONNEES,
                              inputData,
                              &codeErreur);

output_buffer = clCreateBuffer(contexte,
                               CL_MEM_WRITE_ONLY | CL_MEM_COPY_HOST_PTR,
                               sizeof(int) * QTE_DONNEES,
                               outputData,
                               &codeErreur);


// Construire le noyau
noyau = clCreateKernel(programme, "auCarre", &codeErreur);


// Associer les tampons d'échanges avec
// les arguments des fonctions à paralléliser
codeErreur = clSetKernelArg(noyau,
                            0,
                            sizeof(input_buffer),
                            &input_buffer);

codeErreur = clSetKernelArg(noyau,
                            1,
                            sizeof(output_buffer),
                            &output_buffer);


// Mettre le noyau dans la file d'execution
size_t dimensions_globales[] = { QTE_DONNEES, 0, 0 };
codeErreur = clEnqueueNDRangeKernel(file_execution,
                                    noyau,
                                    1,
                                    NULL,
                                    dimensions_globales,
                                    NULL,
                                    0,
                                    NULL,
                                    NULL);


// Récupération des résultats dans le tampon
clEnqueueReadBuffer(file_execution,
                    output_buffer,
                    CL_TRUE,
                    0,
                    sizeof(int) * QTE_DONNEES,
                    outputData,
                    0,
                    NULL,
                    NULL);
```

```
    // Affichage des résultats
    printf("\n\n\n    >>>>> Affichage des resultats <<<<<");

    printf("\n\nContenu de la variable --inputData--\n");
    for (int i = 0; i < QTE_DONNEES; i++) {
        printf("%d; ", inputData[i]);
    }

    printf("\n\nContenu de la variable --outputData--\n");
    printf("Fonction --auCarre--\n");
    for (int i = 0; i < QTE_DONNEES; i++) {
        printf("%d; ", outputData[i]);
    }


    // Libération des ressources
    free(inputData);
    free(outputData);

    clReleaseMemObject(input_buffer);
    clReleaseMemObject(output_buffer);
    clReleaseProgram(programme);
    clReleaseKernel(noyau);
    clReleaseCommandQueue(file_execution);
    clReleaseContext(contexte);

    return 0;
}
```

Triple boucle imbriquée

```c
#include <stdlib.h>
#include <stdio.h>
#include "CL/opencl.h"

#define QTE_DONNEES     20

int main (int argc, const char * argv[])
{
    // Variables pour la Plateforme de travail
    cl_platform_id              plateformeID;
    cl_uint                     qtePlateformes;
    cl_int                      codeErreur;

    // Variables pour les périphériques contenant des unités de calcul
    cl_device_id                TOUS_peripheriqueID;
    cl_uint                     TOUS_qtePeripheriques;

    // Variables pour les contextes de travail
    cl_context                  contexte;
    cl_context_properties       proprietes[3];

    // Variable pour les files d'exécution des commandes
    cl_command_queue            file_execution;

    // Variable pour la fonction à paralléliser
    cl_program                  programme;

    const char                  *maFonctionTripleBoucle = {
        "__kernel void tripleBoucle(__global int *output)\n"
        "{\n"
        "    int i = get_global_id(0);\n"
        "    int j = get_global_id(1);\n"
        "    int k = get_global_id(2);\n"
        "    output += i * j * k;"
        "}\n"
    };

    // Variable pour le noyau qui exécutera
    // le programme contenant la fonction parallèle.
    cl_kernel                   noyau;

    // Variables qui contiendront les données.
    int*                        outputData;
    cl_mem                      output_buffer;


    // Initialisation des variables de données
    outputData = (int*)malloc(sizeof(int));
    *outputData = 0;


    // -------------------------------------------------------------------
    // ----------------- Fin de la section des variables ------------------
    // -------------------------------------------------------------------
```

```
// Création de la plateforme de travail
codeErreur = clGetPlatformIDs(1, &plateformeID, &qtePlateformes);


// Rechercher les PÉRIPHÉRIQUES dans le système, tout type confondu.
codeErreur = clGetDeviceIDs(plateformeID,
                            CL_DEVICE_TYPE_ALL,
                            1,
                            &TOUS_peripheriqueID,
                            &TOUS_qtePeripheriques);


// Création du contexte de travail
proprietes[0] = (cl_context_properties) CL_CONTEXT_PLATFORM;
proprietes[1] = (cl_context_properties) plateformeID;
proprietes[2] = 0;
contexte = clCreateContext(proprietes,
                           TOUS_qtePeripheriques,
                           &TOUS_peripheriqueID,
                           NULL,
                           NULL,
                           &codeErreur);


// Création de la file d'exécution pour le CPU
file_execution = clCreateCommandQueue(contexte,
                                      TOUS_peripheriqueID,
                                      0,
                                      &codeErreur);


// Construire le programme avec la fonction auCarre pour le CPU
programme = clCreateProgramWithSource(contexte,
                                      1,
                                      (const char**)&maFonctionTripleBoucle,
                                      NULL,
                                      &codeErreur);
// Compilation du programme
codeErreur = clBuildProgram(programme,
                            TOUS_qtePeripheriques,
                            &TOUS_peripheriqueID,
                            NULL,
                            NULL,
                            NULL);
if (codeErreur != CL_SUCCESS) {
   // Si erreur de compilation, alors... Affichage du LOG de compilation
    char logErreurs[4096];
    size_t longueur_Log;

    clGetProgramBuildInfo (programme,
                           TOUS_peripheriqueID,
                           CL_PROGRAM_BUILD_LOG,
                           sizeof(logErreurs),
                           logErreurs,
                           &longueur_Log);
    printf("\n[IMPRESSION DU JOURNAL DE COMPILATION]\n\n");
    printf("%s\n\n", logErreurs);
    printf("[FIN DU JOURNAL]\n\n");
```

```
    return 1;
}


// Association des variables de données avec le tampon d'échange
output_buffer = clCreateBuffer(contexte,
                               CL_MEM_WRITE_ONLY | CL_MEM_COPY_HOST_PTR,
                               sizeof(int) * QTE_DONNEES,
                               outputData,
                               &codeErreur);


// Construire le noyau
noyau = clCreateKernel(programme, "tripleBoucle", &codeErreur);


// Associer les tampons d'échanges avec
// les arguments des fonctions à paralléliser
codeErreur = clSetKernelArg(noyau,
                            1,
                            sizeof(output_buffer),
                            &output_buffer);


// Mettre le noyau dans la file d'execution
size_t dimensions_globales[] = { QTE_DONNEES, 0, 0 };
codeErreur = clEnqueueNDRangeKernel(file_execution,
                                    noyau,
                                    1,
                                    NULL,
                                    dimensions_globales,
                                    NULL,
                                    0,
                                    NULL,
                                    NULL);


// Récupération des résultats dans le tampon
clEnqueueReadBuffer(file_execution,
                    output_buffer,
                    CL_TRUE,
                    0,
                    sizeof(int) * QTE_DONNEES,
                    outputData,
                    0,
                    NULL,
                    NULL);


// Affichage des résultats
printf("\n\n\n    >>>>> Affichage des resultats <<<<<");

printf("\n\nContenu de la variable --outputData--\n");
printf("Fonction --tripleBouble--\n");
for (int i = 0; i < QTE_DONNEES; i++) {
    printf("%d; ", *outputData);
}
```

```
    // Libération des ressources
    free(outputData);

    clReleaseMemObject(output_buffer);
    clReleaseProgram(programme);
    clReleaseKernel(noyau);
    clReleaseCommandQueue(file_execution);
    clReleaseContext(contexte);

    return 0;
}
```