Autres Outils pour le GPGPU

Xavier JUVIGNY

ONERA

March 1, 2025

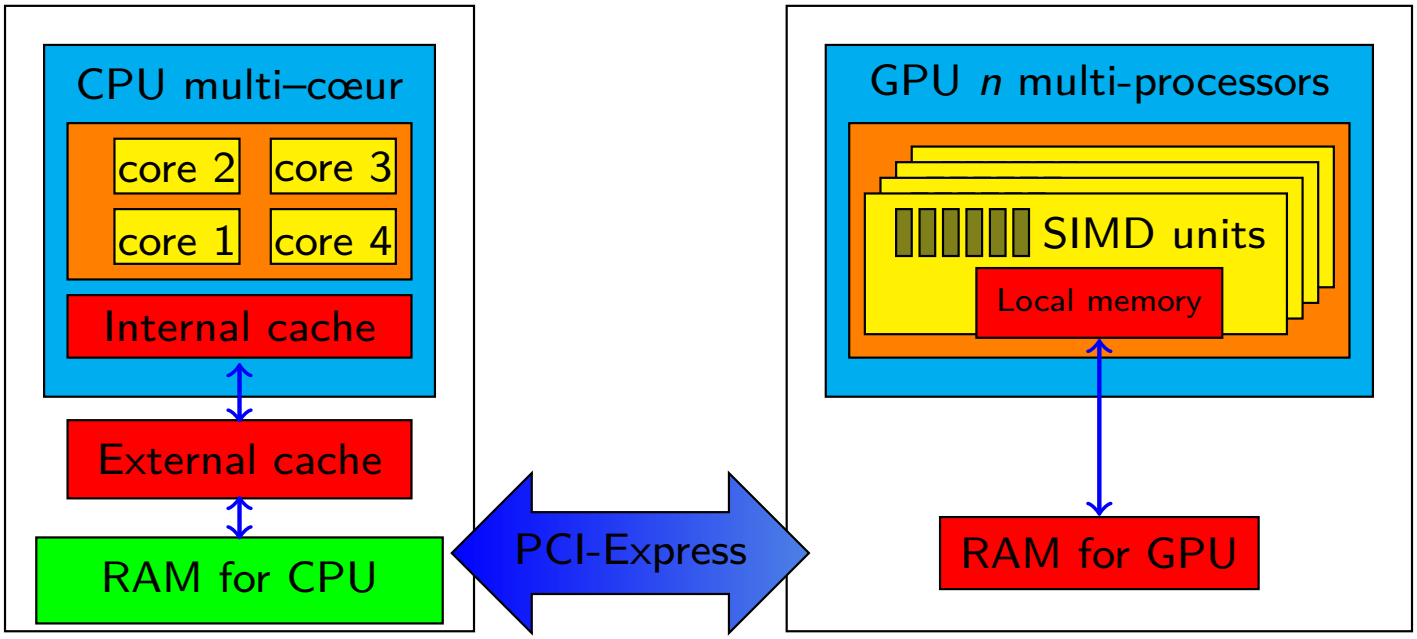
Plan du cours

1. Architecture des GPGPUs
2. Modèle de programmation
   - Outils de compilation
   - Programmation des noyaux
   - Cuda : API C
   - Occupation
3. OpenCL
4. OpenACC
5. OpenMP

Relation CPU–GPGPU

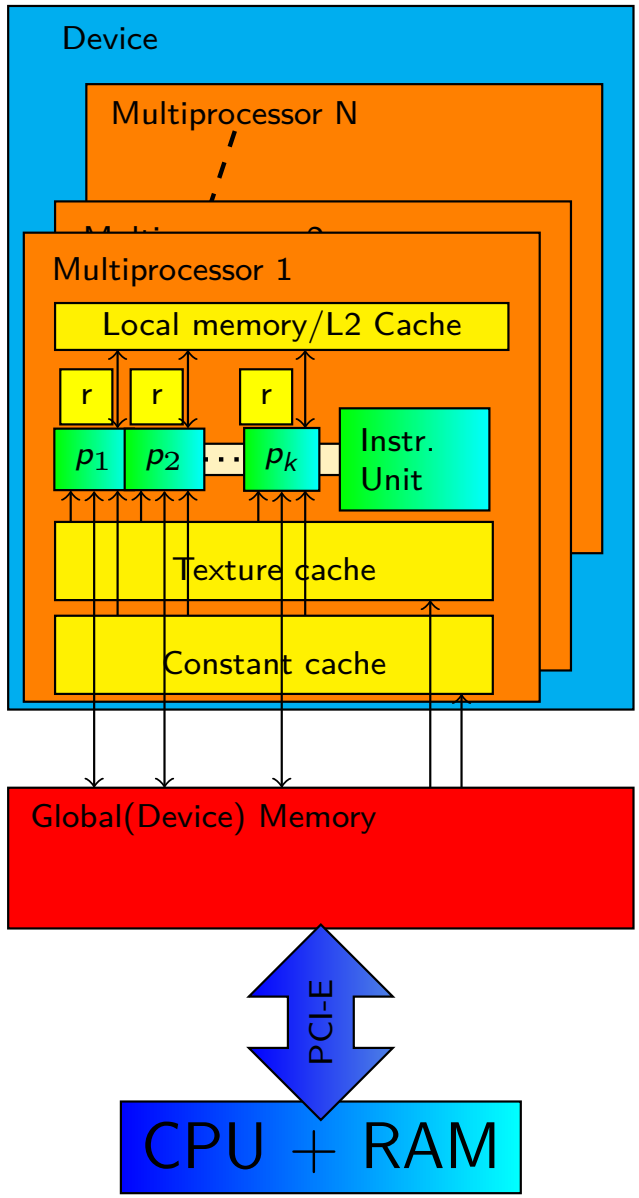
Définition

- Le GPGPU est contrôlé par le CPU comme calculateur hybride MIMD–SIMD pour exécuter des algorithmes adaptés à son architecture;
- CPU et GPGPU sont des calculateurs multi-cœurs et ont une mémoire architecturée sous forme hiérarchique.



Détail de l'architecture GPGPU

- GPGPU : Ensemble de N petites unités SIMD indépendantes partageant une mémoire global commune : N multiprocesseurs;
- Multiprocesseur : Petite unité SIMD avec :
  - k ALU synchronisés;
  - 1 décodeur d'instruction;
  - Trois mémoires partagées pour tous les ALUs (dont deux mémoires caches)
  - R registres distribués parmi les ALUs (locales à chaque *thread*) (Exemple Maxwell : 65536)



NVIDIA : système de numérotation hardware

Numéros de version NVIDIA/Cuda

Deux systèmes de numérotation de version :

1. **Numérotation du hardware** : Un numéro majeur donnant l'architecture mise en œuvre sur le GPGPU utilisé, un numéro mineur donnant les améliorations qui ont pu y être apportées (Exemple : parallélisme dynamique qu'à partir du hardware 3.5).
2. **Numérotation du driver** : La version de la bibliothèque Cuda utilisée (12 pour la plus récente).

Comment connaître ses numéros de version

1. Par l'application `deviceQuery` (voir prochains transparents);
2. En utilisant l'API C : `cudaGetDeviceProperties`

```
cudaDeviceProp prop;
cudaGetDeviceProperties(&prop, num_device);
std::cout << "Capabilité : " << prop.major << "."
                             << prop.minor << std::endl;
```

queryDevice

Utilitaire queryDevice

- Fourni avec les "Samples" proposés à l'installation par NVIDIA ou téléchargeables à part;
- Doit être compilé avant utilisation !
- Localisé au niveau des Samples dans `1_Utilities/deviceQuery`

Exemple sortie obtenue (vue partielle)

```
  CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "GeForce GTX 970M"
  CUDA Driver Version / Runtime Version          8.0 / 8.0
  CUDA Capability Major/Minor version number:    5.2
  Total amount of global memory:                 3040 MBytes (3187343360 bytes)
  (10) Multiprocessors, (128) CUDA Cores/MP:     1280 CUDA Cores
  GPU Max Clock rate:                            1038 MHz (1.04 GHz)
  Memory Clock rate:                             2505 Mhz
  Memory Bus Width:                              192-bit
  L2 Cache Size:                                 1572864 bytes
  ...
```

Organisation des cœurs de calcul

Multiprocesseurs

- Un GPGPU contient plusieurs multi-processeurs (10 dans notre exemple);
- Chaque multi-processeur contient une mémoire locale, des registres et un nombre de cœur (128 dans notre exemple);
- Les cœurs de calcul sont organisés par groupe (Warp) de 16 ou 32 threads (selon les architectures).
- Un Warp est constitué de deux demi–warps. Un demi–warp possède une architecture SIMD.

Organisation de la mémoire sur GPGPU

Hiérarchie mémoire

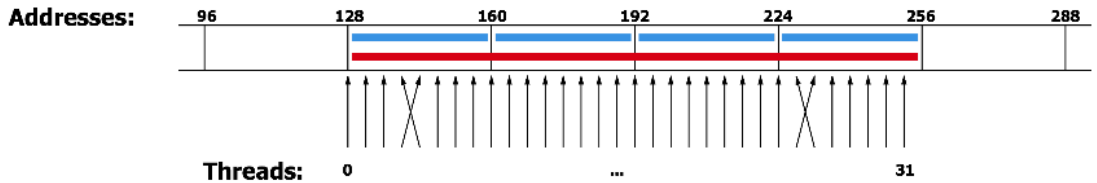
1. Chaque thread possède sa propre mémoire locale (registres), éventuellement partagée avec les threads appartenant au même Warp.
2. Chaque thread partage la même mémoire que les threads appartenant au même “multi–processeur”;
3. Tous les threads partagent la même mémoire globale;

Coalescence

1. La mémoire globale est une mémoire entrelacée à 6 ou 12 voies (dont deux de contrôle) de largeur 32 octets;
2. Les threads d'un même warp accèdent à la mémoire globale par accès de 128 octets : une requête pour des données sur quatre octets, deux requêtes pour des données de huit octets, soit une requête par demi–warp, quatre octets pour des données de seize octets, soit une requête par quart de warp.
3. Pour cela, les données lues et écrites par un warp doivent être contiguës en mémoire et alignées sur 128 octets.

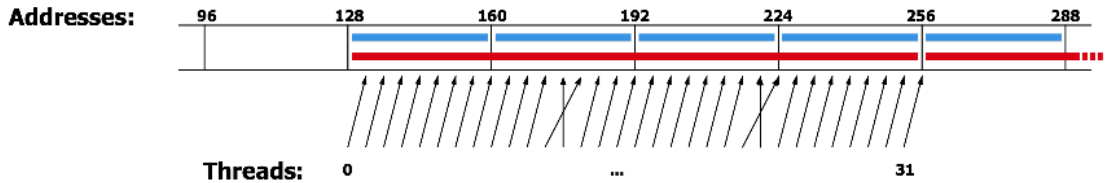
Coalescence

Aligned accesses (sequential/non-sequential)



Compute capability:	2.0 and later	
Memory transactions:	Uncached	Cached
	1x 32B at 128 1x 32B at 160 1x 32B at 192 1x 32B at 224	1x 128B at 128

Mis-aligned accesses (sequential/non-sequential)



Compute capability:	2.0 and later	
Memory transactions:	Uncached	Cached
	1x 32B at 128 1x 32B at 160 1x 32B at 192 1x 32B at 224 1x 32B at 256	1x 128B at 128 1x 128B at 256

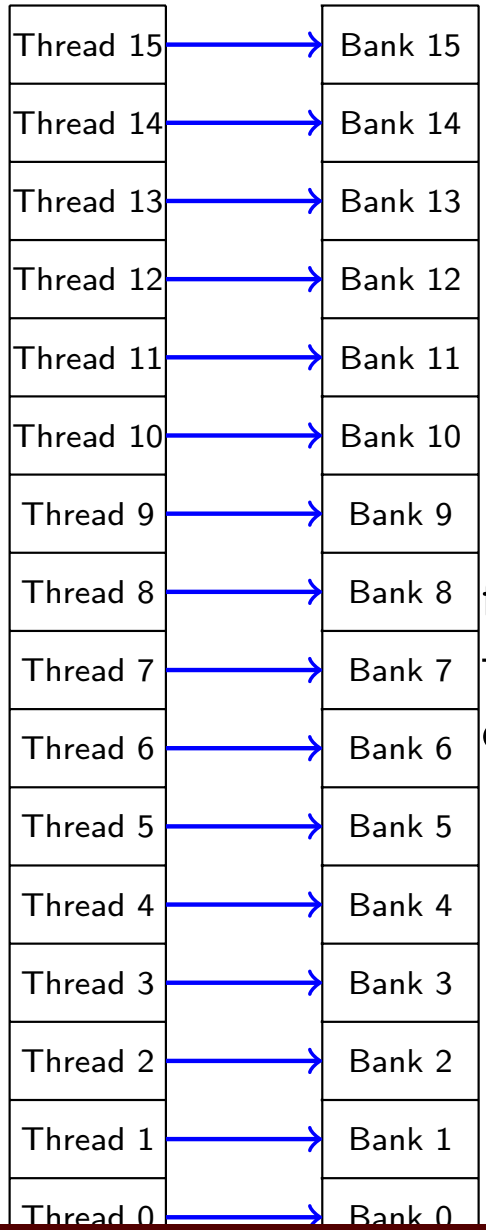
Mémoire partagée

- Des centaines de fois plus rapide que la mémoire globale
  - 16 bancs peuvent être accédés simultanément sur un hardware 1.X
  - 32 bancs peuvent être accédés simultanément sur un hardware 2.0
  - 32 octets consécutifs sont assignés à des bancs successifs
- Des Threads d'un même bloc peuvent coopérer via la mémoire partagée
  - 16 KBytes maximum par multiprocesseur avec un hardware 1.X
  - 48 KBytes maximum par multiprocesseur avec un hardware 2.0
  - Mais sur le hardware 2.0, la mémoire cache L1 est la même mémoire que la mémoire partagée : le programmeur doit contrôler la taille de mémoire utilisée par le cache L1 et la mémoire partagée.
- Permet d'éviter des accès non coalescent en mémoire globale

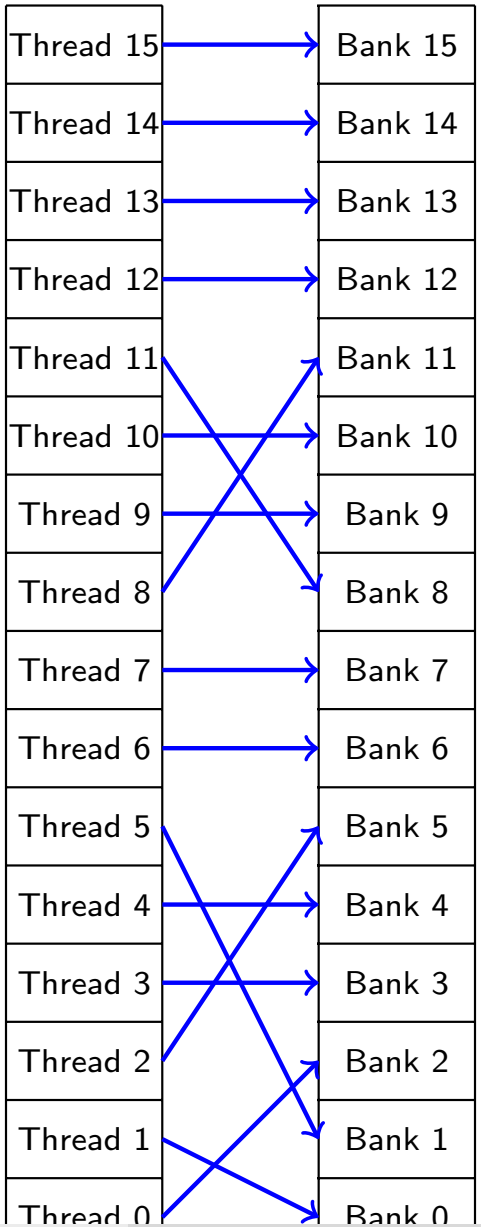
Mémoire partagée : problèmes de performance

- Les cas idéaux :
  - Si tous les threads d'un demi-warp (ou un warp pour le hardware 2.0) accèdent à des bancs différents, pas de conflit de bancs
  - Si tous les threads d'un demi-warp (un warp en 2.0) lisent une adresse identique, pas de conflit de bancs (broadcast)
- Les pires cas :
  - Conflit de banc : Plusieurs threads d'un même (1/2)-warp accèdent à un même banc
  - L'accès est sérialisé
  - Coût = max # d'accès simultanés à un même banc

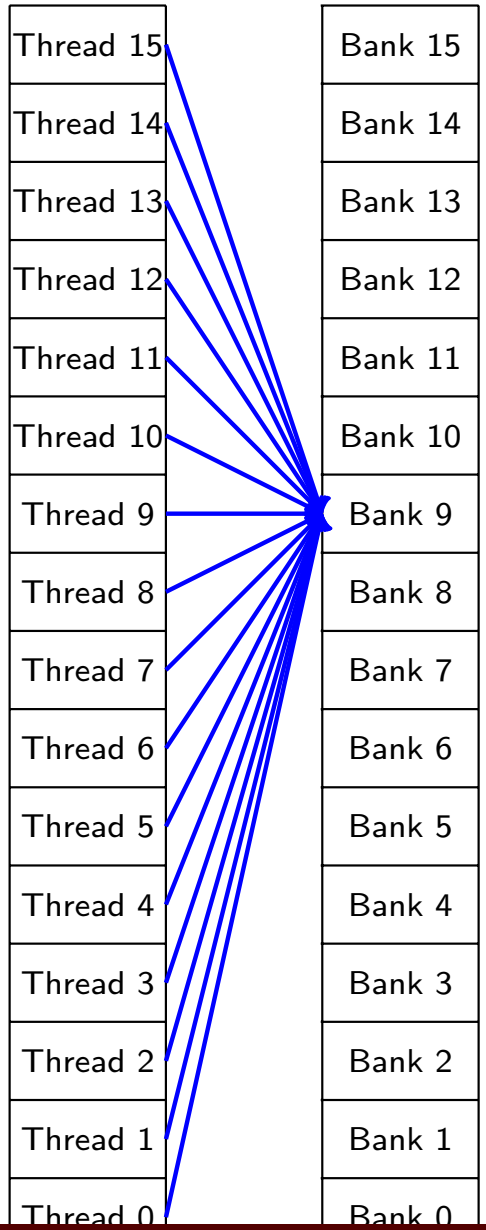
Accès à la mémoire partagée



Motif d'accès sans conflits de bancs : chaque thread du demi-warp accède à un banc différent.

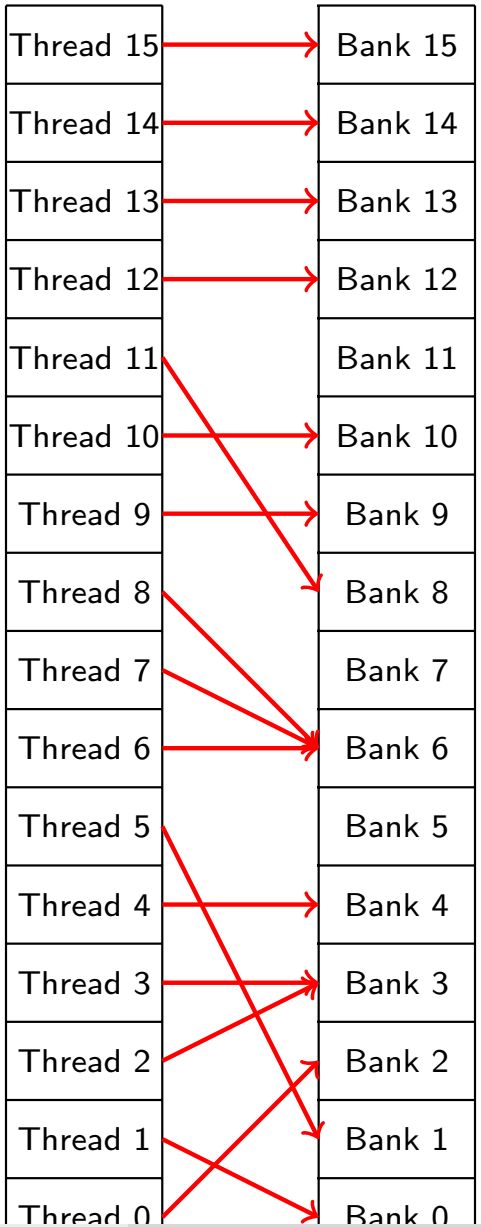


Accès mémoire partagée



← Chaque thread lit une adresse d'un même banc : pas de conflit (broadcasting)

Plusieurs Threads accèdent au même banc : conflit →



Principe de compilation CUDA et C++

Plusieurs cas de figure :

- Compilation d'un code entièrement développé en CUDA;
- Compilation d'un code CUDA avec récupération de code C/C++;
- Compilation code CUDA avec compilateur spécifique pour la partie C/C++ sur CPU.

Compilation d'un code entièrement développé en CUDA

Contenu et production du code

- Définitions variables et fonctions avec "qualificateurs" CUDA.
- Du code C ou C++ avec fonctionnalités CUDA;
- Code C ou C++ "standard".
- Les extensions : ".h" pour les headers, ".cu" pour les sources.
- On compile à l'aide du compilateur NVidia : `nvcc`
- On obtient un code CPU contenant du code GPU intégré.

Pour les codes C/C++ simples

- Possibilité de tout compiler avec nvcc dans des fichiers .cu
- Mais les optimisations pour le CPU peuvent en souffrir.

Compilation d'un code avec récupération sources C/C++

Contenu et production du code

- On compile les fichiers C/C++ (.c, .cc, .h) avec nvcc;
- Les fichiers contenant du code Cuda (.cu, .h) avec nvcc;
- On fait une édition des liens du tout pour obtenir un code binaire contenant les binaires pour le CPU et le GPU.

Problèmes

- A l'édition des liens, des problèmes peuvent apparaître avec des templates...
- Problèmes d'optimisations pour le code CPU pouvant apparaître.

Compilation d'applications CUDA avec compilateur spécifique

Contenu et production du code

- Codes C/C++ (.c, .cc, .h) : On le compile avec son compilateur préféré (gcc, g++, icc, ...);
- Code Cuda : On le compile avec nvcc;
- On fait l'édition de lien des objets obtenus

Problèmes

- Des problèmes de nommage peuvent apparaître (mais pas avec **gcc**).

Principe d'exécution

Exécution d'une application CUDA

- On lance une application CPU d'apparence classique;
- On réalise du "Remote Process Control" (RPC) sur le GPU depuis le CPU (exécution de "kernels");
- Pour être efficace, il faut minimiser les transferts des données;
- On peut exécuter les "kernels" en mode bloquant (synchrone) ou non bloquant (asynchrone) pour le programme CPU : $\rightarrow$ possibilité d'utiliser simultanément le CPU et le GPU.

C étendu

- **Nouv. déclarations** : global, device, shared, local, constant

```
__device__ float filter[N];
__global__ void convolve(float* image) {
__shared__ float region[M];
```

- **nouveaux mots clefs** : threadIdx, blockIdx

```
region[threadIdx] = image[i];
```

- **Intrinsics** : __syncthreads

```
__syncthreads(); image[j] = result;
```

- **API d'exécution** : Memory, symbol, execution management

```
void* myImg = cudaMalloc(bytes);// Alloue memoire sur GPU
```

- **Exécution de fonction**

```
convolve<<<100,10>>>(myImg);// 100 blocs de 10 threads
```

“Qualifieurs” de CUDA

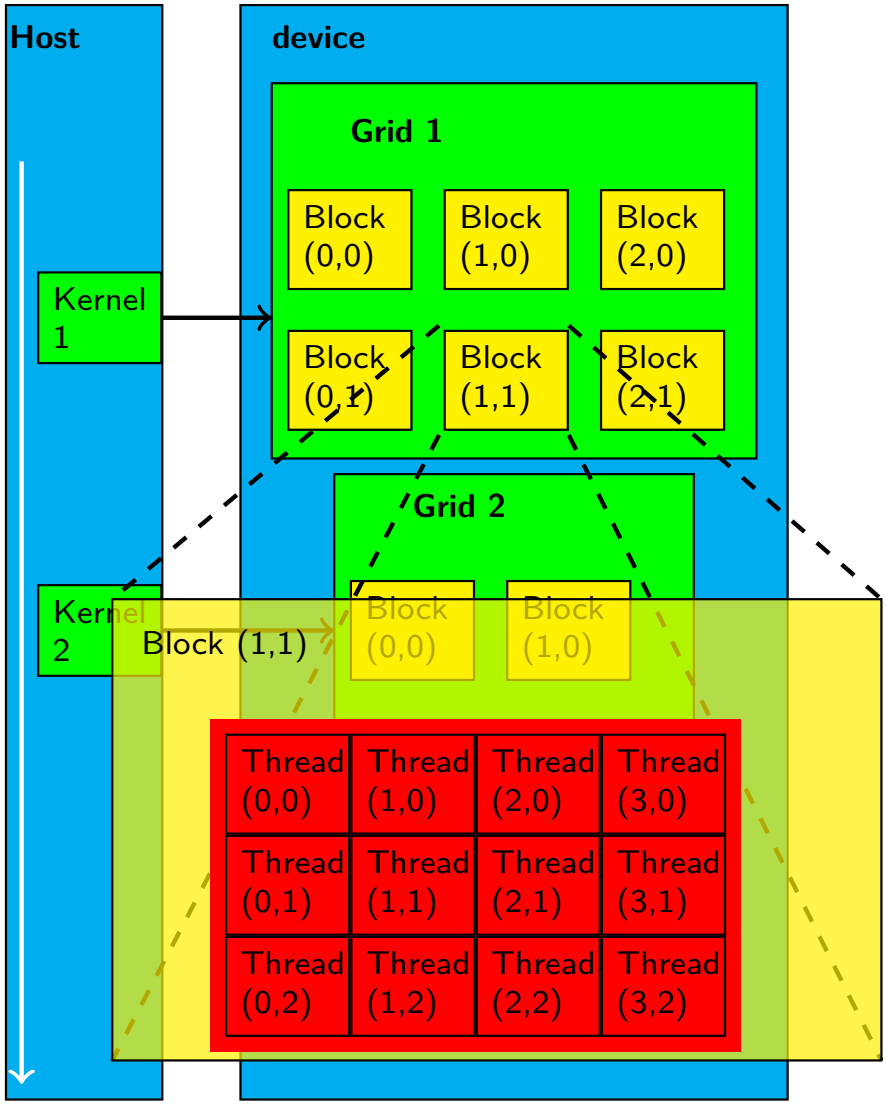
Propriétés des “qualifieurs” de CUDA:

	`__device__`	`__host__`	`__global__`
Fonctions	Appel sur GPU Exécution sur GPU	Appel sur CPU Exécution sur CPU	Appel sur CPU Exécution sur GPU
	`__device__`	`__constant__`	`__shared__`
Variables	Mémoire globale GPU	Mémoire constante GPU	Mémoire partagé multi-processeurs
	Temps de vie de l’application	Temps de vie de l’application	Temps de vie du bloc de thread
	Lisible/enregistrable sur CPU et GPU	Enregistrable CPU, lisible GPU	Lisible sur GPU : utilisé comme *cache* mémoire géré à la main pour la mémoire global GPU

→ Les qualifieurs séparent les codes CPU et GPU.

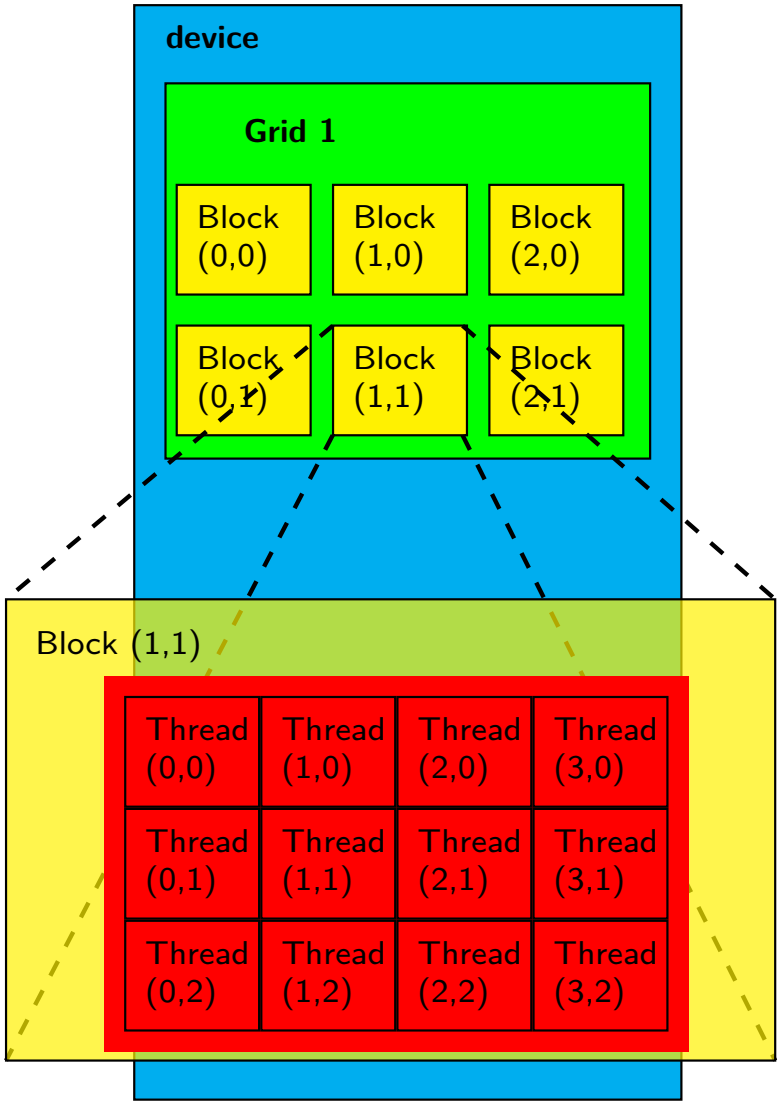
Distribution des threads : grilles et blocs

- Un noyau est exécuté comme une grille de blocs de thread
  - Tous les threads partagent le même espace de mémoire de donné
- Un bloc de threads est un ensemble de threads qui peuvent coopérer les uns les autres en :
  - synchronisant leur exécution
  - partageant leurs données à travers une mémoire partagée rapide
- Deux threads provenant de deux blocs différent ne peuvent pas coopérer :
  - Opérations atomiques



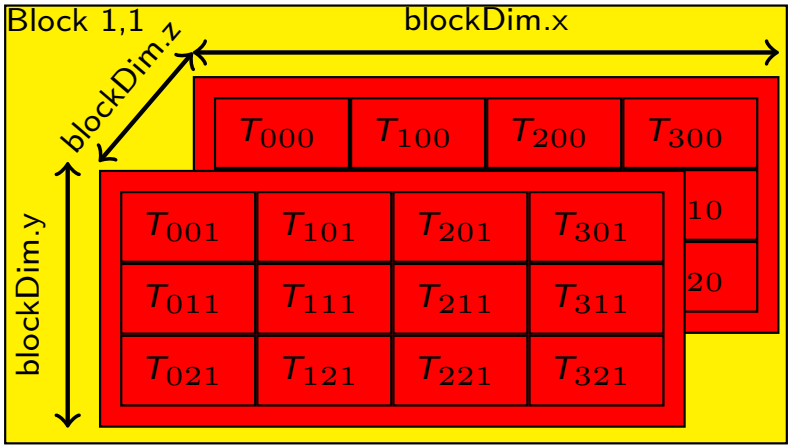
Identification des blocs et des threads

- Chaque thread et bloc ont des Ids :
  - Chaque thread peut décider sur quelles données travailler
  - Block ID : 1D, 2D ou 3D depuis Cuda 3.0
  - Thread ID: 1D, 2D ou 3D.
- Simplifie l'adressage mémoire quand on gère des données multidimensionnelles :
  - Image processing
  - Résolution d'EDP sur des volumes ou surfaces
  - ...



Mots clefs pour les blocs et les threads

- Mots clefs pour les blocs :
  - threadId.[x,y,z] définit la position du thread dans le bloc;
  - blockDim.[x,y,z] définit les dimensions du bloc.
- Mots clefs pour les grilles :
  - blockId.[x,y,z] définit la position du bloc dans la grille
  - gridDim.[x,y,z] définit les dimensions de la grille



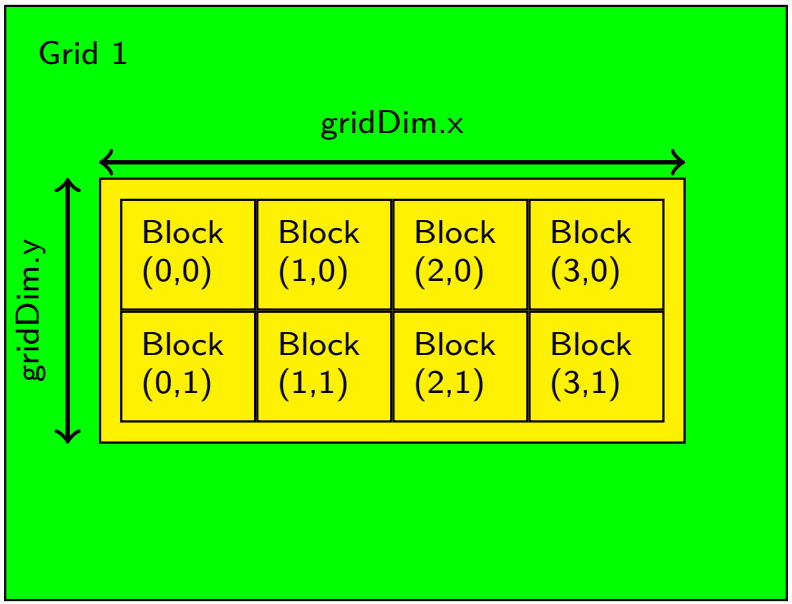


Tableau de threads

Un noyau CUDA est exécuté par un tableau de threads

- Tous les threads exécutent le même code
- Chaque thread a un ID utilisé pour calculer les adresses mémoires et faire des contrôles pour le branchement (if, etc...)

Thread ID

L'ID d'un thread dans un bloc est :

```
t_id = threadIdx.x +
       threadIdx.y*(blockDim.x) +
       threadIdx.z*(blockDim.x*blockDim.y);
```

- `threadIdx.[x,y,z]` : Indice du thread dans la dimension x,y,z
- `blockDim.[x,y,z]` : Taille du bloc dans la dimension x,y,z

Thread ID(2)

- Considérons un bloc de dimension

```
blockDim.x = 8
blockDim.y = 6
blockDim.z = 4
```

- Et un thread d'indices

```
threadIdx.x = 1
threadIdx.y = 2
threadIdx.z = 3
```

- Le thread est alors d'indice global dans le bloc :

```
1+(2*8)+3*(6*8) = 161
```

Exemple 1

Addition de deux vecteurs

```
__global__ void addVector( int dim, const float* u,
                           const float* v, float* w )
{
    int ind = threadIdx.x + blockIdx.x * blockDim.x;

    if (ind < dim)
        w[ind] = u[ind]+v[ind];
}
```

Exemple 2

Addition de deux matrices

```
__global__ void addMatrix(float* A, float* B, float* C, int N)
{
    unsigned int iGlob = threadIdx.x + blockIdx.x * blockDim.x;
    unsigned int jGlob = threadIdx.y + blockIdx.y * blockDim.y;
    unsigned int ind   = iGlob + jGlob * N;
   if ((iGlob<N)&&(jGlob<N)) C[ind] = A[ind] + B[ind];
}
```

Exemple 3

Multiplication matrice–matrice :

```
#define BLOCK_SIZE 16
__global__ void
matrixMul( float* C, const float* A, const float* B, int dim )
{ // On fait une approche par bloc : 1 bloc pour un groupe de thread
  // Indice premier bloc lu par le thread
  int aBegin = dim * BLOCK_SIZE * blockIdx.y;
  int aEnd   = aBegin + dim - 1; // Et indice suivant dernier bloc
  int aStep  = BLOCK_SIZE; // Et pas pour prochain bloc

  int bBegin = BLOCK_SIZE * blockIdx.x;// indice 1er bloc
  int bStep  = BLOCK_SIZE * dim; // Pas pour prochain bloc

  // Chaque thread calcul un coefficient de C :
  int ic = dim * BLOCK_SIZE * blockIdx.y + BLOCK_SIZE * blockIdx.x;
  float Csub = C[ic + dim*threadIdx.y + threadIdx.x];
```

Exemple 3 (suite)

Multiplication matrice–matrice (suite):

```
  ...
  // Boucle sur les blocs :
  for ( int a = aBegin, b = bBegin; a <= aEnd; a += aStep, b += bStep )
  {
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
    // Chaque thread du group charge un elt des blocs courants
    // de A et de B en shared memory :
    As[threadIdx.y][threadIdx.x] = A[a + dim*threadIdx.y + threadIdx.x];
    Bs[threadIdx.y][threadIdx.x] = B[b + dim*threadIdx.y + threadIdx.x];
    // On s'assure que tous les threads ont bien remplis As et Bs :
    __syncthreads();
    // Puis multiplication des deux blocs qu'on rajoute à Csub :
    for ( int k = 0; k < BLOCK_SIZE; ++k )
      Csub += As[threadIdx.y][k] * Bs[k][threadIdx.x];
    __syncthreads();// On s'assure d'avoir fini le calcul bloc
  }
  C[ic + dim*threadIdx.y + threadIdx.x] = Csub;
}
```

Caractéristiques de CUDA : facile et léger

- L'API est une extension du langage C $\rightarrow$ apprentissage aisé;
- Le hardware est conçu pour une exécution et une gestion des tâches légère $\rightarrow$ performance élevée.

Allocation mémoire

- `cudaMalloc()`
  - Alloue des objets sur la **mémoire globale** du GPU
  - Deux paramètres nécessaires :
    1. Adresse du pointeur sur l'objet alloué;
    2. Taille de l'objet alloué;
- `cudaFree()`
  - Libère des objets de la mémoire globale du GPU;
    1. Pointeur sur l'objet à libérer;

Ex.: Alloue une matrice 1024*1024 en simple précision

```
#define MATRIX_SIZE 1024*1024
float* MyMatrixOnDevice;
int size = MATRIX_SIZE*sizeof(float);
cudaMalloc((void**)&MyMatrixOnDevice, size);
cudaFree(MyMatrixOnDevice);
```

Transfert de données en CUDA entre le CPU et le GPU

`cudaMemcpy()`

- Transfert de données
- Quatre paramètres nécessaires :
  - Pointeur vers la source
  - Pointeur vers la destination
  - Nombre d'octets à copier
  - Type de transfert :
    - CPU vers CPU
    - CPU vers GPU
    - GPU vers CPU
    - GPU vers GPU

Des variantes asynchrones supportées depuis la version hardware 1.1HW

Exemples de transfert CUDA entre le CPU et le GPU

- Exemple de code :
  - Transfert une matrice 1024*1024 en simple précision
  - `MyMatrixOnHost` est un pointeur sur la mémoire du CPU et `MyMatrixOnDevice` est un pointeur sur la mémoire globale du GPU
  - `cudaMemcpyHostToDevice` et `cudaMemcpyDeviceToHost` sont des constantes symboliques

```
cudaMemcpy(MyMatrixOnDevice, MyMatrixOnHost, size,
           cudaMemcpyHostToDevice);
cudaMemcpy(MyMatrixOnHost, MyMatrixOnDevice, size,
           cudaMemcpyDeviceToHost);
```

Déclaration de fonctions CUDA

	Exécuté sur	Appelable seulement de
`__device__ float DeviceFunc()`	GPU	GPU
`__global__ void KernelFunc()`	GPU	CPU
`__host__ float HostFunc()`	host	host

- `__global__` définit une fonction noyau : doit retourner toujours `void`.
- `__device__` fonctions sur GPU dont on ne peut récupérer l'adresse (semblable à des fonctions inline);
- Pour les fonctions exécutées sur le GPU :
  - Pas de fonctions récursives
  - Pas de déclaration de variables statiques dans la fonction
  - Pas de nombre d'arguments variables

Appeler un noyau : création de threads

- Une fonction noyau doit être appelée avec une configuration d'exécution :

```
__global__ void KernelFunc(...);
dim3 DimGrid(100,50);   // 5000 Thread blocks
dim3 DimBlock(8,8,4);   // 256 threads per block

KernelFunc<<<DimGrid, DimBlock>>>(...);
```

- Tout appel à un noyau est asynchrone, une synchronisation explicite nécessaire pour des rendez-vous.

Optimiser le nombre de threads par bloc

- Choisir les nombre de threads par bloc comme un multiple de la taille d'un warp
  - Essayer d'éviter le gâchis de warp en sous effectifs
- Plusieurs threads par bloc = meilleur recouvrement de la latence mémoire
  - L'invocation de noyau peut se planter si trop de registres utilisés.
- Heuristiques
  - Minimum requis par le hardware : 64 Threads par bloc
    - Seulement si beaucoup de blocs concurrents
  - 192 ou 256 threads est un meilleur choix :
    - Généralement assez de registre pour arriver à compiler et exécuter
  - Tout cela dépend de votre calcul, alors expérimentez !

Heuristique taille Grille/Bloc

- # de blocs > # de multiprocesseurs
  - Pour que tous les multiprocesseurs aient au moins un bloc à exécuter
- # de blocs / # de multiprocesseurs > 2
  - Plusieurs blocs peuvent être en concurrence dans un multiprocesseur
  - Les blocs qui n'attendent pas un `__syncthreads()` sont toujours actifs
  - Selon les ressources valables – registre, mémoire partagée
- # de blocs > 100 pour s'adapter aux futurs hardware
  - Blocs sont exécutés en pipeline sur un multiprocesseur
  - 1000 blocs par grille devrait s'adapter aux générations futures de GPU

Occupation

- Les instructions dans les threads sont exécutées simultanément, alors exécuter d'autres warps est le seul moyen de cacher les latences et de garder le hardware occupé.
- Occupation = nombre de warps s'exécutant en concurrence sur un multiprocesseur divisé par le nombre maximal de warps qui peut être exécuté en concurrence.
- Limité par l'utilisation des ressources :
  - Registres
  - Mémoire partagée
  - threads/blocs

Cas d'occupation

- Hardware 1.0/1.1

768 threads :	$3 \times 256 (16 \times 16)$	8×64 (66% utilisé)
16 kBytes partagé	3×5kbytes	8×1.9kbytes
8192 registers	10 per thread	15 per thread
8 blocks	3 blocks	8 blocks

- Hardware 1.2/1.3

1024 threads :	$4 \times 256 (16 \times 16)$	8×64 (50% utilisé)
16 kBytes partagé	4×3.9kbytes	8×1.9kbytes
16384 registers	15 per thread	30 per thread
8 blocks	4 blocks	8 blocks

- Hardware 2.0

1024 threads :	$4 \times 256 (16 \times 16)$	8×64 (50% utilisé)
32 kBytes partagé	4×7.8kbytes	8×3.8kbytes
32768 registers	30 per thread	60 per thread
8 blocks	4 blocks	8 blocks

Retour exemple 1

Addition deux vecteurs : fonction appel noyau

```
void add_vector(const float* u, const float* v, float* w, int N)
{
  int grdSize, blockSize = 256;
  float *u_dev, *b_dev, *c_dev;
  // Alloue et copie les vecteurs u, v et alloue w sur le GPU
  cudaMalloc(((void**)&u_dev, sizeof(float)*N);
  cudaMemcpy(u_dev, u, sizeof(float)*N, cudaMemcpyHostToDevice);
  cudaMalloc(((void**)&v_dev, sizeof(float)*N);
  cudaMemcpy(v_dev, v, sizeof(float)*N, cudaMemcpyHostToDevice);
  cudaMalloc(((void**)&w_dev, sizeof(float)*N);
  // Calcule la configuration d'execution du noyau
  dim3 dimBlock(blockSize);
  grdSize = (N%blockSize>0 ? N/blockSize+1: N/blockSize);
  dim3 dimGrid(grdSize);
  // Appel du noyau :
  addVector<<<dimGrid,dimBlock>>>(N,u_dev,v_dev,w_dev);
  // Copie le resultat sur le CPU et libere la memoire GPU
  cudaMemcpy(w, w_dev, sizeof(float)*N, cudaMemcpyDeviceToHost);
  cudaFree(u_dev); cudaFree(v_dev); cudaFree(w_dev);
}
```

OpenCL en quelques mots

Pourquoi OpenCL

- CUDA : bibliothèque conviviale, puissante et rapide mais uniquement portable sur des cartes NVIDIA !;
- Besoin d'avoir une bibliothèque plus universelle permettant de gérer des accélérateurs de calcul, d'autres cartes graphiques, utilisable sur smartphone et tablettes, etc..
- Permettre une accélération de calcul pour les pages web : WebCL.

OpenCL en quelques mots

- Standard mis au point par le Khronos Group (qui font aussi la standardisation d'OpenGL);
- Permet la programmation des GPGPUs, mais aussi des CPUs (Intel mais aussi les CELLs d'IBM);
- Compilateur intégré à la bibliothèque (comme pour les shaders avec OpenGL);

OpenCL : Pour et Contre

Pros

- Portable sur un grand nombre de plateformes;
- Programmation des noyaux proche de CUDA;
- Standard ouvert non propriétaire;
- Support de plusieurs versions d'OpenCL prévu !

Cons

- L'API pour la compilation et l'exécution des noyaux est complexe et lourde;
- Moins performante que CUDA sur les NVIDIAs;
- Intel pour ces processeurs many-cœurs a plutôt choisi les options multithreading (TBB en particuliers pour les Knights Landing);

Programmation du noyau

Noyau OpenCL

```
global float filter[N];
kernel void
convolve(float* image) {
    local float region[M];
    ...
    int ind =
          get_global_id(0);
    region[ind] = image[i];
    barrier(CLK_LOCAL_MEM_FENCE);
    ...
    image[j] = result;
}
```

Noyau CUDA

```
__device__ float filter[N];
__global__ void
    convolve(float* image) {
    __shared__ float region[M];
    ...
    int ind = threadId.x+
          blockId.x*blockDim.x;
    region[ind] = image[i];
    __syncthreads();
    ...
    image[j] = result;
}
```

API d'OpenCL : Plateforme

Plateforme

- Plateforme OpenCL $\equiv$ mise en œuvre du standard OpenCL;
- Plusieurs plateformes possibles sur une machine donnée;
- clGetPlatformIDs(cl_uint nb_entries,cl_platform_id *platforms,cl_uint *nb_platforms) :

```
cl_uint nbEntries;
clGetPlatformIDs(0, nullptr, &nbEntries);
std::vector<cl_platform_id> platforms(nbEntries);
clGetPlatformIDs(platforms.size(), platforms.data(), nullptr);
```

- On peut ensuite interroger chaque plateforme pour connaître les device supportés et leur type (CPU ou GPGPU) `clGetDeviceIDs(cl_platform_id platform, cl_device_type device_type, cl_uint nb_entries, cl_device_id *dev, cl_uint* nb_dev )` :

```
cl_uint nbDev;
clGetDeviceIDs(platforms[0], CL_DEVICE_TYPE_ALL, 0,
               nullptr, &nbDev);
std::vector<cl_device> devs(nbDev);
clGetDeviceIDs(platforms[0], CL_DEVICE_TYPE_ALL, nbDev,
               devs.data(), nullptr);
```

API d'OpenCL : contexte

- Pour chaque device utilisé, il faut créer un contexte;
- Un contexte en OpenCL permet de gérer les queues de commande, la mémoire
- le programme et les noyaux OpenCL;
- `cl_context clCreateContext( cl_context_properties *properties, cl_uint num_devices, const cl_device_id *devices, void *pfn_notify ( const char *errinfo, const void *private_info, size_t cb, void *user_data), void *user_data, cl_int *errcode_ret )` : Créé un contexte !

```
cl_int ret;
context = clCreateContext(nullptr, 1, &devs[0],
                          nullptr, nullptr, &ret);
```

API d'OpenCL : Queue de commande

- Permet de configurer une queue de commande qui : exécute les noyaux dans l'ordre d'appel ou dans un ordre dicté uniquement par la dépendance des données;
- `cl_command_queue clCreateCommandQueue( cl_context context, cl_device_id device, cl_command_queue_properties properties, cl_int *errcode_ret)` :

```
cl_command_queue command_queue;
command_queue = clCreateCommandQueue(context, device_id, 0, &ret);
```

API d'OPENCL : Allocation mémoire

- Se fait au travers des objets de type `cl_mem`
- Permet de réserver et de copier ou de réserver seulement.
- `cl_mem clCreateBuffer ( cl_context context, cl_mem_flags flags, size_t size, void *host_ptr, cl_int *errcode_ret)`

```
cl_mem u_dev, v_dev, w_dev;
u_dev = clCreateBuffer(context, CL_MEM_READ_ONLY,
                        dim * sizeof(float), u, &ret);
v_dev = clCreateBuffer(context, CL_MEM_READ_ONLY,
                        dim * sizeof(float), v, &ret);
w_dev = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
                        dim * sizeof(float), nullptr, &ret);
```

Création d'un noyau de calcul

Code source : `vecadd.cl`

```
kernel void vect_add_cl(global const float* u,
                        global const float* v,
                        global       float* w,
                        const int    dim )
{
    const ind = get_global_id(0);
    if ( ind < dim )
      w[ind] = u[ind] + v[ind];
    }
```

Création d'un noyau de calcul (suite)

Création d'un programme composé de noyaux :

```
FILE *fp;
char fileName[] = "./vecadd.cl";
char *source_str;
size_t source_size;

/* Load the source code containing the kernel*/
fp = fopen(fileName, "r");
source_str = (char*)malloc(MAX_SOURCE_SIZE);
source_size = fread(source_str, 1, MAX_SOURCE_SIZE, fp);
fclose(fp);

cl_program program =
   clCreateProgramWithSource(context, 1,
                             (const char **)&source_str,
                             (const size_t *)&source_size, &ret);

ret = clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);

kernel = clCreateKernel(program, "vecadd", &ret);
```

Exécution du noyau et lecture du résultat

Passage des arguments

```
ret = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&u_dev);
ret = clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)&v_dev);
ret = clSetKernelArg(kernel, 2, sizeof(cl_mem), (void *)&w_dev);
ret = clSetKernelArg(kernel, 3, sizeof(int), (void *)&dim);
```

Exécution du noyau

```
ret = clEnqueueTask(command_queue, kernel, 0, NULL,NULL);
```

Recopie du résultat en mémoire vive

```
ret = clEnqueueReadBuffer(command_queue, w_dev, CL_TRUE, 0,
                          dim * sizeof(float),w, 0, NULL, NULL);
```

Finalisation et libération des ressources

Finalisation

```
clFlush(command_queue);
clFinish(command_queue);
```

Libération

```
clReleaseKernel(kernel);
clReleaseProgram(program);
clReleaseMemObject(u_dev);
clReleaseMemObject(v_dev);
clReleaseMemObject(w_dev);
clReleaseCommandQueue(command_queue);
clReleaseContext(context);
```

Pourquoi OpenACC ?

Naissance d'OpenACC

- En 2012, le comité de standardisation d'OpenMP veut étendre le langage OpenMP pour gérer les GPGPUs;
- Difficultés de trouver un consensus parmi tous les intervenants du comité;
- Cray, CAPS, NVidia et PGI décident en attendant que le consensus soit trouvé de créer un autre standard de programmation OpenACC pour gérer les GPGPUs "à la OpenMP".

Pour

- Non intrusif : permet de rapidement porter du code sur GPGPU;
- Permet d'utiliser des plateformes Nvidia mais aussi ATI;
- Simplicité d'utilisation d'OpenACC : permet d'obtenir une bonne accélération à moindre coût;

Contre

- Ne permet pas des performances optimales comme Cuda;
- Peu de compilateur le supportent : les compilateurs PGI (gratuits pour usage non commercial) et gnu c/c++ à partir de la version 6.1 (encore au stage d'ébauche !)

Exemple de code

```
#include <stdlib.h>
#include <stdio.h>
void saxpy(long n, float a, float *x, float * y) {
#pragma acc parallel loop
  for (long i = 0; i < n; ++i)
    y[i] = a * x[i] + y[i];
}
int main(int argc, char **argv) {
  float sum;
  long N = 1000000000; // 1 billion floats
  if (argc > 1) N = atoi(argv[1]);
  float *x = (float*)malloc(N * sizeof(float));
  float *y = (float*)malloc(N * sizeof(float));
  for (long i = 0; i < N; ++i) {
    x[i] = 2.0f; y[i] = 1.0f;
  }
  saxpy(N, 3.0f, x, y);
  sum = 0.0f;
  for ( long i = 0; i < N; ++i ) sum += y[i];
  free(x); free(y);
  printf("sum = %f\n",sum);
  return 0;
}
```

GPGPU avec OpenMP

Historique

- Support des GPGPUs par OpenMP depuis la version 4.0 de la norme;
- Pour l'instant, encore très limité : les compilateurs Intel ne supportent que les Xeon Phi, Cray ne propose que OpenACC.
- OpenMP 4.0 pour GPU encore au stade rudimentaire pour GCC
- Valable pour Clang et compilateurs PGIs

Pour

- Approche unifiée avec le reste d'OpenMP;
- Même simplicité que OpenACC;
- Évite de mélanger plusieurs directives de compilation !

Contre

- Ne permet pas d'avoir des performances optimales;
- Peu de compilateur supportent OpenMP 4.0 avec GPU aujourd'hui !

Exemple de code OpenMP pour GPGPU

```
#include <malloc.h>
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char* argv[]) {
    if (argc != 2) {
        printf("Usage: %s \n", argv[0]); return 0;
    }
    int n = atoi(argv[1]);
    double* x = (double*)malloc(sizeof(double) * n);
    double* y = (double*)malloc(sizeof(double) * n);
    double idrandmax = 1.0 / RAND_MAX, a = idrandmax * rand();
    for (int i = 0; i < n; i++) {
        x[i] = idrandmax * rand(); y[i] = idrandmax * rand();
    }
#   pragma omp target data map(tofrom: x[0:n],y[0:n])
    {
        #pragma omp target
        #pragma omp for
        for (int i = 0; i < n; i++)
            y[i] += a * x[i];
    }
    double avg = 0.0, min = y[0], max = y[0];
    for (int i = 0; i < n; i++) {
        avg += y[i];
        if (y[i] > max) max = y[i]; if (y[i] < min) min = y[i];
    }
    printf("min = %f , max = %f , avg = %f\n", min, max, avg / n);
    free(x); free(y);
    return 0;
}
```