Introduction à la programmation des GPUs

Anne-Sophie Mouronval*

Mesocentre de calcul de l'Ecole Centrale Paris
* Laboratoire MSSMat

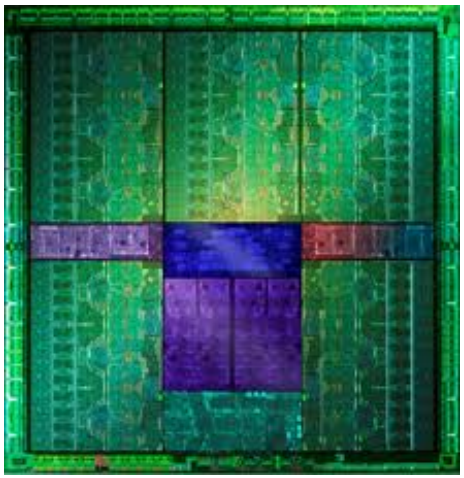
Avril 2013

Plan

Plan

GPU : *Graphics Processing Unit*

- Processeur **massivement parallèle, disposant de sa propre mémoire** assurant les fonctions de calcul de l'affichage.

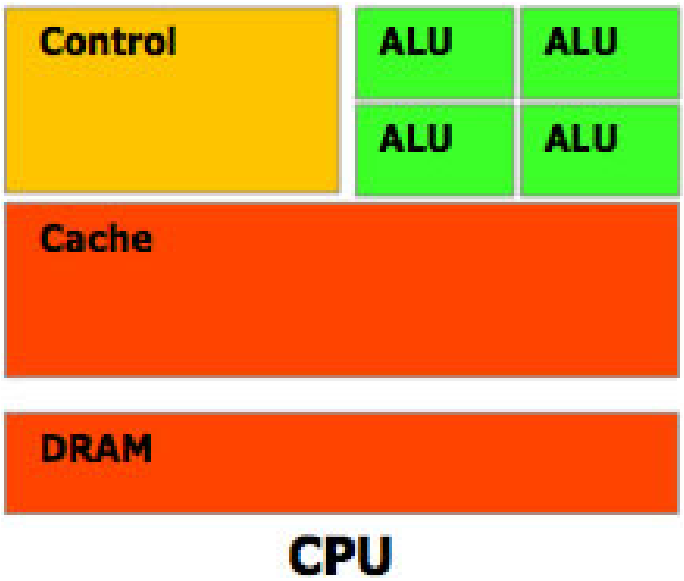


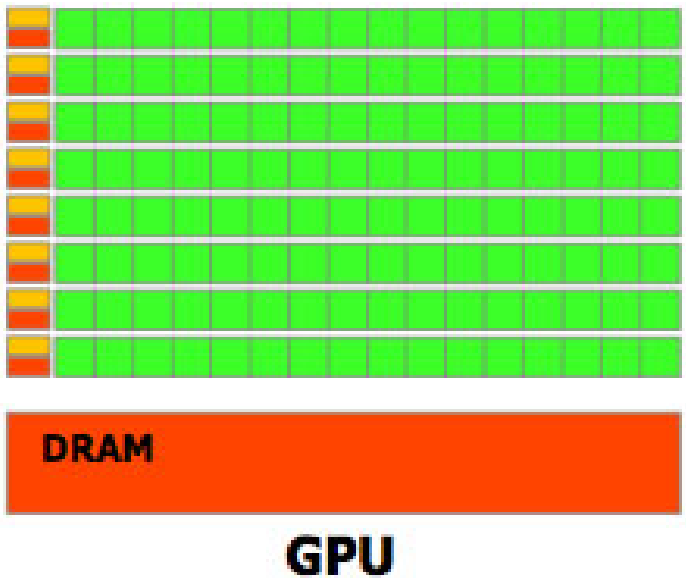
Puce NVIDIA Kepler 20

Carte Tesla K20

CPU *vs* GPU

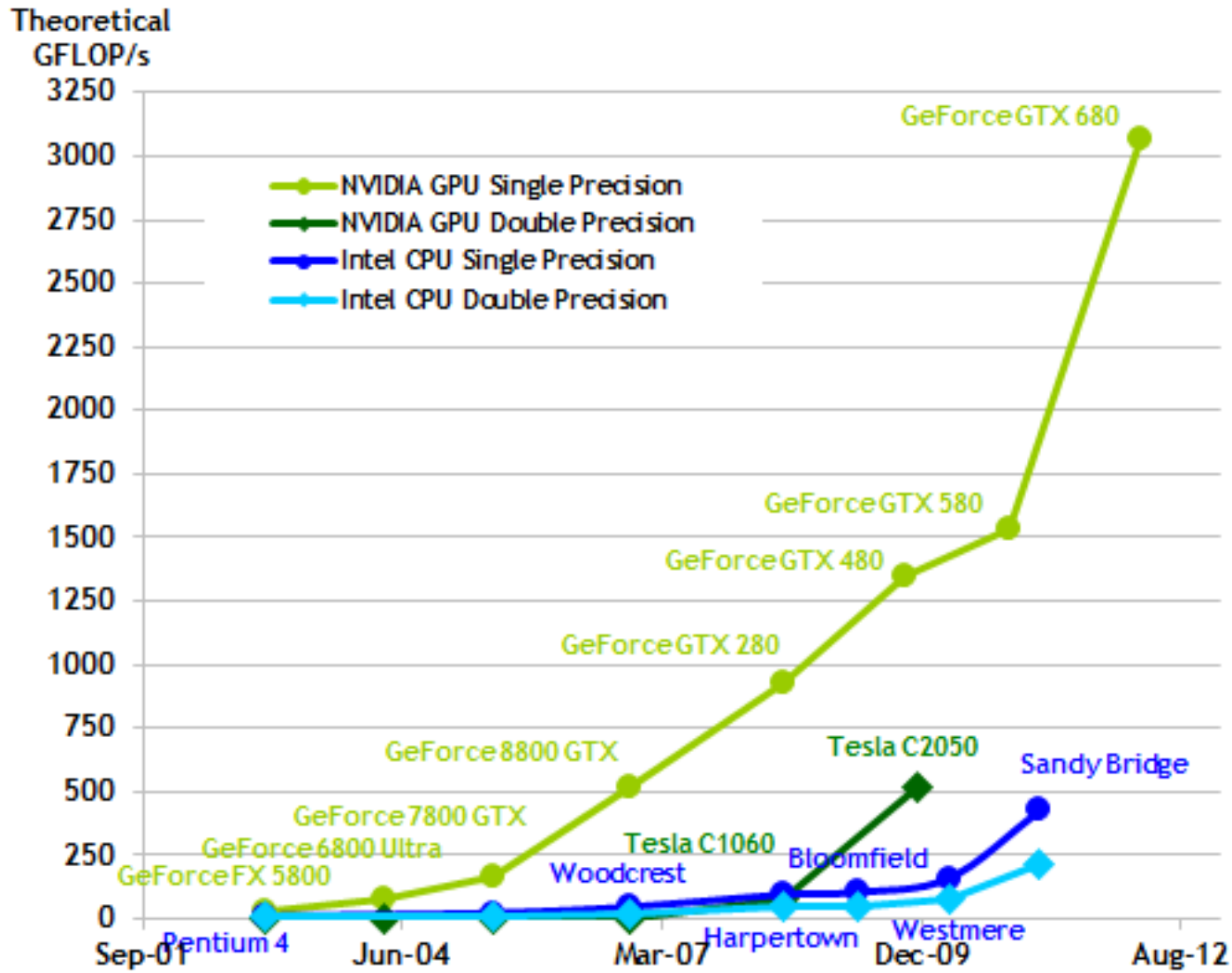
- CPU : optimisé pour exécuter rapidement une série de tâches de tout type.
- GPU : optimisé pour exécuter des opérations simples sur de très gros volumes de données.
  $\Rightarrow$ **beaucoup plus de transistors dédiés au calcul sur les GPUs et moins aux unités de contrôle**.





Plan

Pourquoi utiliser des GPUs pour le HPC ?



Pourquoi utiliser des GPUs pour le HPC ?

A performance égale, les GPUs sont*, par rapport aux CPUs :

- des solutions plus denses (9× moins de place) ;
- moins consommateurs d'électricité (7× moins) ;
- moins chers (6× moins).

∗ Chiffres issus de "*High Performance Computing : are GPU going to take over ?*", A. Nedelcoux, Université du SI, 2011.

Ces chiffres ont été obtenus sur une application réelle et varient d'une application à l'autre.

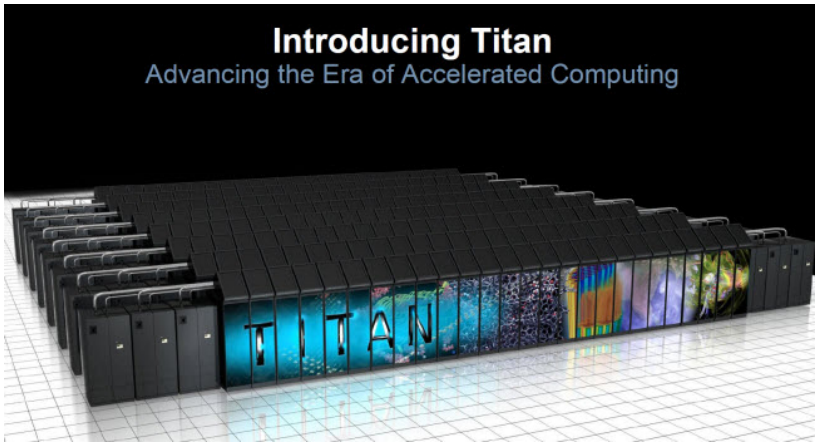
Cartes graphiques

- Cartes NVIDIA.
  Chaque architecture (Tesla, Fermi, Kepler, Maxwell...) est déclinée en 3 familles :
  - GeForce (graphisme grand public) ;
  - Quadro (visualisation professionnelle) ;
  - Tesla (orientée GPGPU).
- Cartes AMD (ATI).
  - Radeon ;
  - ...

Top500 (11/2012), supercalculateur #1

Titan (Cray XK7, USA)

- $R_{peak} = 27.1\ PFlops$.
  $R_{max} = 17.5\ PFlops$
  (Linpack).
- 560640 cœurs.
  - 18688 Procs. AMD Opteron de 16 cœurs à 2.2 *GHz* ;
  - 18688 **GPUs Nvidia K20x** de 14 SMx à 732 *MHz*.
- 710 *TiB* de ram,
  10 *PiB* de disque.
- 8.2 *MW*.

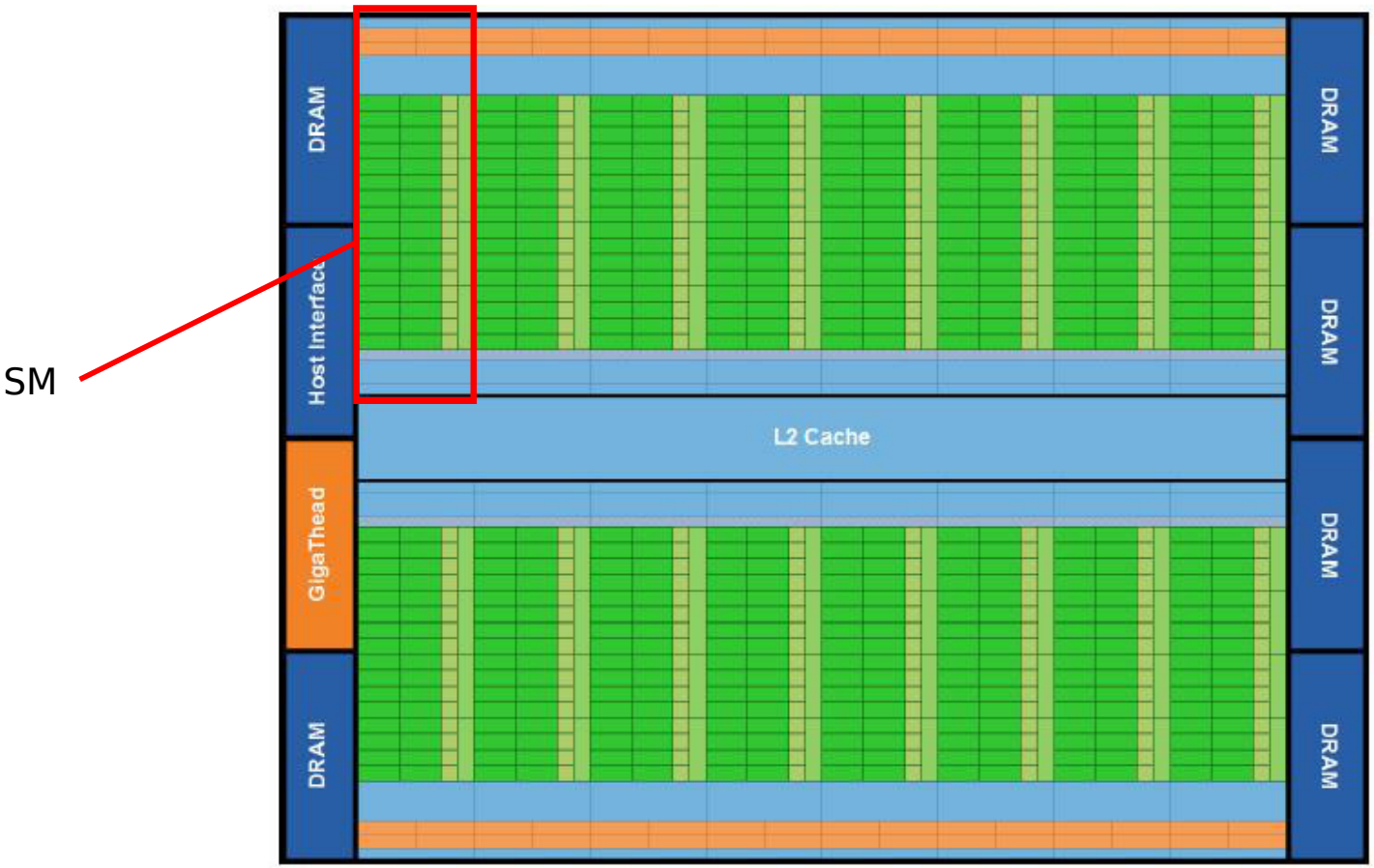


TOP500 supercomputer sites :
http ://www.top500.org/

Plan

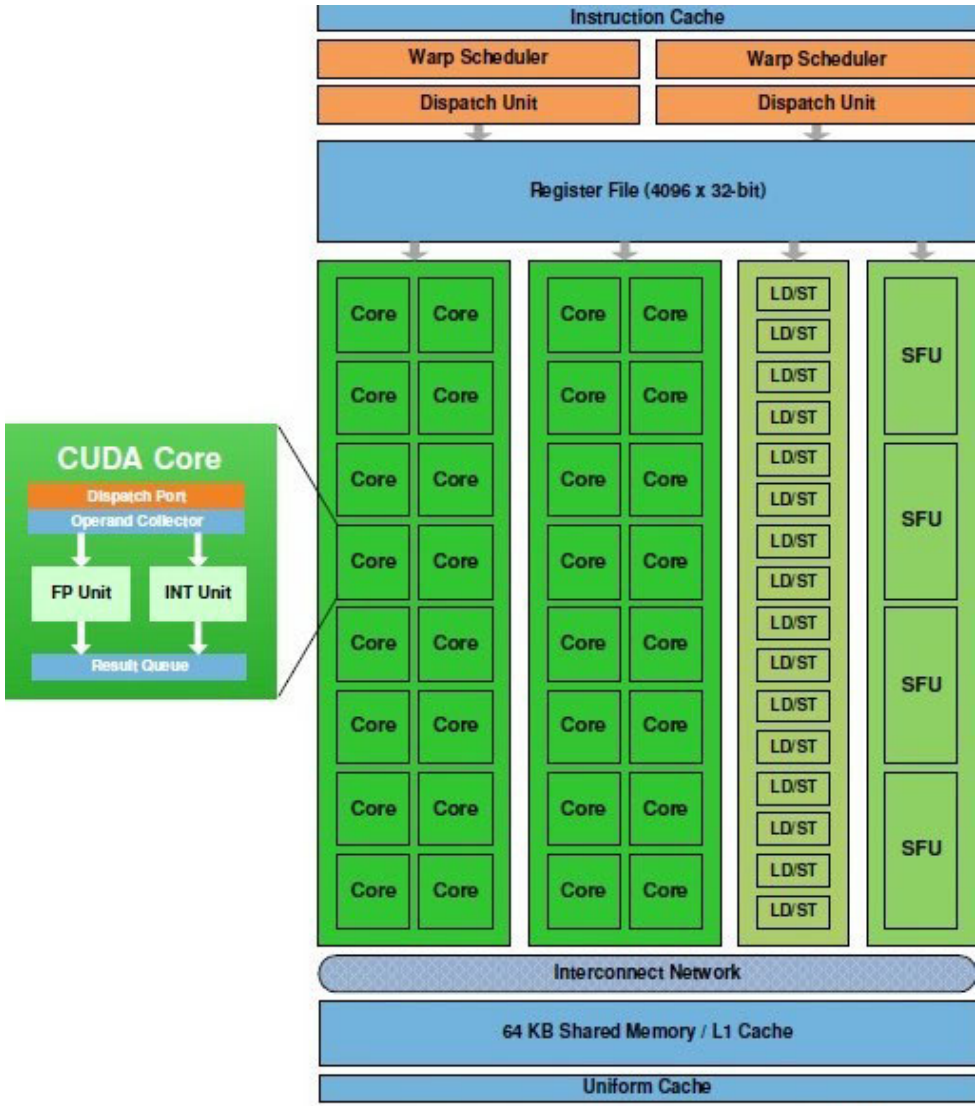
Architecture Fermi (NVIDIA)

- Jusqu'à 16 Streaming Multiprocessors (SM) de 32 cœurs CUDA soit 512 cœurs.



SM *Streaming Multiprocessor*

- Regroupe 32 cœurs ayant des unités de calcul 64 *bits*.
- Ordonnance les threads par paquets (via le *Warp Scheduler*).
- Dispose de registres, de mémoire partagée et de caches.



Fermi Streaming Multiprocessor (SM)

Plan

Hiérarchie mémoire

- Mémoires sur la puce :
  - **registres**, très rapides (RW), 32 *bits* (32768 par SM) ;
  - **mémoire partagée** (RW), jusqu'à 48 *KiB* par SM.
- Mémoires hors de la puce :
  - **mémoire globale** (RW), quelques *GiB*, lente (quelques centaines de cycles d'horloge) ;
  - mémoire de texture (RO) ;
  - mémoire de constantes (RO) : accès rapide ;
  - mémoire locale (RW).

Transferts entre mémoires CPU-GPU

- **Le GPU a sa propre mémoire** (mémoire globale).
  Bande passante (BP) de l'ordre de 100 GiB/s.
  Exemple : carte Tesla 2050 avec RAM à 1500 MHz, $BP = 144\ GiB/s$.

- En comparaison, la bande passe mémoire pour un CPU est de l'ordre de 10 GiB/s.
  Exemple : proc. Nehalem avec RAM à 1066 MHz, $BP = 25.6\ GiB/s$.

- Le GPU n'accède pas à la mémoire centrale du CPU.
  Les transferts entre la mémoire du CPU et celle du GPU se font via le bus Pci-express.
  BP de l'ordre de 10 GiB/s et grande latence $\Rightarrow$ pénalisant !
  Exemple : Pci-express x16, Gen2, $BP = 8\ GiB/s$.

Qu'est-ce que le GPGPU ?

GPGPU : *General-Purpose computing on Graphics Processing Units*

3 approches pour utiliser le GPU pour faire du HPC.

- Bibliothèques (cuBLAS, MAGMA, Thrust...)
- Langages
  - "Legacy GPGPU" (avant CUDA, vers 2004)
  - CUDA (2007)
  - OpenCL (2008)
- Directives
  - OpenHMPP (2007)
  - OpenACC (2011)

Exemple de configuration test pour GPGPU

Nœud test "igloo-gpu" du mesocentre ECP :

- 1 nœud Altix ayant 2 processeurs quadri-cœurs Nehalem-EP à 2.26 *GHz* et 48 *GiB* de mémoire ;
- + 1 carte graphique Tesla M2050 (14 SMs de 32 cœurs soit 448 cœurs) à 1.15 *GHz* et 3 *GiB* de mémoire ;
- CUDA (Driver / Runtime / Capability : 5.0 / 5.0 / 2.0) ;
- PGI version 11.8 pour Cuda-Fortran.

Un PC avec une carte graphique peut également convenir !

Configurations pour GPGPU en France

- Crihan (mesocentre de Haute-Normandie) : Antares (IBM), 13 *TFlops* en hybride, 28 GPUs NVIDIA ;
- CCRT / TGCC (Genci) : Curie (BULL), 200 *TFlops* en hybride, 288 GPUs NVIDIA ;

- CCRT / TGCC (Genci) : Titane (BULL), 192 *TFlops* (SP) en hybride, 192 GPUs NVIDIA.
- ...

Plan

CUDA

CUDA : *Compute Unified Device Architecture*

- Développé par NVIDIA depuis 2007.
- Fait référence à l'architecture et au modèle de programmation.
- Permet de programmer des GPUs en C* / C++.
- Octobre 2012 : Cuda 5.0.

* PGI propose un compilateur pour Cuda-Fortran.

Prérequis

- Installation
  - Driver ;
  - CUDA Toolkit (compilateur, outils de profilage, bibliothèques cuBLas, cuFFT...) ;
  - CUDA SDK (*Software Development Kit* : exemples de codes...)
    Exemple : le programme `bandwidthTest` permet d'effectuer des mesures de bande passante CPU-GPU.

- CUDA fonctionne avec les cartes *NVIDIA** GeForce, Quadro et Tesla.

* possibilité d'utilisation avec des GPUs AMD via l'outil opensource Ocelot.

Un peu de vocabulaire

- **Host** : CPU.
- **Device** : GPU.
- **Kernel** : fonction s'exécutant sur le GPU.

Eléments clé d'un programme CUDA

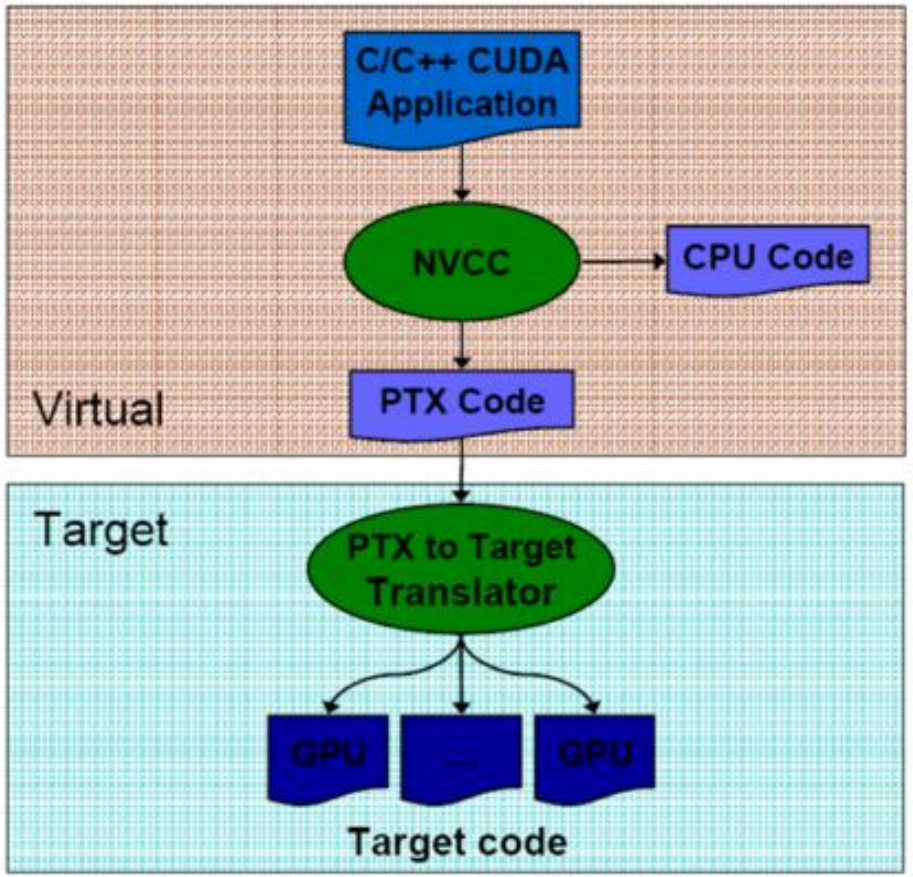
- **Le programme contient le code pour le host et le device** (extension `.cu`).
- Allocation de mémoire sur le GPU, transferts entre mémoires...
- Lancement du kernel sur le device par le host en précisant une grille d'exécution (contexte).

Exemple de programme simple CUDA

```
#include <stdio.h>
/* a simple CUDA kernel */
__global__ void add( int a, int b, int *c ) {
  *c = a + b;
}

/* main */
int main( void ) {
  int c;
  int *dev_c;

  // GPU device memory allocation
  cudaMalloc( (void**)&dev_c, sizeof(int) ) ;

  // perform computation on GPU
  add<<<1,1>>>( 2, 7, dev_c );

  // get back computation result into host CPU memory
  cudaMemcpy( &c, dev_c, sizeof(int), cudaMemcpyDeviceToHost ) ;

  // output result on screen
  printf( "2 + 7 = %d\n", c );

  // de-allocate GPU device memory
  cudaFree( dev_c ) ;

  return 0;
}
```

Compilateur nvcc

```
nvcc programme.cu -o programme
```



Modèle de programmation

- Chaque SM a en charge la création, l'organisation et l'exécution de threads.
- Les threads exécutent simultanément des *kernels*.
- D'un point de vue matériel, les threads sont organisés en *warp* (paquets de 32).
- Chaque SM peut contrôler jusqu'à 48* warps simultanément, les warps en attente d'un accès mémoire sont mis en attente.

$\Rightarrow$ Accès mémoire globale sur GPU masqués par calculs ; optimisation de l'utilisation des ressources.

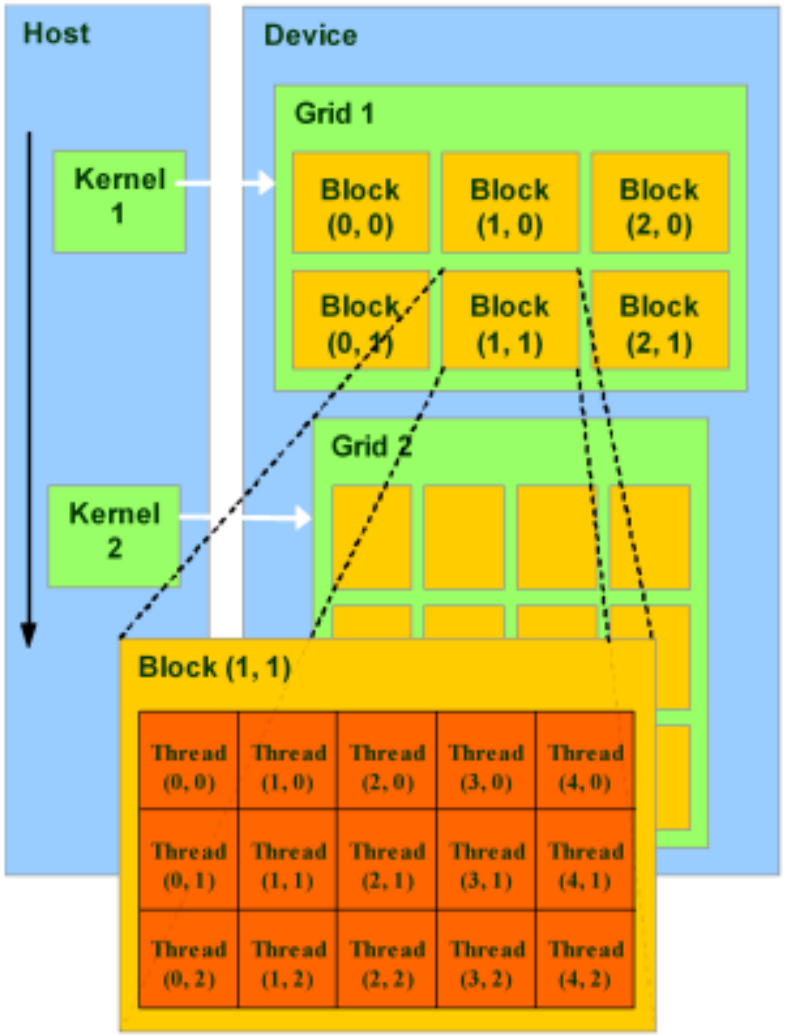
* dépend de la version de capacité de calcul ("compute capability").

Organisation (logique) des threads en un ensemble structuré

Grille 3D^* de blocs 3D de threads

- Dimensions de la grille en blocs : `gridDim(.x,.y,.z)` ;
- Dimensions des blocs en threads : `blockDim(.x,.y,.z)` ;
- Indice d'un bloc dans la grille : `blockIdx(.x,.y,.z)` ;
- Indice d'un thread dans son bloc : `threadIdx(.x,.y,.z)`.

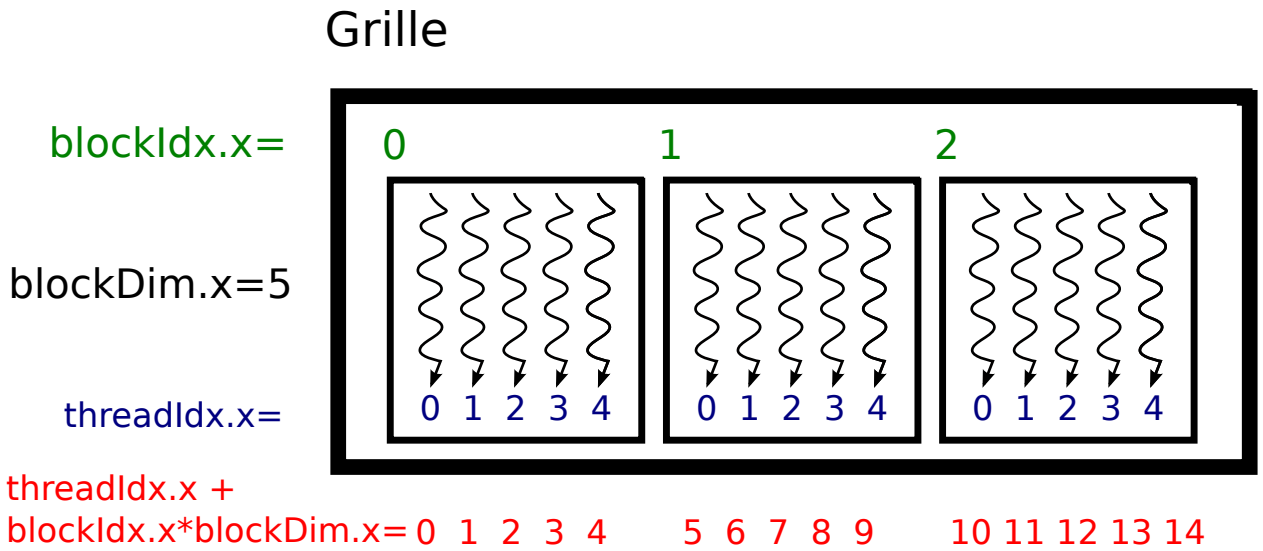
* dépend de la version de capacité de calcul.



Index d'un thread

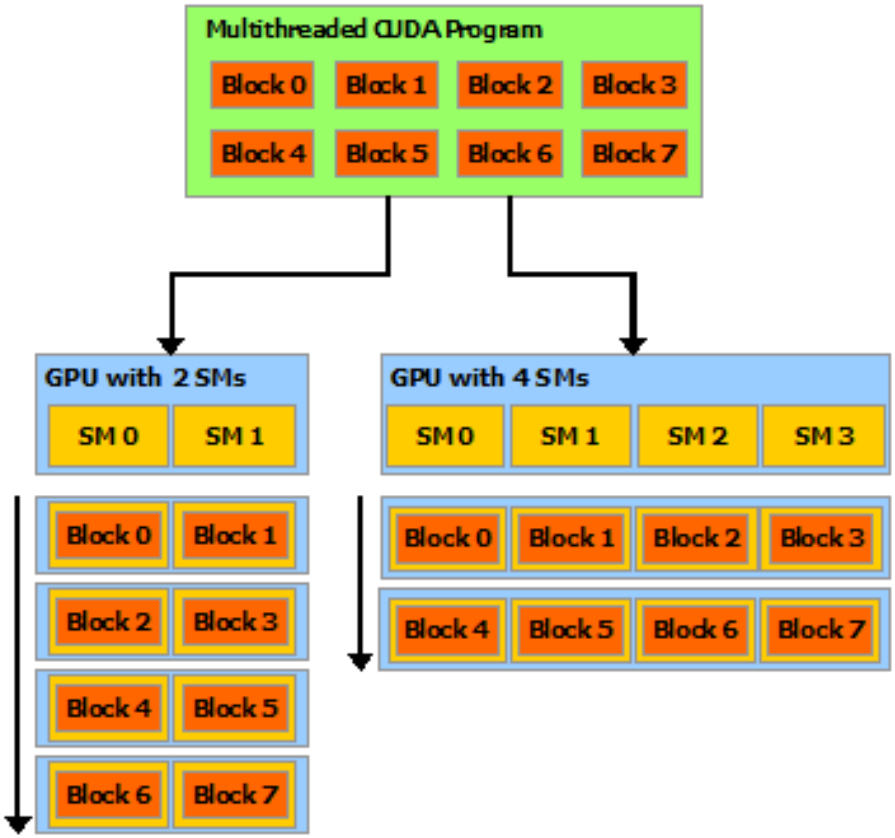
- Un thread d'une grille est identifié par un index global unique. Exemple en 1D :

```
index = threadIdx.x + blockIdx.x*blockDim.x;
```

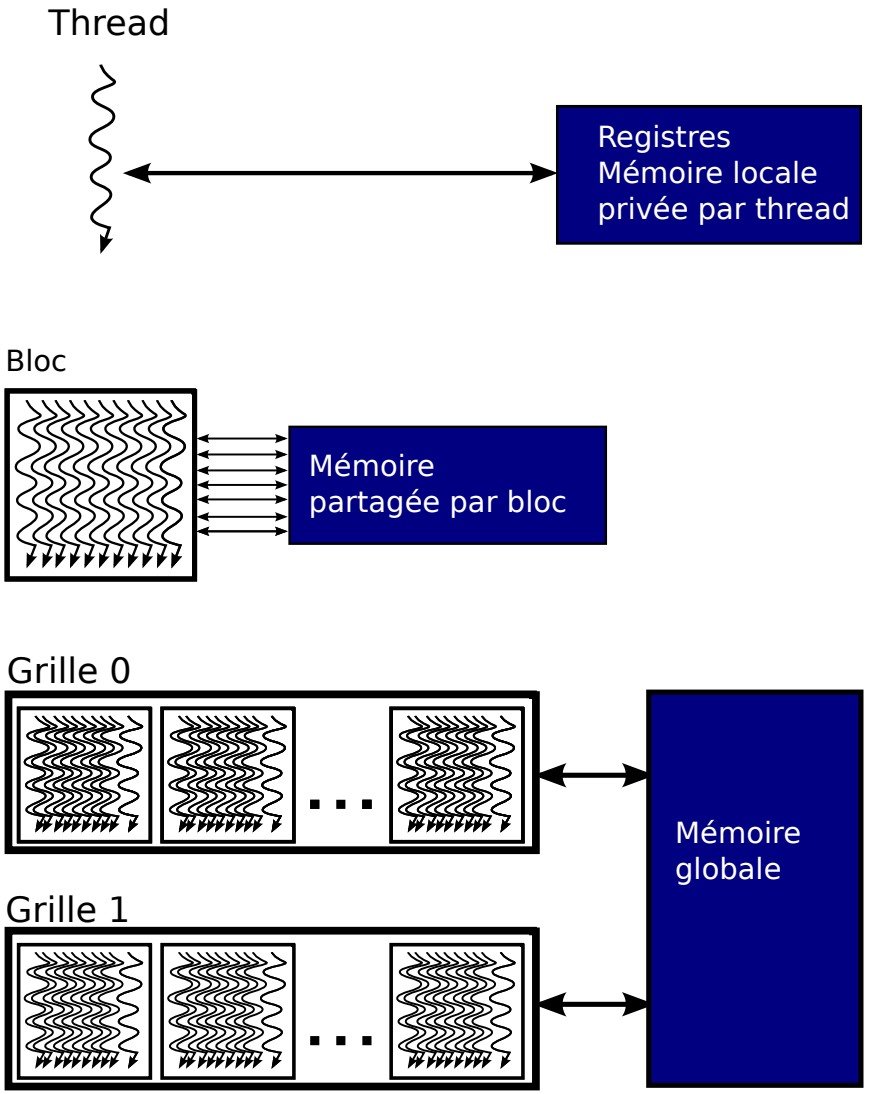


- L'index permet d'établir une correspondance entre le thread et les données (éléments *i* d'un tableau 1D par exemple...).

Répartition des blocs sur les SM et extensibilité



Modèle mémoire



Transferts entre mémoires du host et du device

- Allocation / désallocation mémoire sur le device

```
cudaError_t cudaMalloc(void** devPtr, size_t count);
cudaError_t cudaFree(void* devPtr);
```

- A comparer à l'allocation / désallocation de mémoire sur le CPU

```
void *malloc(size_t size);
void free(void *ptr);
```

Transferts entre mémoires du host et du device

- Copies entre mémoires du host et du device

```
cudaError_t cudaMemcpy(void* dst, const void* src,
                       size_t count,
                       enum cudaMemcpyKind kind);
```

copie `count` *bytes* de la zone mémoire pointée par `src` à celle pointée par `dst`.
`kind` indique la direction de la copie (`cudaMemcpyHostToDevice`, `cudaMemcpyDeviceToHost`...).

- Les transferts (allocations, copies dans les 2 directions...) sont contrôlés par le CPU.

Exemple de copie vers le device

```
int main( void ) {
  // array size
  int N = 16;
  // host variable
  int *a
  // device variable
  int *dev_a

  // CPU memory allocation / initialization
  a = (int *) malloc(N*sizeof(int));
  for (int i=0; i<N; i++) {
    a[i]=i;
  }

  // GPU device memory allocation / initialization
  cudaMalloc( (void**)&dev_a, N*sizeof(int) ) ;
  cudaMemcpy( dev_a, a, N*sizeof(int),
                              cudaMemcpyHostToDevice ) ;
```

Kernel

Fonction C lancée depuis le host et tournant sur le device :

- de type `__global__` ;
- retournant un `void` ;
- lancée à l'aide d'une syntaxe C modifiée de type :

```
myKernel<<< gridDim, blockDim >>>(parameters);
```

`gridDim` : dimensions de la grille en blocs, `blockDim` : dimensions des blocs en threads

- Tout kernel a accès automatiquement aux variables `gridDim`, `blockDim`, `blockIdx` et `threadIdx`.

Exemple de kernel CUDA (addition de vecteurs)

```
/* a simple CUDA kernel */
__global__ void add( int *a, int *b, int *c, int n ) {

  int i = threadIdx.x + blockIdx.x*blockDim.x;
  if (i<n)
    c[i] = a[i] + b[i];
}
```

- A comparer à l'équivalent sur le host :

```
/* Version CPU pour comparaison */
  void add( int *a, int *b, int *c, int n ) {

  int i ;
  for  (i = 0; i < n; ++i) {
    c[i] = a[i] + b[i]; }
}
```

- Appel du kernel par le host :

```
int nbThreads = 8;

dim3 blockSize(nbThreads,1,1);

dim3 gridSize(N/nbThreads,1,1);

add<<<gridSize,blockSize>>>( dev_a, dev_b, dev_c, N );
```

Quelques questions fréquentes...

- Combien de threads mettre par bloc ?
- Combien de blocs ?
- Combien de mémoire partagée accessible par bloc ?
- ...

- Pour répondre à ces questions, une bonne connaissance des caractéristiques de sa carte (nombre maximum de threads par bloc...) est nécessaire !

  Ces informations peuvent être récupérées dans le code par la fonction `cudaGetDeviceProperties` ou affichées via `deviceQuery` (SDK).

deviceQuery

```
deviceQuery Starting...
 CUDA Device Query (Runtime API) version (CUDART static linking)
Detected 1 CUDA Capable device(s)
Device 0: "Tesla M2050"
  CUDA Driver Version / Runtime Version          5.0 / 5.0
  CUDA Capability Major/Minor version number:    2.0
  Total amount of global memory:                 2687 MBytes (2817982464 bytes)
  (14) Multiprocessors x ( 32) CUDA Cores/MP:    448 CUDA Cores
  GPU Clock rate:                                1147 MHz (1.15 GHz)
  Memory Clock rate:                             1546 Mhz
  Memory Bus Width:                              384-bit
  L2 Cache Size:                                 786432 bytes
(...)
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 32768
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  1536
  Maximum number of threads per block:           1024
  Maximum sizes of each dimension of a block:    1024 x 1024 x 64
  Maximum sizes of each dimension of a grid:     65535 x 65535 x 65535
  Maximum memory pitch:                          2147483647 bytes
(...)
  Device has ECC support:                        Enabled
(...)
```

A voir aussi...

De nombreux points n'ont pas été abordés ici :

- L'utilisation des différents types de mémoires (registres, mémoire partagée, mémoire de constantes...) ;
- La notion de flux (*CUDA Streams*) ;
- Les aspects de synchronisation entre threads ;
- ...

Plan

OpenCL

OpenCL *Open Computing Language*

- Standard libre maintenu par Khronos Group depuis 2008.
- Projet initié par Apple.
- Nombreux partenaires investis dans le développement (AMD, NVIDIA, Intel...).
- Permet de programmer des systèmes hétérogènes combinant par exemple CPU et GPU (NVIDIA, AMD...).
- API et langage de programmation dérivé du C, utilisable dans des codes C / C++ et Python (PyOpenCL).
- 2011 : OpenCL 1.2

OpenCL *vs* CUDA

- OpenCL considéré comme plus complexe mais plus puissant.
- OpenCL permet de programmer plus d'accélérateurs (HWAs *HardWare Accelerator* ; GPUs NVIDIA et AMD, FPGA, DSP...), des processeurs Cell *etc*.
- Modèle mémoire et modèle d'exécution hérités de CUDA (work-item = threads de CUDA, work-group = bloc, ND-Range = grille...).
- Connaissance préalable de CUDA conseillée.

Etapes d'un code OpenCL

Etapes typiques à effectuer dans le code de l'hôte :

- Détection et sélection des devices (fonctions `clGetPlatformIDs`, `clGetDeviceIDs`...) ;
- Initialisation du contexte OpenCL (`clCreateContext`) ;
- Création d'une file de commandes (`clCreateCommandQueue`) ;
- Création de buffers mémoire sur le device (`clCreateBuffer`) ;

Etapes d'un code OpenCL (suite)

- Création d'une collection de kernels ("cl program") à partir des sources des kernels, compilation "just in time", création d'objets kernel (`clCreateProgramWithSource`, `clBuildProgram`, `clCreateKernel`) ;
- Initialisation des arguments (`clSetKernelArg`) ;
- Définition du domaine d'exécution (variable `global_work_size`...) ;
- Lancement du kernel (`clEnqueueNDRangeKernel`) ;
- Récupération des résultats (`clEnqueueReadBuffer`).

Exemple de kernel OpenCL

```
__kernel void vector_add_gpu (__global const float* src_a,
                              __global const float* src_b,
                              __global float* res,
                             const int num)
{
   /* ID of the thread in execution */
   const int idx = get_global_id(0);

   if (idx < num)
      res[idx] = src_a[idx] + src_b[idx];
}
```

- Compilation du code avec gcc (par exemple) et édition des liens avec `libopencl` (le compilateur OpenCL est "embarqué" dans la bibliothèque).

Plan

OpenHMPP

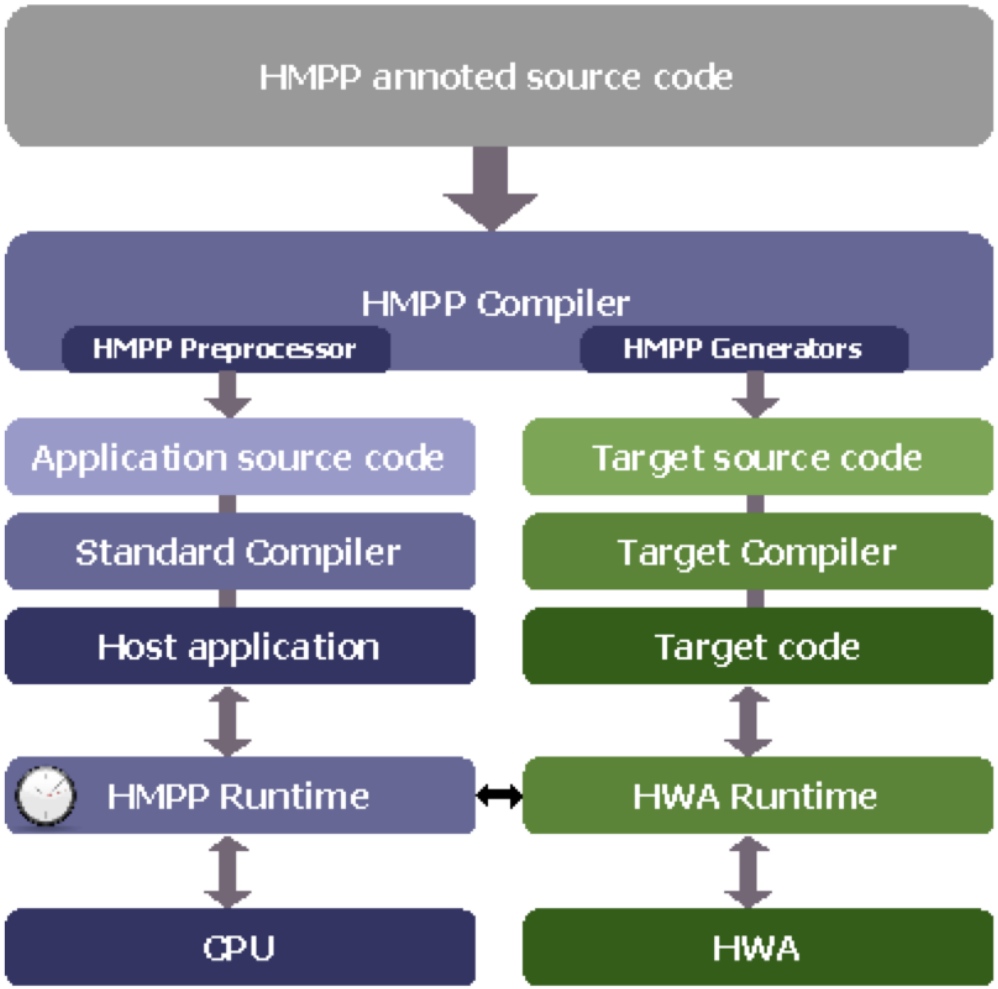
OpenHMPP
Open Hybrid Multicore Parallel Programming

- Standard opensource initié en 2007, développé par CAPS et adopté par PathScale.
- OpenHMPP repose sur les directives HMPP version 2.3 (2009).
- Compilateurs : HMPP (CAPS) et PathScale ENZO compiler suite.

OpenHMPP *vs* CUDA et OpenCL

- Permet de programmer les architectures parallèles hybrides avec différents types de HWAs (GPU NVIDIA, GPU AMD, FPGAs...).
- Repose sur des directives.
- Permet de générer, entre autres, des *codelets* CUDA ou OpenCL.
- Plus haut niveau d'abstraction que CUDA et OpenCL.
  Plus simple à utiliser, il est préférable cependant de connaître CUDA ou OpenCL pour tirer parti de toutes ses possibilités.
- Utilisable dans les codes C, C++ et Fortran.

Du code à l'exécution sur le CPU et le HWA...



Exemple de programme simple OpenHMPP

```
#pragma hmpp myCall codelet, target=CUDA
void myFunc( int n, int A[n], int B[n])
{
    int i;
    for (i=0 ; i<n ; ++i)
        B[i] = A[i] + 1;
}
void main(void)
{
    int X[10000], Y[10000], Z[10000];
    ...
    #pragma hmpp myCall callsite
    myFunc(10000, X, Y);
    ...
}
```

- Compilation : `hmpp gcc programme.c`

Directives OpenHMPP

- Permettent d'indiquer les fonctions à exécuter sur le HWA, gérer leur appel et les transferts de données entre mémoires...
- Syntaxe C

```
#pragma hmpp [codelet_label] directive_type [,parmameters] [&]
```

- Syntaxe Fortran

```
!$hmpp [codelet_label] directive_type [,parameters] [&]
```

codelet et callsite

- 2 directives importantes : `codelet` et `callsite`
- `codelet` identifie la fonction à porter sur l'accélérateur.
  - le paramètre `target` permet de spécifier le langage cible (CUDA, OpenCL...).

```
#pragma hmpp myLabel codelet, target=CUDA, ...
void myFunc(...){
... }
```

- `callsite` spécifie l'usage d'un codelet à un point donné dans le programme.

```
...
#pragma hmpp myLabel callsite, asynchronous, ...
myFunc(...);
...
```

Plan

OpenACC

OpenACC

- Standard ouvert.
- Consortium : depuis 2011, CAPS, Cray, PGI (Portland Group Inc) et NVIDIA.
  Depuis 2012, Allinea, Georgia Tech, ORLN...
- Décembre 2012 : OpenACC 1.0.
- Compilateurs supportant OpenACC : PGI (à partir de 12.6), celui de Cray et celui de CAPS (HMPP, depuis 3.1).

OpenACC *vs* OpenHMPP

- Comme OpenHMPP, OpenACC
  - Repose sur des directives.
  - Utilisable dans les codes C, C++ et Fortran.
  - ..
- OpenACC est actuellement moins riche que OpenHMPP
  - Pas de gestion de devices multiples.
  - Pas d'utilisation de bibliothèques pour accélérateurs par des directives.
  - Pas d'intégration possible de kernels CUDA déjà écrits.
  - ...

Exemple de programme simple OpenACC

- Exemple sur le calcul de π (tiré de Nvidia Developper Zone)

```
program picalc
    implicit none
    integer, parameter :: n=1000000
    integer :: i
    real(kind=8) :: t, pi
    pi = 0.0
!$acc parallel loop reduction(+:pi)
    do i=0, n-1
        t = (i+0.5)/n
        pi = pi + 4.0/(1.0 + t*t)
    end do
!$acc end parallel loop
    print *, 'pi=', pi/n
end program picalc
```

- Compilation avec hmpp : `hmpp gfortran programme.f90`

Directives OpenACC

- Permettent d'indiquer les parties du code à exécuter sur le HWA, gérer les transferts de données entre mémoires...
- Syntaxe C

```
#pragma acc directive-name[clause[[,]clause]...]
{
...
}
```

- Syntaxe Fortran

```
!$acc directive-name[clause[[,]clause]...]

!$acc end directive-name
```

Constructions kernels, parallel

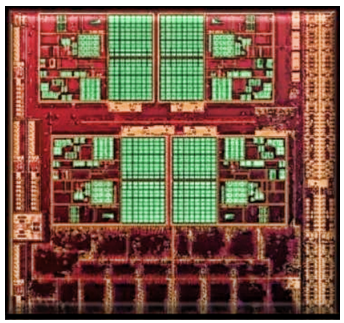
- Directives importantes pour le parallèlisme : `parallel`, `kernels` et `loop`
- Gangs et workers : clauses `num_gangs[]`, `num_workers[]`

```
#pragma acc parallel, num_gangs[32], num_workers[256]
{
  #pragma acc loop gang
  for(i=0; i < n; i++) {
     ...
  }
...
}
```

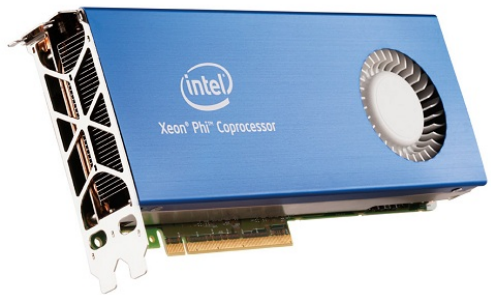
- Directive `data` (clauses `create`, `copy`, `pcopy`...) pour la gestion des données.

Accélérateurs : nouvelles tendances

- Chez AMD : APU (*Accelerated Processing Unit*)
  AMD Fusion : CPU et GPU sur une même puce avec partition mémoire.
- Chez NVIDIA : Denver Project, Tegra
  (CARMA : kit de développement CUDA pour les systèmes ARM accélérés par les GPUs NVIDIA).
- Chez Intel : Co-processeur Xeon-Phi
  (Mic : *Intel Many Integrated Core Architecture*).



Puce AMD Fusion



Xeon Phi

Conclusion

- Architectures évoluant rapidement (lien avec les systèmes embarqués).
- Plusieurs solutions de programmation souvent récentes et en évolution.
- Certains solutions permettent de programmer plusieurs types de GPUs et d'autres HWAs mais **problème de "portabilité de la performance"**.

Références

- *Introduction to GPU computing with CUDA*, P. Kestener, PATC Training, Maison de la Simulation, 2012.
- *OpenCL, an introduction*, G. Colin de Verdière, Formation "Initiation à la programmation des GPU", Maison de la Simulation, 2011.
- *Programmation par directives pour le calcul hybride : HMPP et OpenACC*, CAPS, Ecole "Programmation hybride : une étape vers le many-coeurs ?", Groupe Calcul, Autrans, 2012 (disponible sur http ://calcul.math.cnrs.fr).

Références (suite)

- Site de NVIDIA (CUDA) : http ://www.nvidia.com
- Site de Khronos (OpenCL) : http ://www.khronos.org/opencl/
- Site de CAPS (HMPP) : http ://www.caps-entreprise.com
- Site de OpenACC : http ://www.openacc-standard.org/