Modèles et techniques en programmation parallèle hybride et multicœurs

Programmation parallèle de plusieurs GPUs

Marc Tajchman

CEA - DEN/DM2S/STMF/LMES

01/02/2023

Intérêt d’utiliser plusieurs GPUs :

- accélérer le calcul

    faire attention au coût de transferts de données entre GPUs

- traiter des cas plus gourmands en mémoire

    la mémoire associée à un GPU est souvent plus petite que celle associée au CPU

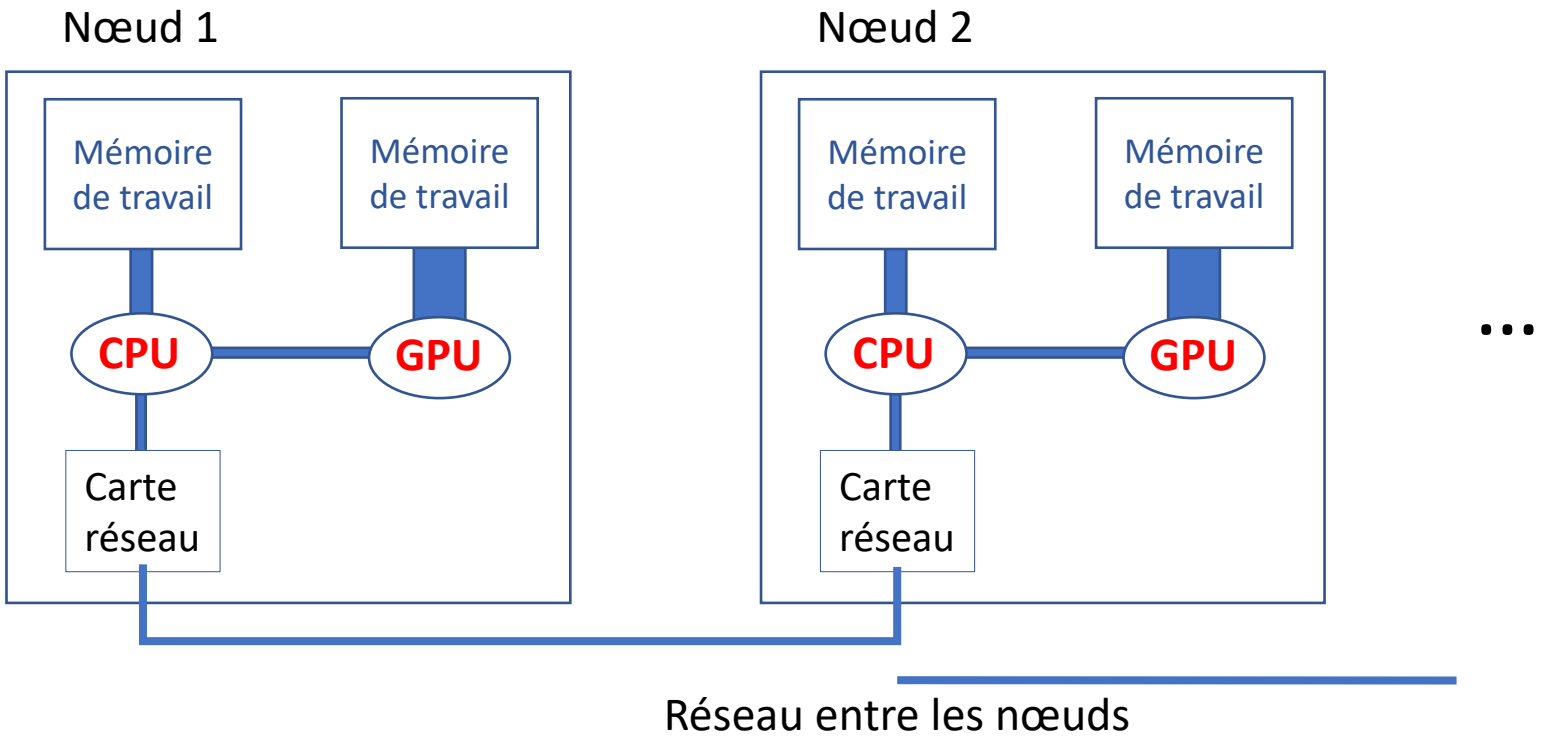
Dans ce qui suit, des illustrations sont reprises de

https://developer.nvidia.com/blog/introduction-cuda-aware-mpi

Première configuration:

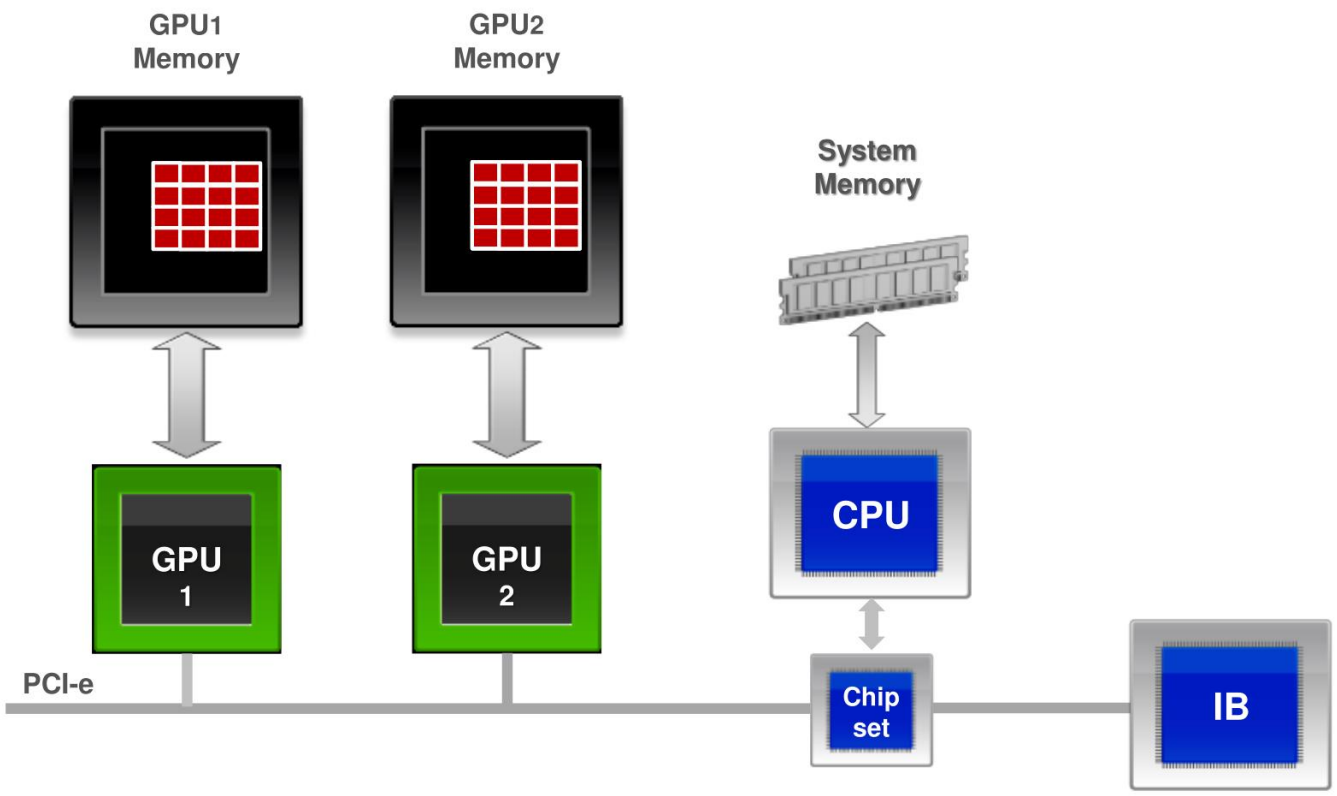
On considère une machine parallèle avec n nœuds, chacun contenant un (ou plusieurs) CPU et un (ou plusieurs) GPU.

Dans chaque nœud, le(s) CPU et le(s) GPU sont associés à leur mémoire de travail propre.



Deuxième configuration :

Un CPU est attaché à plusieurs GPUs



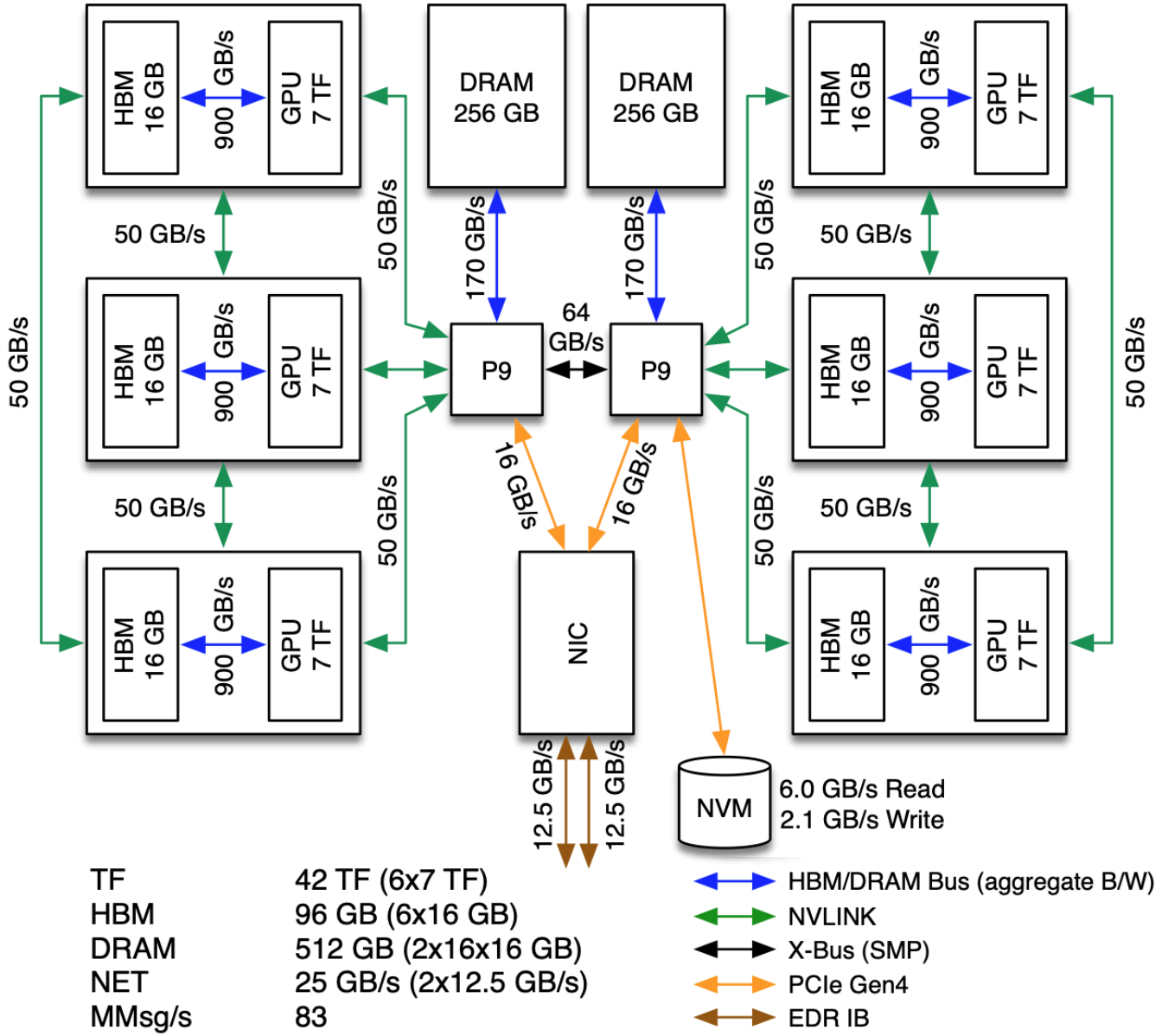
Troisième configuration :

Combinaison des 2 premières : machine parallèle contenant n nœuds, chacun contenant plusieurs cœurs CPU et un ou plusieurs GPU

Exemple : machine Summit
(Oak Rigde Nat. Lab, USA)

4806 nœuds chacun avec 2 CPU (IBM Power PC) + 6 GPU (Nvidia Volta)

Schéma de l'intérieur d'un nœud:



Attention : les GPUs possèdent leur propre mémoire de travail, si un GPU doit accéder à des données contenues dans la mémoire d'un autre GPU, il faut transférer ces données d'une mémoire à l'autre, y compris si les 2 GPUs sont situés dans le même nœud de calcul d'une machine parallèle.

La mémoire est partagée par les cœurs CPU mais pas par les GPU.

Exemple de programmation dans la 2ème configuration:
Calcul d'un vecteur de taille nTotal (dans la mémoire du CPU), par plusieurs GPU

```
int nTotal = …
double * A_h;
cudaMallocHost(&A_h, nTotal * sizeof(double));
```

Récupération du nombre de GPU associés au CPU

```
int nTotal = …
double * A_h;
cudaMallocHost(&A_h, nTotal * sizeof(double));
int nDevices;
cudaGetDeviceCount(&nDevices);
```

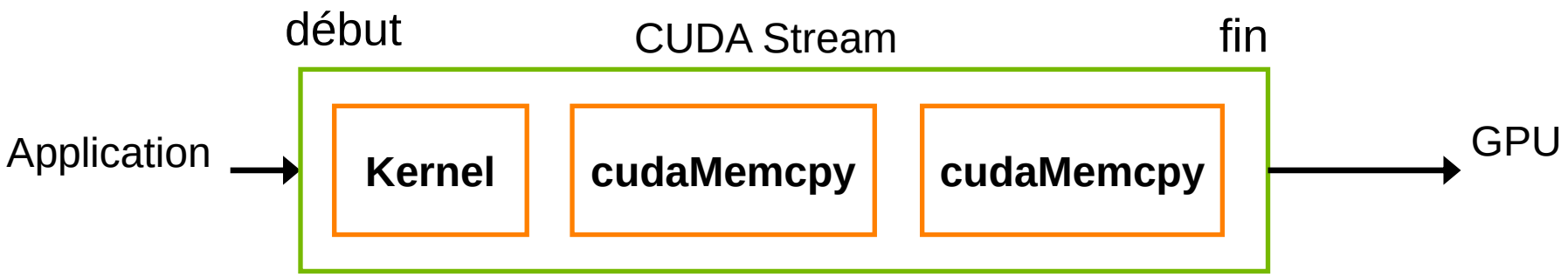
Création de streams cuda et réservation de mémoire sur chaque GPU

```
int nTotal = …
double * A_h;
cudaMallocHost(&A_h, nTotal * sizeof(double));
int nDevices;
cudaGetDeviceCount(&nDevices);

cudaStream_t * stream = new cudaStream_t[nDevices];
double **A_d = new double *[nDevices];
size_t nPartiel = nTotal/nDevices;
for (i = 0; i < nDevices; i++)
{
  cudaSetDevice(i);
  cudaStreamCreate(stream[i]);
  cudaMalloc(&(A_d[i]), nPartiel * sizeof(double));
}
```

Cuda stream :

- Suite d’opérations de copie mémoire, de lancement de noyau sur un GPU.
- Un Cuda stream est associé à un GPU et un seul.
- Plusieurs Cuda streams peuvent être associés au même GPU, ou à différents GPU.
- Un Cuda stream est exécuté indépendamment des autres Cuda streams



Dans cet exemple, nDevices Cuda streams sont créés, un pour chaque GPU.

Les opérations dans chaque Cuda stream, peuvent être exécutées en même temps.

Lancement d'un noyau Cuda sur chaque GPU de façon asynchrone

```
for (i = 0; i < nDevices; i++)
{
  cudaSetDevice(i);

  cudaMemcpyAsync(A_d[i], A_h + nPartiel * i, nPartiel * sizeof(double),
                                cudaMemcpyHostToDevice, stream[i]);

  noyau<<<gridSize, blockSize, 0, stream[i] >>> (A_d[i], nPartiel);


  cudaMemcpyAsync(A_h + nPartiel * i, A_d[i], nPartiel * sizeof(double),
                                cudaMemcpyDeviceToHost, stream[i]);
}
```

On termine en vérifiant que tous les GPU ont terminé leurs calculs et les transferts :

```
for (i = 0; i < nDevices; i++)
{
  cudaSetDevice(i);
  cudaStreamSynchronize(stream[i]);
  cudaDeviceSynchronize();
}

// ne pas oublier de libérer correctement la mémoire
// (entr'autres: appeler cudaStreamDestroy pour chaque cuda stream)
```