Sommaire intermédiaire

Environnements existants

Les deux principaux (concurrents) sont :

- CUDA, langage de haut-niveau, Nvidia `http://www.nvidia.com/cudazone`, pour l'instant l'approche la plus courante,
- OpenCL, langage de haut-niveau, « starter group »AMD-ATI et Apple, puis Intel, les spécifications sont faites par le Khronos Group `http://www.khronos.org/opencl/` (comme OpenGL, OpenGL-ES ou WebGL) qui regroupent des acteurs majeurs de l'industrie.

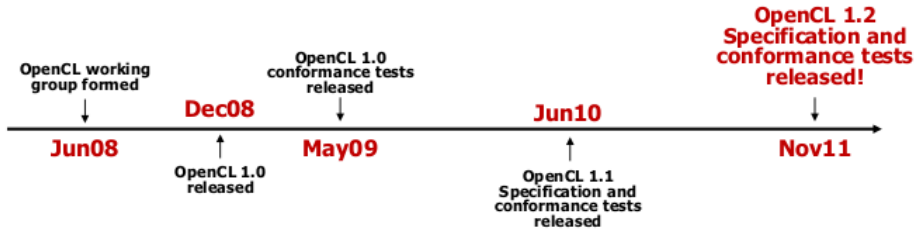
Environnements existants

Mais on peut noter également :

- DirectCompute, Microsoft, dérive de l'API DirectX qui avait déjà installé les Vertex, Fragment et Geometry Shaders, disponible seulement sur les plate-formes MsWindows.
- PGI Accelerator Compilers, qui souhaite donner une interface commune CPU/GPU et sont basés sur Nvidia Cuda `http://www.pgroup.com/resources/accel.htm`
- HMPP, Caps, `http://www.caps-entreprise.com/`, framework payant avec compilation et exécution hybride des applications,
- et quelques autres GPUSs, StarPU, QUARK, OpenMPC.

Timeline OpenCL

OpenCL est une technologie très récente. Apple lance l'idée en 2008, puis le *Khronos Compute Working Group* est créé pour mettre au point spécifications et norme.



Le principe fondamental d'OpenCL

Comme nous l'avons fait et vu dans les techniques précédentes (cf. partie n 14, slide 115), on peut très bien utiliser les BuffersObjects d'OpenGL et les shaders de GLSL pour envoyer des données au GPU, effectuer des calculs massivement parallèles (par toutes les unités de traitement de la carte graphique) et recevoir les résultats en RAM.

Traditional loops

```
void
trad_mul(int n,
         const float *a,
         const float *b,
         float *c)
{
  int i;
  for (i=0; i<n; i++)
    c[i] = a[i] * b[i];
}
```

Data Parallel OpenCL

```
kernel void
dp_mul(global const float *a,
       global const float *b,
       global float *c)
{
  int id = get_global_id(0);

  c[id] = a[id] * b[id];

} // execute over "n" work-items
```


Le principe fondamental d'OpenCL

OpenCL propose de généraliser ce fonctionnement en adoptant un procédé similaire. Le *shader* devient un *kernel*, les unités de traitement sont réparties en *working items / working groups*[21].

Traditional loops

```
void
trad_mul(int n,
         const float *a,
         const float *b,
         float *c)
{
  int i;
  for (i=0; i<n; i++)
    c[i] = a[i] * b[i];
}
```

Data Parallel OpenCL

```
kernel void
dp_mul(global const float *a,
       global const float *b,
       global float *c)
{
  int id = get_global_id(0);

  c[id] = a[id] * b[id];

} // execute over "n" work-items
```

21. réf. "OpenCL Overview", http://www.khronos.org/opencl/

Le principe fondamental d'OpenCL

Comme le code des *shaders* OpenGL, le code des *kernels* OpenCL est compilé via le driver de la carte graphique (*online compilation*). L'exécution est donc très dépendante des capacités de la machine et de la qualité du pilote.

Traditional loops

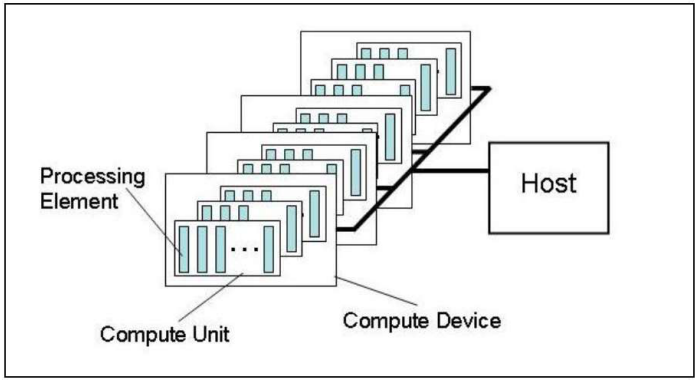
```
void
trad_mul(int n,
         const float *a,
         const float *b,
         float *c)
{
  int i;
  for (i=0; i<n; i++)
    c[i] = a[i] * b[i];
}
```

Data Parallel OpenCL

```
kernel void
dp_mul(global const float *a,
       global const float *b,
       global float *c)
{
  int id = get_global_id(0);

  c[id] = a[id] * b[id];

} // execute over "n" work-items
```

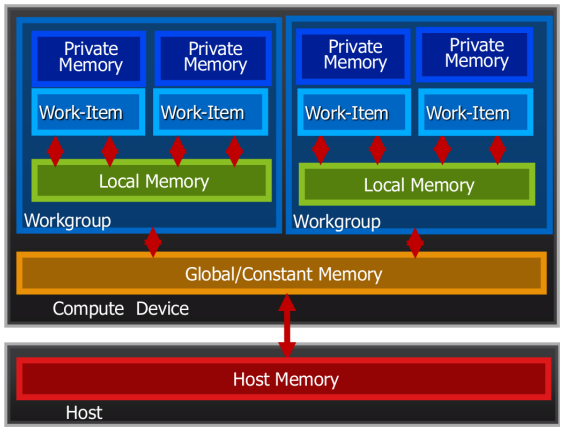

Schéma global

Pour OpenCL un périphérique de calcul (un accélérateur) est un *Compute Device*. Celui-ci est utilisable par *Compute Unit* qui eux-mêmes contiennent des *Processing Elements*.



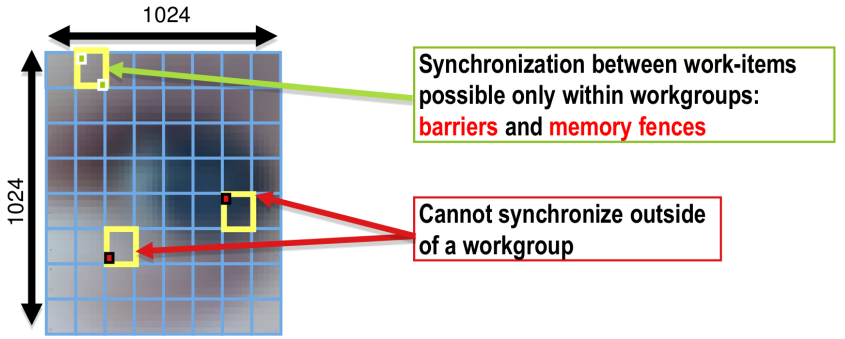
Différents types d'accès mémoire

- **privé**, pour chaque workitem,
- **local**, partagée pour un workgroup,
- **global**, visible par tous les worgroups, analogue aux données venant de BufferObjects
- **hôte**, hébergé par le CPU.



Le passage d'un emplacement mémoire à un autre est explicite. Au développeur de le spécifier !

Une application envoie donc les tâches (*jobs*) à faire à l'accélérateur. Il faut définir un contexte (partage de mémoire, taille du *Workgroup* et du *WorkItem*). Un workgroup contient un ensemble de workitems, la mémoire peut-être partagée dans un workgroup par tous les workitems.

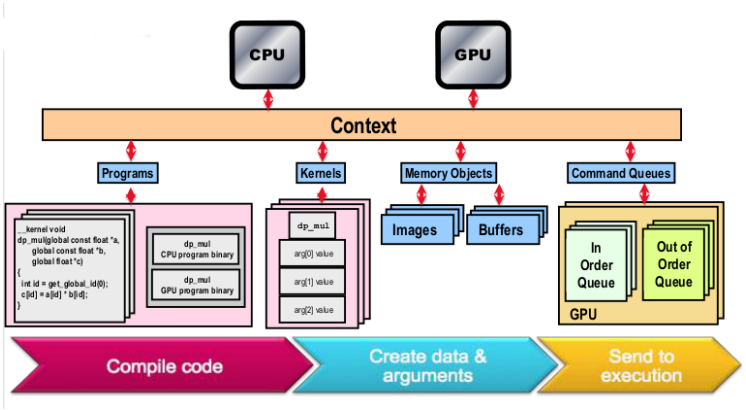


Nouvelle API = nouvelles variables, nouvelles fonctions

Le matériel disponible est géré par `Devices`, pour lequel on gère des files de tâches (`Queues`). Un ou plusieurs Devices sont plongés dans des `Contexts`.
L'exécution d'une tâche se fait par un `Kernel` qui est une instance d'un `Program`.
La mémoire est décrite par des `Buffers` (des blocs) ou des `Images` (2D, 3D).
Les exécutions déclenchent des événements (pour la synchronisation ou le profilage).

Pipeline OpenCL

Le process de compilation et d'exécution d'OpenCL est très proche de l'utilisation des shaders.



Initialisation

La première phase consiste à gérer les Devices existants et à leur assigner une file de tâches. Les devices peuvent être des CPU et/ou des GPU.

1. recensement des Device(s),

```
cl_uint num_devices;
cl_device_id devices[2];

err = clGetDeviceIDs(NULL, CL_DEVICE_TYPE_GPU, 1, &devices[0], num_devices);
err = clGetDeviceIDs(NULL, CL_DEVICE_TYPE_CPU, 1, &devices[1], &num_devices);
```

Note : les devices peuvent partager des données s'ils sont dans le même contexte. Les devices peuvent être partagés en Sub-Devices pour plus de flexibilité.

Initialisation

La première phase consiste à gérer les Devices existants et à leur assigner une file de tâches. Les devices peuvent être des CPU et/ou des GPU.

1. recensement des Device(s),

```
cl_uint num_devices;
cl_device_id devices[2];

err = clGetDeviceIDs(NULL, CL_DEVICE_TYPE_GPU, 1, &devices[0], num_devices);
err = clGetDeviceIDs(NULL, CL_DEVICE_TYPE_CPU, 1, &devices[1], &num_devices);
```

2. création des contextes,

```
cl_context context;
context = clCreateContext(0, 2, devices, NULL, NULL, &err);
```

Note : les devices peuvent partager des données s'ils sont dans le même contexte. Les devices peuvent être partagés en Sub-Devices pour plus de flexibilité.

Initialisation

La première phase consiste à gérer les Devices existants et à leur assigner une file de tâches. Les devices peuvent être des CPU et/ou des GPU.

1. recensement des Device(s),

```
cl_uint num_devices;
cl_device_id devices[2];

err = clGetDeviceIDs(NULL, CL_DEVICE_TYPE_GPU, 1, &devices[0], num_devices);
err = clGetDeviceIDs(NULL, CL_DEVICE_TYPE_CPU, 1, &devices[1], &num_devices);
```

2. création des contextes,

```
cl_context context;
context = clCreateContext(0, 2, devices, NULL, NULL, &err);
```

3. création des files (*Command queues*)

```
cl_command_queue queue_gpu, queue_cpu;
queue_gpu = clCreateCommandQueue(context, devices[0], 0, &err);
queue_cpu = clCreateCommandQueue(context, devices[1], 0, &err);
```

Note : les devices peuvent partager des données s'ils sont dans le même contexte. Les devices peuvent être partagés en Sub-Devices pour plus de flexibilité.

Obtenir des informations sur ces équipements

On peut utiliser la fonction clGetDeviceInfo(device, param_name, *value) pour obtenir les capacités de sa machine. On remplit pour cela le param_name dont on veut la valeur :

- nombre d'unité de traitements, via CL_DEVICE_MAX_COMPUTE_UNITS,
- fréquence d'horloge, via CL_DEVICE_MAX_CLOCK_FREQUENCY,
- taille de la mémoire, via CL_DEVICE_GLOBAL_MEM_SIZE,
- extensions disponibles (double précision, opérations atomiques) ...

Structures de buffers

Les données stockables en nombre par OpenCL sont :

- `Buffers`, analogue à OpenGL, ce sont simplement des tableaux. Ils peuvent être utilisés par les `Kernels` de différentes façons (tableau, `struct`, pointeurs). Ils peuvent être lus et écrits,
- `Sub-Buffers`, ce sont des régions de `Buffers`, qui permettent la phase de répartition des données (sur plusieurs `devices`),
- `Images`, 1D, 2D ou 3D, formatés ou non (RGBA ?), sonc accessibles seulement par les fonctions `read_image()` et `write_image()` des `kernels`. Une image peut être lue ou écrite par un `kernel` mais pas les deux. Comme en GLSL on passe par des `samplers` qui permettent d'adresser l'image en 2D et selon les canaux (`CL_RGBA` et `CL_FLOAT` par ex.).

Exemple de création d'Image

Voilà par exemple la création d'images en lecture et en écriture avec le format `CL_RGBA` et `CL_FLOAT` :

```
cl_image_format format;
format.image_channel_data_type = CL_FLOAT;
format.image_channel_order = CL_RGBA;

cl_mem input_image;
input_image = clCreateImage2D(context, CL_MEM_READ_ONLY, &format,
                      image_width, image_height, 0, NULL, &err);

cl_mem output_image;
output_image = clCreateImage2D(context, CL_MEM_WRITE_ONLY, &format,
                      image_width, image_height, 0, NULL, &err);
```


E/S sur les buffer

Différentes fonctions sont possibles selon le type d'opération à mener [21] :

- transfert de la mémoire du `Device` vers la mémoire Hôte
  `clEnqueueReadBuffer(queue, object, blocking, offset, size, *ptr, ...)`
- transfert des données de l'Hôte vers le `Device`
  `clEnqueueWriteBuffer(queue, object, blocking, offset, size, *ptr, ...)`
- mappage d'une zone mémoire entre le `Device` et l'Hôte
  `clEnqueueMapBuffer(queue, object, blocking, flags, offset, size, ...)`
- copie de zones de mémoire du `Device` (comme `memcopy()`)
  `clEnqueueCopyBuffer(queue, srcobj, dstobj, src_offset, dst_offset, ...)`

21. des opérations limités en taille existent : `clEnqueueXXXBufferRect`

Synchronisation

Deux modes de programmation sont possibles, qui entraîneront des difficultés de synchronisation des retards d'exécution : *in order* et *out of order*. Dans le 1er mode, on doit explicitement rendre séquentielles les exécutions des tâches. Il faut donc synchroniser les tâches entre elles, d'un `kernel` à l'autre mais aussi entre des `Devices` différents.

On se sert des événements déclenchés par les tâches pour la synchronisation. Par exemple, à la mise en file on peut associer des signaux [22] :

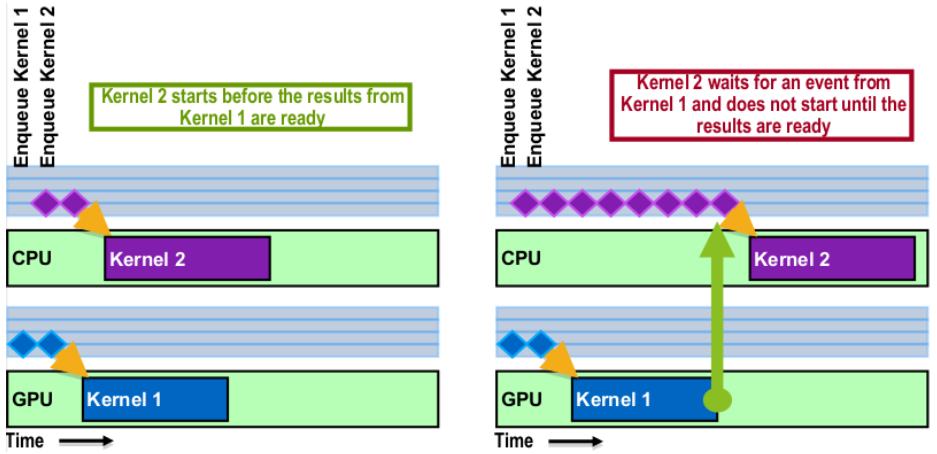
```
clEnqueue*(..., num_events_in_waitlist, *event_waitlist, *event_out)
```

L'utilisateur peut créer ses propres événements :

- `clCreateUserEvent (context, errcode_ret),`
- `clSetUserEventStatus (event, execution_status).`

22. existait en OpenGL : `enum glClientWaitSync(GLsync sync, GLbitfield flags, GLuint64 timeout)`

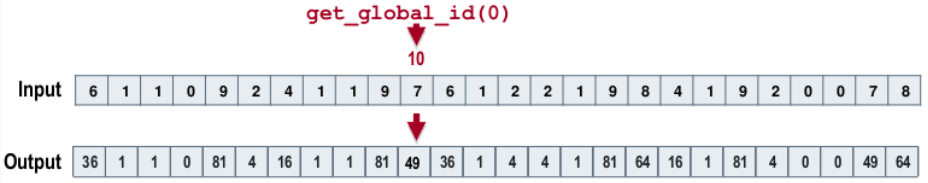
Exemple de synchronisations



Exemple de `kernel` et *global Id*

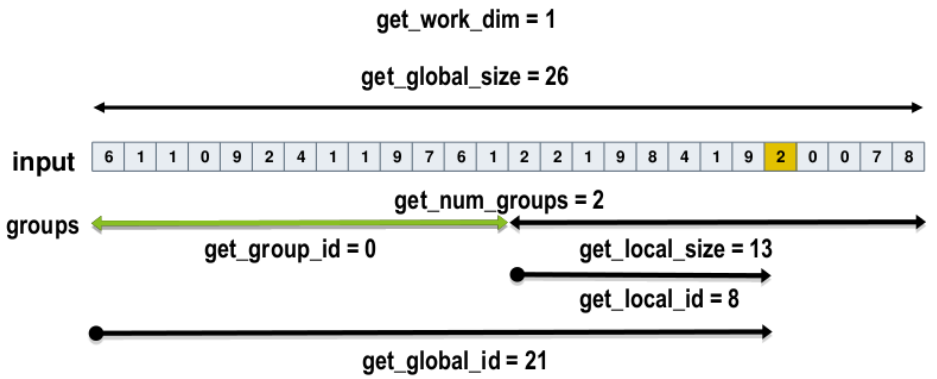
Chaque `kernel` sait où il en est dans les index globaux (géré dans le `Workgroup`), c'est le `global Id`.

```
kernel void square(global float* input, global float* output)
{
  int i = get_global_id(0);
  output[i] = input[i] * input[i];
}
```



Exemple de `kernel` et *local Id*

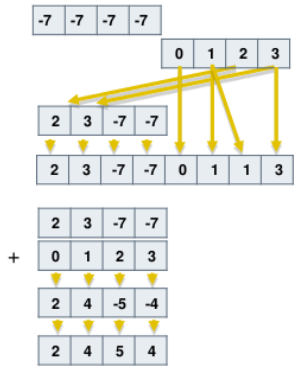
Le `kernel` sait également où il en est dans son indice local, par rapport à son `Work item`, c'est le `local Id`.



Les vecteurs, un air de déjà vu

Les opérations sur les vecteurs (`int4`, `float4`, ...) sont les mêmes qu'en GLSL.

```
int4 vi0 = (int4) -7;
int4 vi1 = (int4)(0, 1, 2, 3);

vi0.lo = vi1.hi;
int8 v8 = (int8)(vi0, vi1.s01, vi1.odd);

vi0 += vi1;
vi0 = abs(vi0);
```



Quelques autres curiosités

OpenCL a des fonctions de conversion (non implicite, non ambiguës) : `convert_uchar4()`. Ces fonctions ont des extensions pour permettre le seuillage, la saturation de la valeur retournée.

De la même façon, il permet la ré-interprétation de type, grâce à `as_XXX`, comme par exemple `as_float4()` ou `as_int4()`. Ce n'est pas une conversion mais un ré-interprétation de l'écriture binaire.

OpenCL ne supporte pas les changements d'adressage mémoire "à la volée", il faut que le transfert hôte-`device` soit explicite. Il fait donc une différence entre des variables `global`, `constant` ou non.

Pour les fonctions mathématiques, OpenCL permet 3 approximations des calculs (quantifiée), haute /moyenne /basse précision, selon les besoins en performance ou en qualité.

Impact sur les cartes graphiques

Cela entraîne des modifications des cartes graphiques (des *accélérateurs*), le maître mot est ***asynchrone*** !

- les cartes graphiques sont maintenant des *accélérateurs*, sans sortie graphique (la série Tesla chez Nvidia),
- il faut créer un modèle de mémoire unifié (RDMA chez Nvidia, GMAC chez ATI) pour accéder/échanger des données entre GPUs ou optimiser le transfert CPU $\leftrightarrow$ GPU,
- il faut mettre en place des méthode de synchronisations entre tâches (*barrier*),
- et surtout un gestionnaire de tâches haut-niveau, pour répartir les tâches (*load-balancing*) entre les GPUs / CPUs (*TM* (!) chez ATI).

Quelques éléments de comparaison avec des calculs en CPU

Les spécifications des cartes accélératrices Nvidia Kepler (génération 2012, anciennement Fermi) :

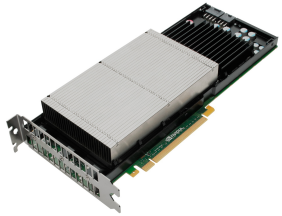
Features	Tesla K20X	Tesla K20	Tesla K10
Number and Type of GPU	1 Kepler GK110		2 Kepler GK104s
GPU Computing Applications	Seismic processing, CFD, CAE, Financial computing, Computational chemistry and Physics, Data analytics, Satellite imaging, Weather modeling		Seismic processing, signal and image processing, video analytics
Peak double precision floating point performance	1.31 Tflops	1.17 Tflops	190 Gigaflops (95 Gflops per GPU)
Peak single precision floating point performance	3.95 Tflops	3.52 Tflops	4577 Gigaflops (2288 Gflops per GPU)
Memory bandwidth (ECC off)	250 GB/sec	208 GB/sec	320 GB/sec (160 GB/sec per GPU)
Memory size (GDDR5)	6 GB	5 GB	8GB (4 GB per GPU)
CUDA cores	2688	2496	3072 (1536 per GPU)

La consommation électrique se situe entre 200 et 300W.

Quelques éléments de comparaison avec des calculs en CPU

Les spécifications des cartes accélératrices Nvidia Kepler (génération 2012, anciennement Fermi) :

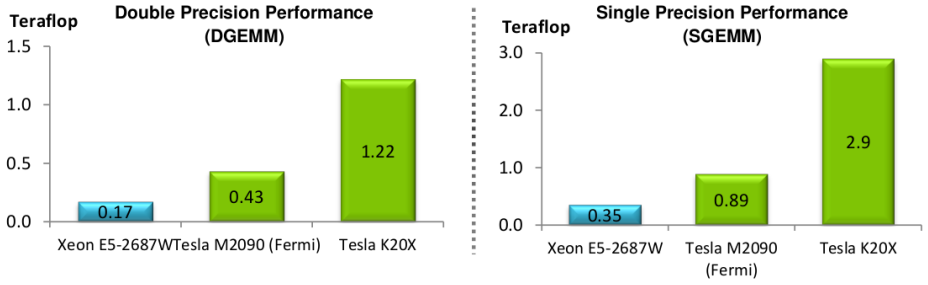
Features	Tesla K20X	Tesla K20	Tesla K10
Number and Type of GPU	1 Kepler GK110		2 Kepler GK104s
GPU Computing Applications	Seismic processing, CFD, CAE, Financial computing, Computational chemistry and Physics, Data analytics, Satellite imaging, Weather modeling		Seismic processing, signal and image processing, video analytics
Peak double precision floating point performance	1.31 Tflops	1.17 Tflops	190 Gigaflops (95 Gflops per GPU)
Peak single precision floating point performance	3.95 Tflops	3.52 Tflops	4577 Gigaflops (2288 Gflops per GPU)
Memory bandwidth (ECC off)	250 GB/sec	208 GB/sec	320 GB/sec (160 GB/sec per GPU)
Memory size (GDDR5)	6 GB	5 GB	8GB (4 GB per GPU)
CUDA cores	2688	2496	3072 (1536 per GPU)



Pour rappel, l'Intel Xeon E5-2687W à les spécifications suivantes : 3.1GHz, 8 cœurs, 16 threads, 150W.

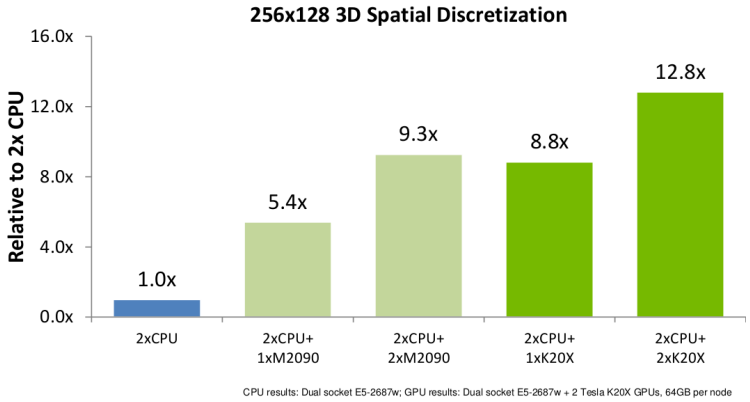
Quelques éléments de comparaison avec des calculs en CPU

Comparaison de calculs en flottants (simple et double précision) Intel Xeon vs Nvidia Tesla Fermi/Kepler :



Quelques éléments de comparaison avec des calculs en CPU

Comparaison sur de la discrétisation d'objets 3D :



Il reste des points durs

OpenCL n'est quand même pas tout rose (pour l'instant), même cela doit être un standard de généricité et de portabilité.

- l'accélérateur doit être compris, les exécutions et les transferts optimisés (un très bon pilote, un bon système, de bons bus),
- même si les interfaces de programmation ne changent pas d'un accélérateur à l'autre, les algorithmes doivent être adaptés à chaque machine (débit des bus, nombres d'unités de traitement, choix d'ordonnancement),
- hormis les problèmes d'architecture de l'hôte et de l'accélérateur, les librairies sont très dépendantes du constructeur [23], et le domaine est très versatile,
- le débogage et l'optimisation sont complexes. Il y a encore peu d'outils disponibles pour la partie `Kernel` de l'exécution (ATI a implémenté un `cl_printf`),

23. L'implémentation libre Mesa-Clover est encore balbutiante

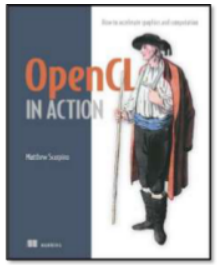
Il reste des points durs

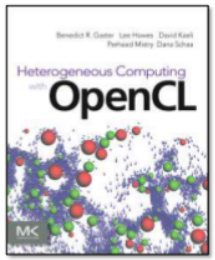
OpenCL n'est quand même pas tout rose (pour l'instant), même cela doit être un standard de généricité et de portabilité.

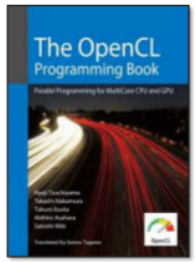
> Finalement, il faut également modifier le développeur pour qu'il s'adapte à cet environnement de développement mais aussi aux contraintes techniques de chaque accélérateur !

23. L'implémentation libre Mesa-Clover est encore balbutiante

Quelques livres de référence







Et quelques liens

- `http://gpuscience.com/`
- `http://gpgpu.org/`
- `http://www.khronos.org/opencl/`
- `http://www.openacc-standard.org/`
- `http://developer.amd.com/tools/heterogeneous-computing/`
- `https://developer.nvidia.com/gpudirect`
- `http://www.malideveloper.com/`

Après OpenCL ?

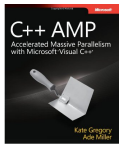
- WebGL est le portage d'OpenGL sur un client léger (grâce à HTML5), WebCL sera celui d'OpenCL (`http://www.khronos.org/webcl/`),
- besoin d'un langage de haut-niveau, non architecture-dépendant, fonctionnant en environnement hétérogène. Quelques pistes :
  - SOCL (Inria, cf *Programmation multi-accélérateurs unifiée en OpenCL, S. Henry, RenPar'20 / SympA'14 / CFSE 8, 2011*),
  - OpenCL-HLM (Khronos) en phase de spécification,
  - OpenHMPP, `http://www.openhmpp.org`, tentative de spécification ouverte, guidée par CAPS et Pathscale,
  - C++ AMP, Accelerated Massive Parallelism, Microsoft (`http://msdn.microsoft.com/en-us/library/hh265137.aspx`) ;
  - OpenACC, porté par Nvidia, avec Cray, CAPS, et The Portland Group (`http://www.openacc-standard.org/`).

Microsoft C++ AMP

Microsoft dépendant (s'appuie sur DirectX11/DirectCompute et MsVisualStudio 2012).

```
void StandardMethod() {

    int aCPP[] = {1, 2, 3, 4, 5};
    int bCPP[] = {6, 7, 8, 9, 10};
    int sumCPP[5];

    for (int idx = 0; idx < 5; idx++)
    {
        sumCPP[idx] = aCPP[idx] + bCPP[idx];
    }

    for (int idx = 0; idx < 5; idx++)
    {
        std::cout << sumCPP[idx] << "\n";
    }
}
```

```
#include <amp.h>

using namespace concurrency;

const int size = 5;

void CppAmpMethod() {
    int aCPP[] = {1, 2, 3, 4, 5};
    int bCPP[] = {6, 7, 8, 9, 10};
    int sumCPP[size];

    // Create C++ AMP objects.
    array_view<const int, 1> a(size, aCPP);
    array_view<const int, 1> b(size, bCPP);
    array_view<int, 1> sum(size, sumCPP);
    sum.discard_data();

    parallel_for_each(
        // Define the compute domain,
        //which is the set of threads that are created.
        sum.extent,
        // Define the code to run on each thread on the accelerator.
        [=](index<1> idx) restrict(amp)
    {
        sum[idx] = a[idx] + b[idx];
    }
    );

    // Print the results. The expected output is "7, 9, 11, 13, 15".
    for (int i = 0; i < size; i++) {
        std::cout << sum[i] << "\n";
    }
}
```



	0	1	2	3	4	5
0	2	2	9	7	1	4
1	4	4	8	8	3	4
2	1	5	1	2	5	2
3	6	8	3	2	7	2

idx.global=index<2>(1,3)
idx.local=index<2>(1,1)
idx.tile=index<2>(0,1)
idx.tile_origin=index<2>(0,2)
sample[idx]=8
sample(idx.y,idx.x)=8

Nvidia OpenACC

Le principe s'inspire d'OpenMP, en ajoutant des *pragmas* à la programmation classique et en laissant faire le compilateur. S'appuie sur PGI de The Portland Group qui est une surcouche de compilation, payante (et utilise CUDA).

C Version

```
#include <stdio.h>
#define N 1000000

int main(void) {
double pi = 0.0f; long i;
#pragma acc parallel loop
for (i=0; i<N; i++) {
  double t= (double)((i+0.5)/N);
  pi +=4.0/(1.0+t*t);
}
printf("pi=%16.15f\n",pi/N);
return 0;
}
```


SyCL

Khronos se rend compte (par la concurrence de Nvidia Cuda) que l'adaptation/l'adoption d'OpenCL est difficile.
Le langage OpenCL est trop bas-niveau [24]. Un nouveau standard : SyCL, propose une implémentation en C++.
Un example :

```
queue {}.submit([&](handler &h) {
    auto accA = bufA.get_access<access::mode::read>(h);
    auto accB = bufB.get_access<access::mode::write>(h);
    h.parallel_for<class myKernel>(myRange, [=](item i) {
        accA[i] = accB[i] + 1;
    });
});
```

24. Vous aimez Vulkan ?

OpenMP est à suivre...

Comme OpenMP serait la spécification des accélérateurs logiciels, avec l'utilisation des multi-cœurs, comme les Intel Xeon Phi [25].

On observe une convergence OpenMP / GPGPU :
https ://www.openmp.org/updates/openmp-accelerator-support-gpus/

TriSycl [26] est une bibliothèque open-source (qui permet de valider la norme SyCL de Khronos). Elle utilise OpenMP pour la gestion des tâches.

25. https://en.wikipedia.org/wiki/Xeon_Phi
26. https://github.com/triSYCL/triSYCL

Pour finir en beauté

Peut-être qu'un dessin vaut mieux que de longs discours pour expliquer la puissance des GPU [27]...

27. http://www.nvidia.com/content/nvision2008/art_science/index.html