



GP-GPU

Feedback on hybrid programming (CUDA + OpenMP/MPI)

Stéphane Vialle



université
PARIS-SACLAY

ÉCOLE DOCTORALE
Sciences et technologies
de l'information
et de la communication (STIC)



Stephane.Vialle@centralesupelec.fr
<http://www.metz.supelec.fr/~vialle>

1



Feedback on hybrid programming (CUDA + OpenMP/MPI)

1 – Produit de matrices sur cluster de GPU

- Algorithme en anneau sur cluster de PC
- Algorithme sur cluster de GPU – v1
- Algorithme sur cluster de GPU – v2
- Performance sur cluster de GPU

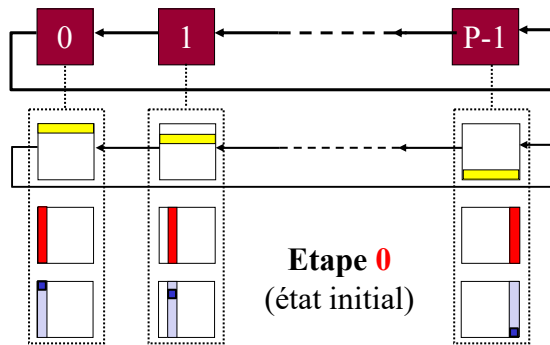
2 – Parallélisation simultanée sur CPUs et GPUs

2

Algorithme en anneau sur cluster de PC

Principe algorithmique sur cluster ($C = A \times B$)

- A partitionnée en blocs de lignes
- B et C partitionnées en blocs de colonnes
- **Circulation de A**
- B et C statiques



Topologie

Partitionnement
et circulation de A

Partitionnement
statique de B

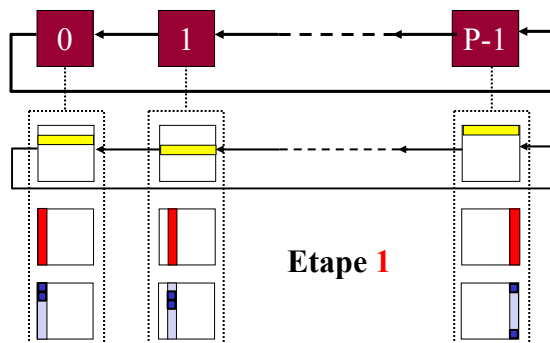
Partitionnement
statique de C

3

Algorithme en anneau sur cluster de PC

Principe algorithmique sur cluster ($C = A \times B$)

- A partitionnée en blocs de lignes
- B et C partitionnées en blocs de colonnes
- **Circulation de A**
- B et C statiques



Topologie

Partitionnement
et circulation de A

Partitionnement
statique de B

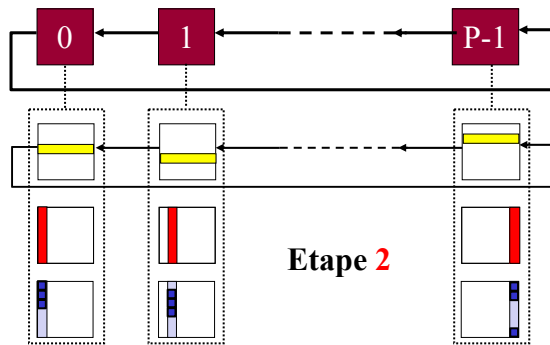
Partitionnement
statique de C

4

Algorithme en anneau sur cluster de PC

Principe algorithmique sur cluster ($C = A \times B$)

- A partitionnée en blocs de lignes
- B et C partitionnées en blocs de colonnes
- **Circulation de A**
- B et C statiques



Topologie

Partitionnement
et circulation de A

Partitionnement
statique de B

Partitionnement
statique de C

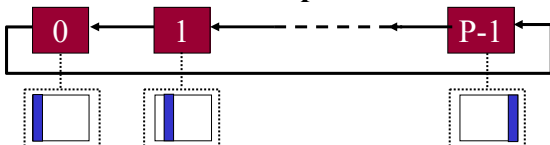
5

Algorithme en anneau sur cluster de PC

Principe algorithmique sur cluster ($C = A \times B$)

- A partitionnée en blocs de lignes
- B et C partitionnées en blocs de colonnes
- **Circulation de A**
- B et C statiques

Résultats à la fin des P étapes :



Topologie

Partitionnement
statique de C

Bilan :

- Chaque PC a calculé un bloc de colonnes de C
 - Les P PC ont travaillé en parallèle
- **Calcul de tous les blocs de colonnes en parallèle**, en P étapes

6

POLYTECH PARIS-SACLAY

Produit de matrices sur cluster de GPU

Algorithme sur cluster de GPU – v1

Algorithme sur cluster de GPU v1

A chaque étape : chaque nœud :

- transfère sa tranche de A sur son GPU (en synchrone)
- lance un kernel GPU (en asynchrone)
- exécute ses communication MPI (en synchrone)
- permute les buffers *A slice 0* et *A slice 1* sur le CPU

Recouvrement « simple/natif »

Transfert sync. CPU→GPU → GPU Kernel async. → MPI comm sync. →

Pourrait se faire avec un seul *A slice* sur CPU

Kernel sérialisé avec le transfert, car même *stream*
 → le nouveau transfert ne peut pas écraser les données courantes du kernel

7

POLYTECH PARIS-SACLAY

Produit de matrices sur cluster de GPU

Algorithme sur cluster de GPU – v1

Algorithme sur cluster de GPU v1

Recouvrement « simple/natif »

Transfert sync. CPU→GPU → GPU Kernel async. → MPI comm sync. →

```

MPI_Status status;

// Transfer of the local strip of B and the node id to the GPU
gpuSetDataOnGPU();
// Computation and circulation loop
for (int step = 0; step < NbPE; step++) {
  int idx = step%2;
  // Transfer of the current local strip of A from the CPU to the GPU
  gpuSetAOnGPU(idx);
  // Computation
  gpuKernelLocalProduct(step, GPUKernelId); // Async call of the GPU kernel
  // Input data circulation
  if (NbPE > 1) {
    MPI_Sendrecv(&A[idx][0][0], LOCAL_SIZE*SIZE, MPI_DOUBLE, (Me+1)%NbPE, step, &A[
    [1-idx][0][0], LOCAL_SIZE*SIZE, MPI_DOUBLE, (Me-1+NbPE)%NbPE, step, &
    MPI_COMM_WORLD, &status);
  }
}
// Get back results from the GPU to the CPU
gpuGetResultOnCPU();
  
```

Code :
MPI +
CUDA

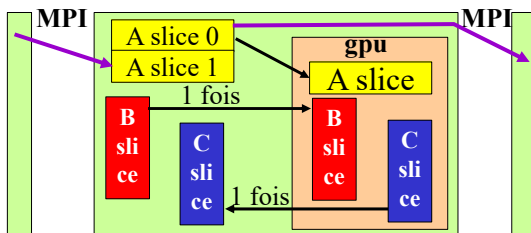
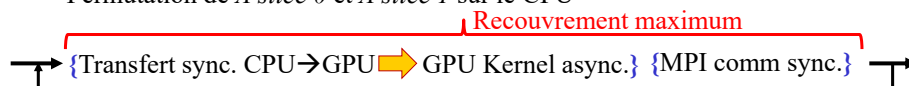
8

Algorithme sur cluster de GPU – v2

Algorithme sur cluster de GPU v2

A chaque étape : chaque nœud exécute 2 threads (OpenMP)

- Thread 0 : - transfère la tranche de A sur le GPU (en synchrone)
- lance un kernel GPU (en asynchrone)
- Thread 1 : - exécute les communication MPI (en synchrone)
- Barrière de synchronisation des 2 threads OpenMP
- Permutation de *A slice 0* et *A slice 1* sur le CPU



Permet de recouvrir les comm MPI avec le kernel GPU et avec les transferts CPU-GPU

Les 2 buffers *A slice* sont nécessaires sur CPU

9

Principe algorithmique V2 :

Recouvrement des comm MPI avec {kernel + transferts}

```
#pragma omp parallel
{
    int thId = omp_get_thread_num();

    for (int step = 0; step < NbPE; step++) {
        int idx = step % 2;

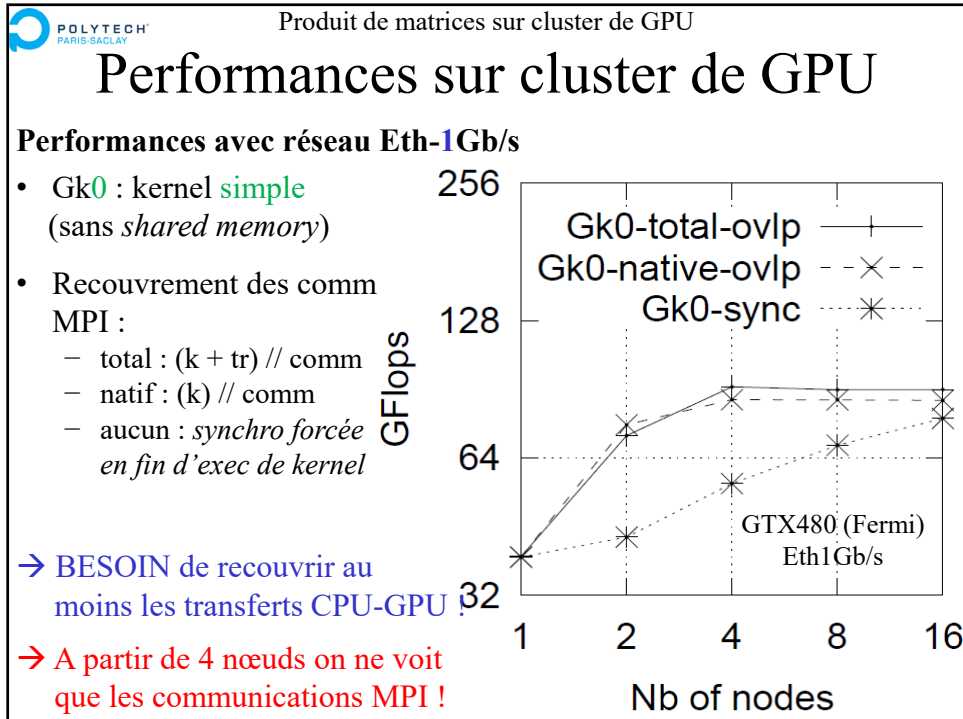
        switch (thId) {
            // Computation thread
            case 0 :
                // Initialize GPU usage at step 0
                if (step == 0) {
                    cudaSetDevice(0); // Indicates that thread 0 uses the GPU 0 (optional)
                    gpuSetDataonGPU(); // Data transfer to the GPU
                }
                // Transfer of the current local version of A to the GPU
                gpuSetAonGPU(idx);
                // Local computation
                gpuKernelLocalProduct(step, GPUkernelId);
                // Finalize GPU usage from thread 0 at last step
                if (step == NbPE-1) {
                    gpuGetResultonCPU(); // Get back the results from the CPU
                }
                break;

            // Communication thread
            case 1 :
                if (NbPE > 1) {
                    MPI_Sendrecv(&A[idx][0][0], LOCAL_SIZE*SIZE, MPI_DOUBLE, (Me+1)%NbPE, step-1,
                                &A[1-idx][0][0], LOCAL_SIZE*SIZE, MPI_DOUBLE, (Me-1)%NbPE, step,
                                MPI_COMM_WORLD, &status);
                }
                break;
        }

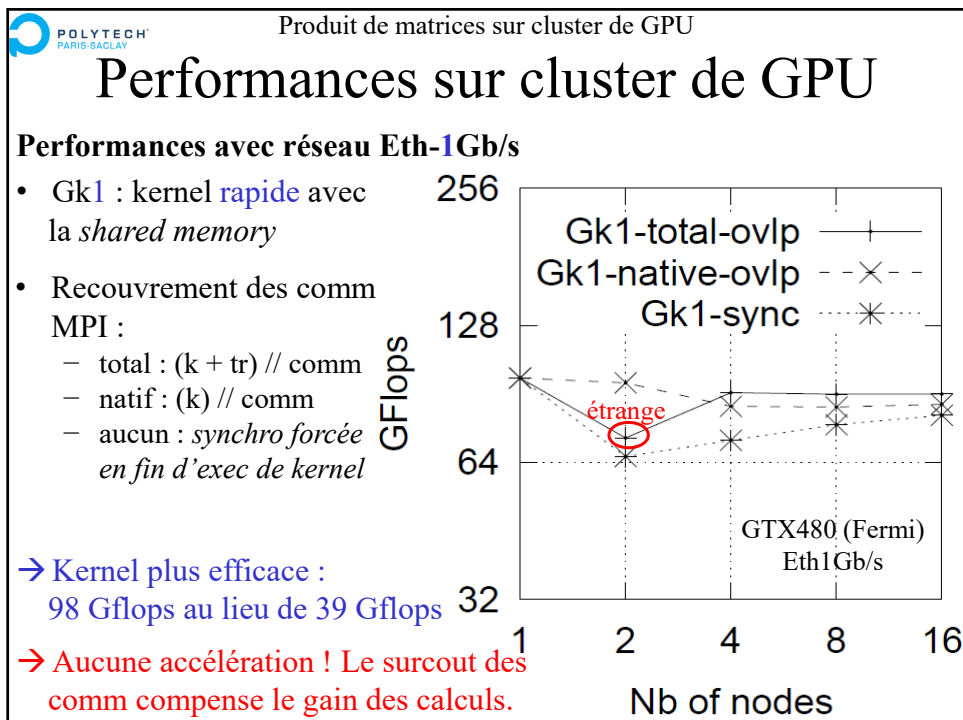
        // Synchronization barrier: wait for termination of both computation and communication
        #pragma omp barrier
    }
} // end of parallel region: end of the threads
```

Code :
MPI +
OpenMP +
CUDA

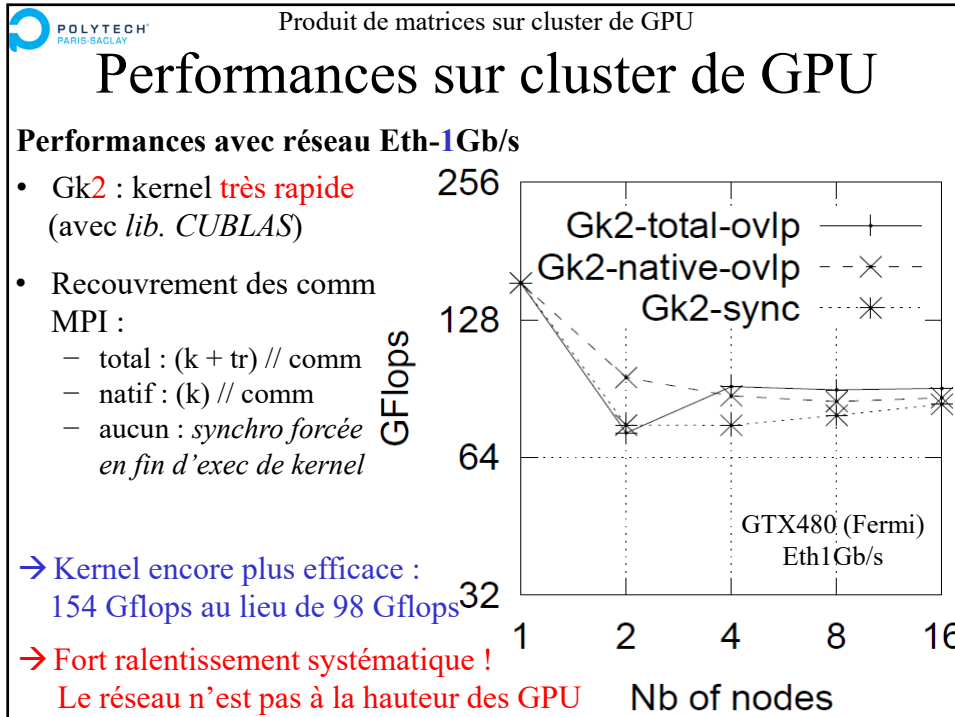
10



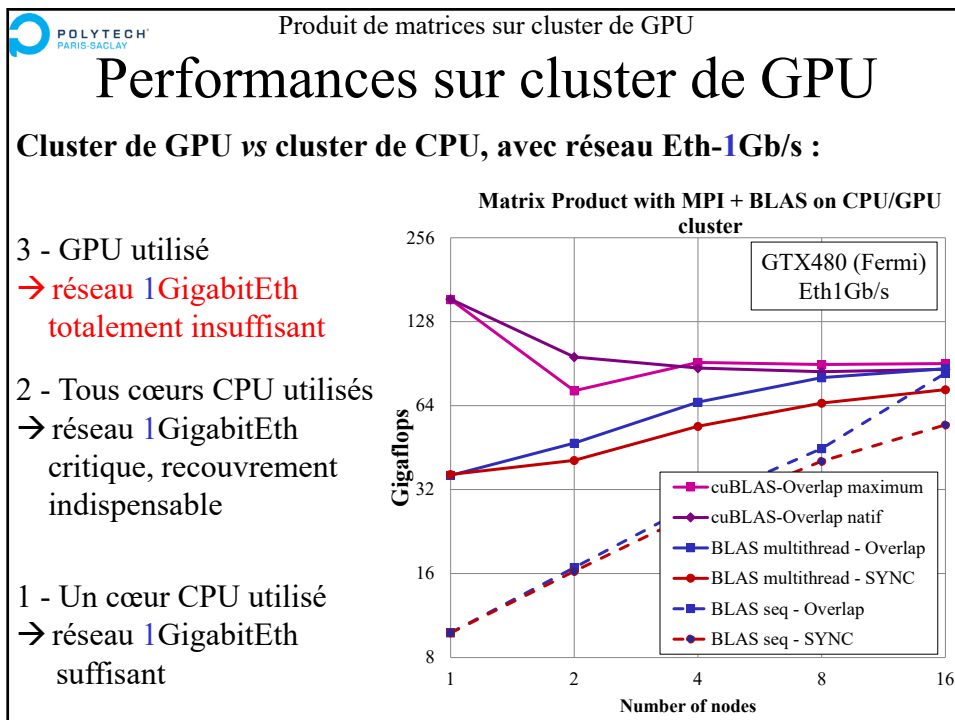
11



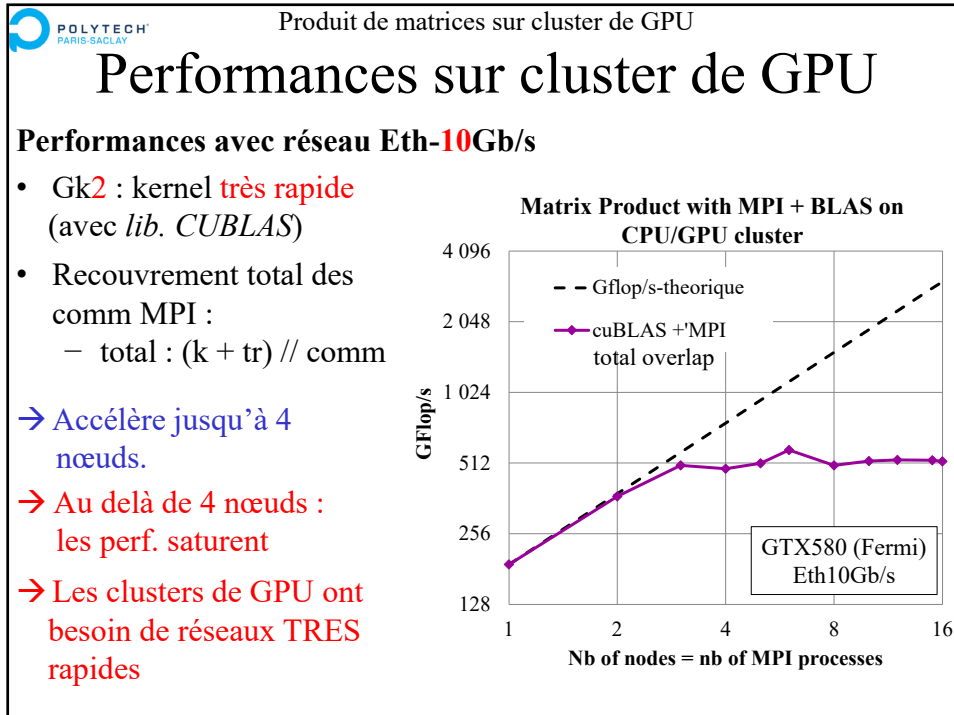
12



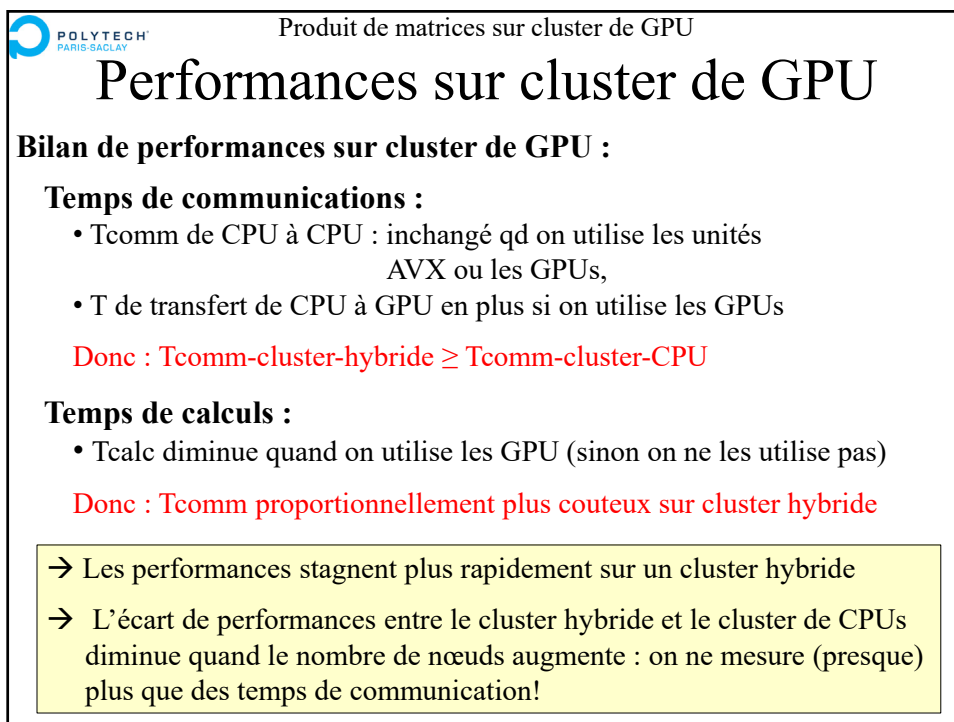
13



14



15



16

Feedback on hybrid programming (CUDA + OpenMP/MPI)

- 1 – Produit de matrices sur cluster de GPU
- 2 – **Parallélisation simultanée sur CPUs et GPUs**
 - Multithreading CPU et CUDA
 - Equilibrage de charge CPU/GPU

17

Multithreading CPU et CUDA

Cas général :

- un nœud de calcul avec N_c cœurs CPU et N_g GPUs.

Objectifs :

- utiliser toute la puissance du nœud de calcul
- utiliser tous les GPUs et tous les cœurs CPUs

1^{ère} démarche possible (fréquente) :

- **implanter un pgm CPU multithreads, avec N_c threads CPU**
- **dédier N_g ($< N_c$) threads CPU pour piloter un GPU chacun**

→ Un thread CPU peut fixer le GPU sur lequel il veut agir :
`cudaSetDevice (#gpuIdx) ;`

→ **La synchronisation du pgm reste celle des threads CPU**

18

Multithreading CPU et CUDA

Cas général :

- un nœud de calcul avec N_c cœurs CPU et N_g GPUs.

Objectifs :

- utiliser toute la puissance du nœud de calcul
- utiliser tous les GPUs et tous les cœurs CPUs

2^{ème} démarche possible :

- chaque GPU est exploité par plusieurs threads CPU, pour l'utiliser au mieux de ses capacités,
 - avec une synchronisation reposant sur le *scheduler* du GPU... ou avec une synchronisation sur « la ressource GPU » faite dans les threads CPU (sorte de *mutex*/file d'attente du GPU).
- La synchro du pgm se fait entre les threads CPU, et entre chaque GPU et ses threads CPU clients.

19

Equilibrage de charge CPU/GPU

Stratégie d'équilibrage de charge **statique** :

- Mesures de performances séparées sur GPU et sur CPU
 - *off-line* : avant lancement de l'applicationOU
 - lancement de micro-benchmarks durant la phase d'initialisation (bon résultats dans une étude avec l'ONERA)
 - Calcul de la répartition optimale théorique entre CPU et GPU
 - Répartition des données et exécution des calculs
- Développement (assez) simple
- Performances souvent bonnes mais parfois sous-optimales (la répartition reste sensible et fonction de la taille des calculs)

20

Equilibrage de charge CPU/GPU

Stratégie d'équilibrage de charge **dynamique** :

- Définition d'un mécanisme de demande de tâche par les threads CPUs
 - Traitement d'une tâche récupérée par un thread CPU sur un (ou plusieurs) cœur CPU, ou sur son GPU associé.
 - Quand un thread CPU à fini sa tâche, il en redemande une autre...
- Développement plus complexe !
- Performances meilleures ... si le mécanisme de gestion des tâches n'est pas trop couteux !

21

Equilibrage de charge CPU/GPU

Stratégie d'équilibrage de charge **dynamique** :

- Il est possible de s'appuyer sur un mécanisme existant de répartition dynamique de tâche entre threads CPU
 - Ex : gestion dynamique de tâches d'OpenMP ou de certains *thread-pools*
- Le développement devient alors beaucoup plus simple

Bon retour d'expérience avec cette démarche sur des problèmes de géophysique (recherche pétrolière)

22

Feedback on hybrid programming (CUDA + OpenMP/MPI)

END