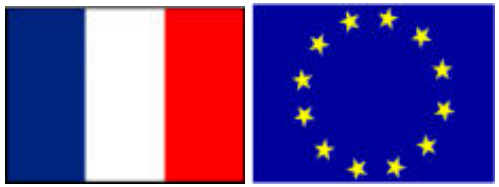


GP-GPU

Feedback on hybrid programming (CUDA + OpenMP/MPI)

Stéphane Vialle

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

POLYTECH
PARIS-SACLAY

Feedback on hybrid programming (CUDA + OpenMP/MPI)

1 – Produit de matrices sur cluster de GPU

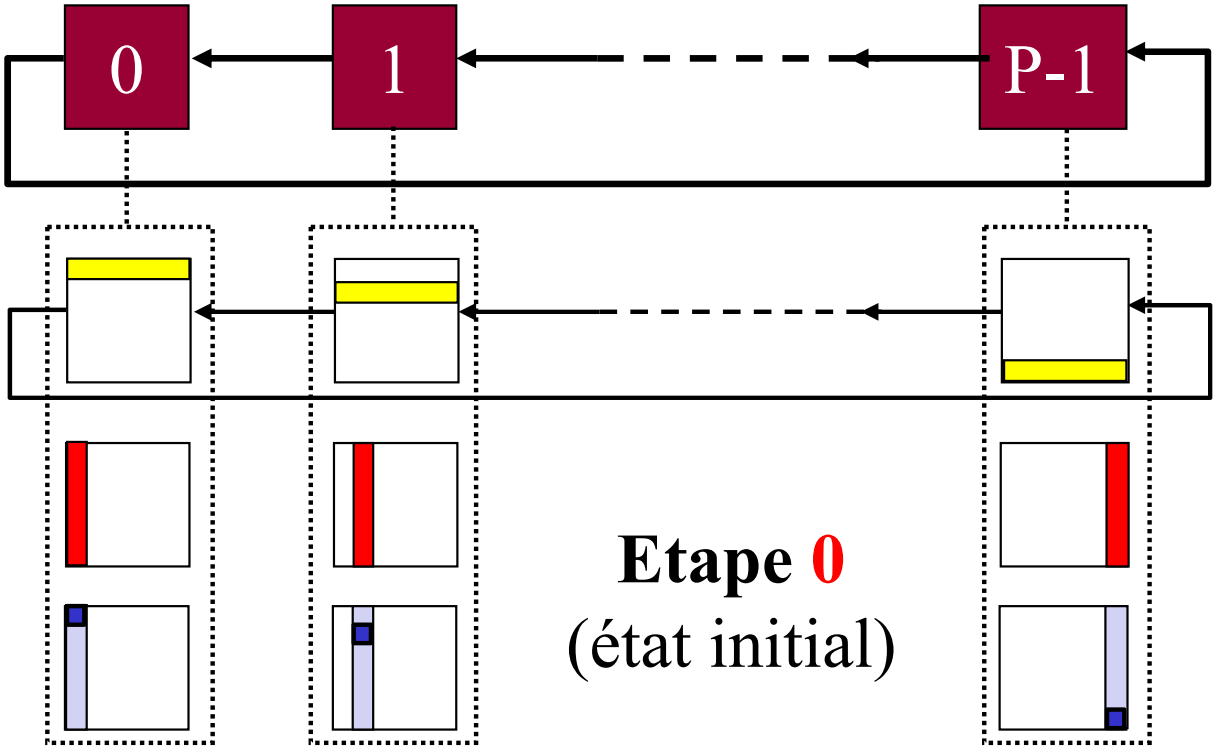
- Algorithme en anneau sur cluster de PC
- Algorithme sur cluster de GPU – v1
- Algorithme sur cluster de GPU – v2
- Performance sur cluster de GPU

2 – Parallélisation simultanée sur CPUs et GPUs

Algorithme en anneau sur cluster de PC

Principe algorithmique sur cluster (C = A×B)

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



Etape 0
(état initial)

Topologie

Partitionnement et circulation de A

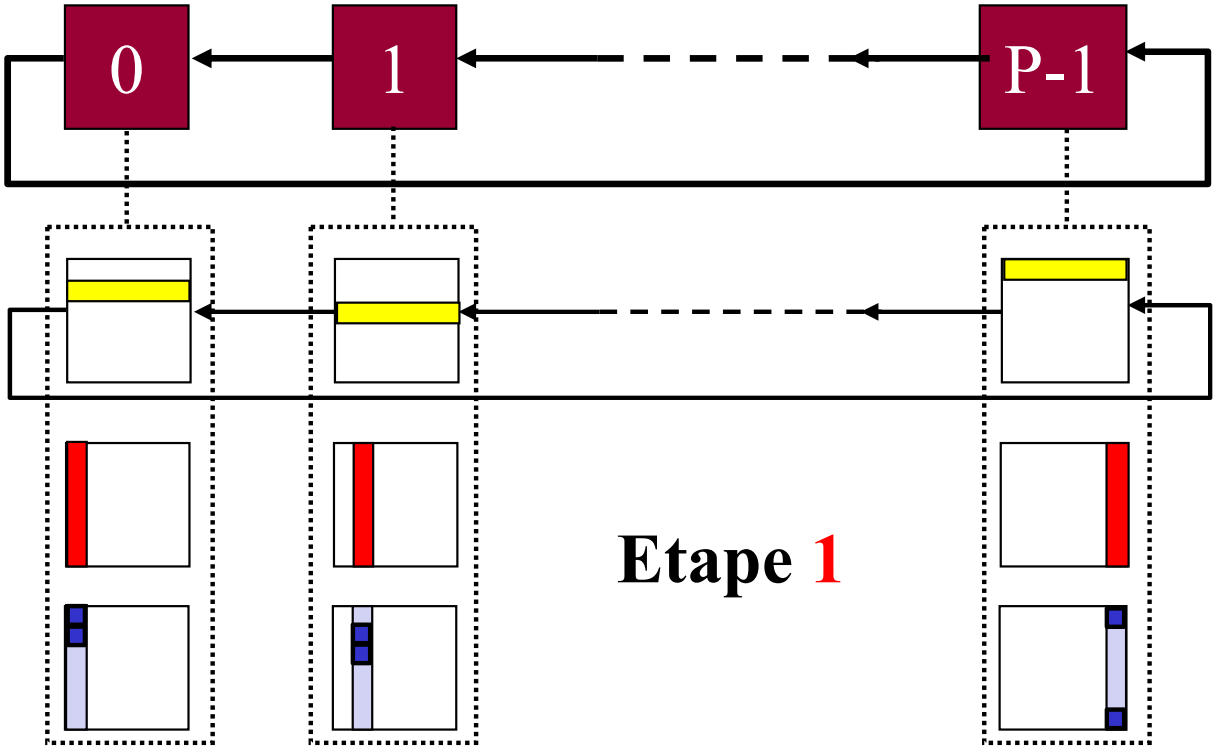
Partitionnement statique de B

Partitionnement statique de C

Algorithme en anneau sur cluster de PC

Principe algorithmique sur cluster (C = A×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

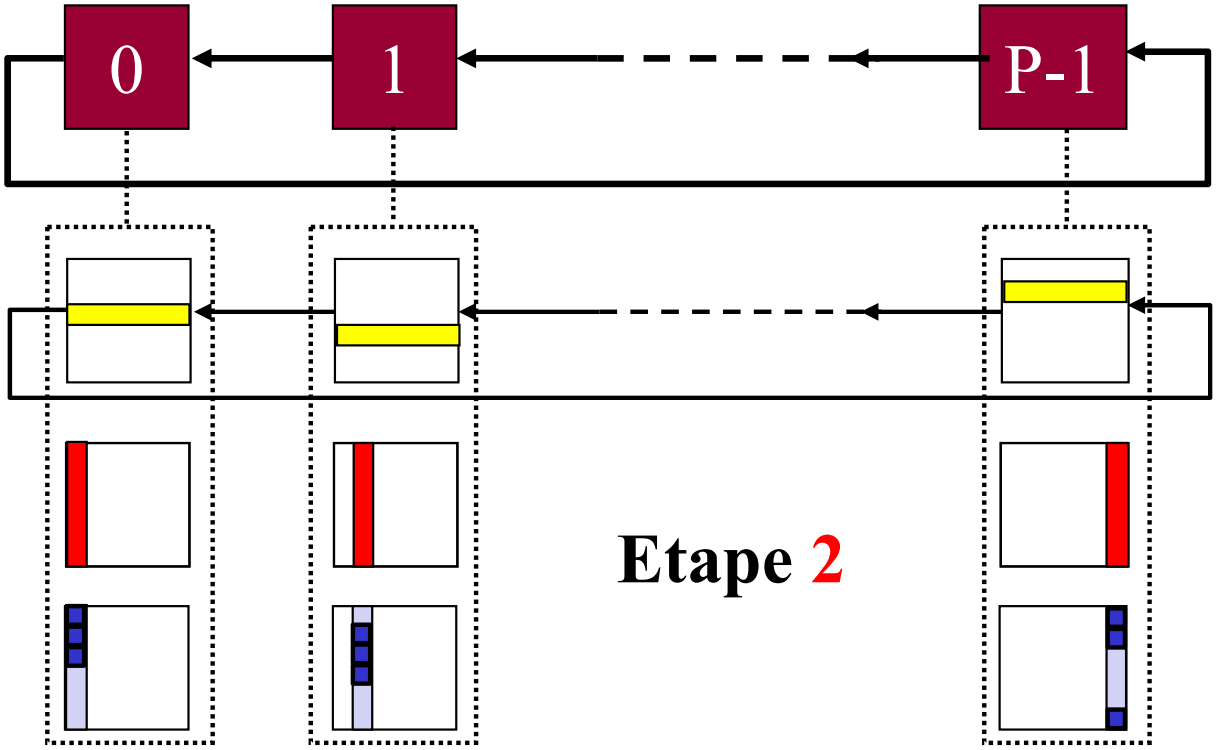
Partitionnement statique de B

Partitionnement statique de C

Algorithme en anneau sur cluster de PC

Principe algorithmique sur cluster (C = A×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

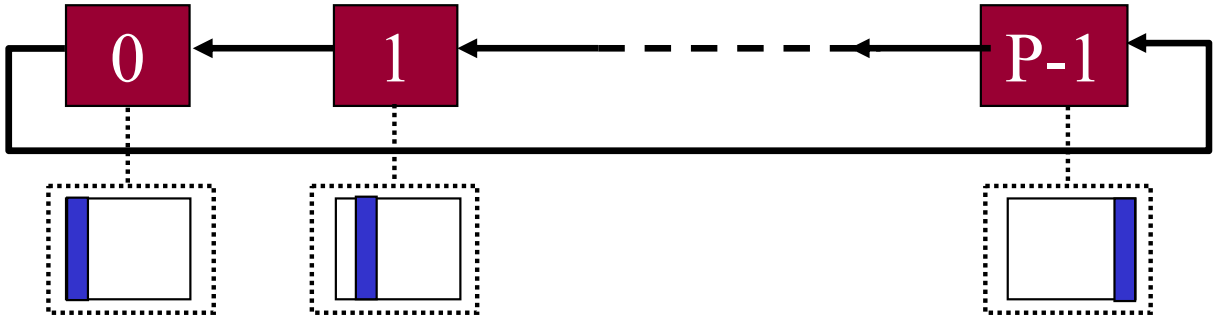
Partitionnement statique de C

Algorithme en anneau sur cluster de PC

Principe algorithmique sur cluster (C = A×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

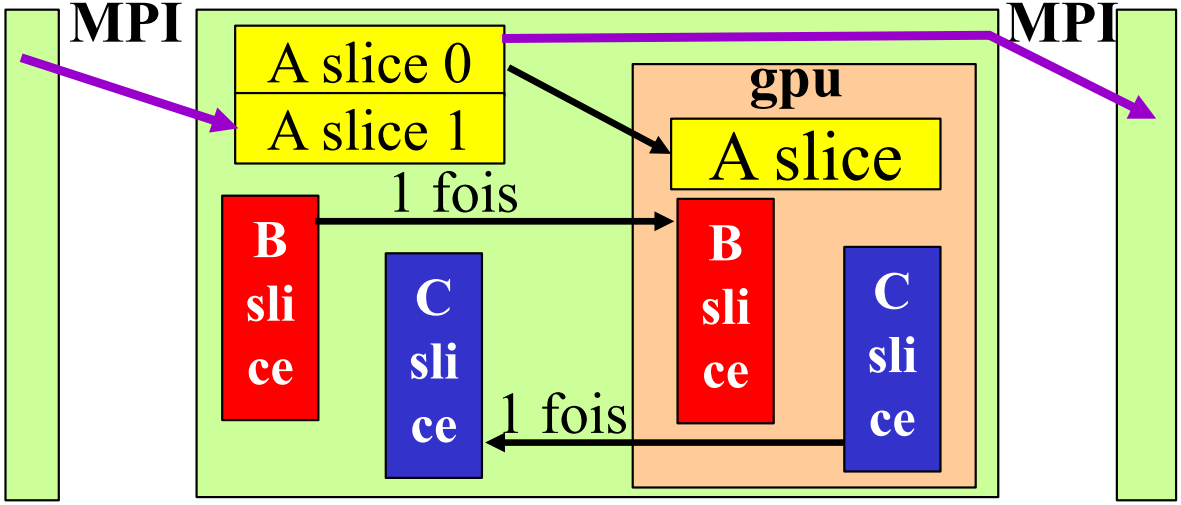
→ **Calcul de tous les blocs de colonnes en parallèle**, en *P* étapes

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

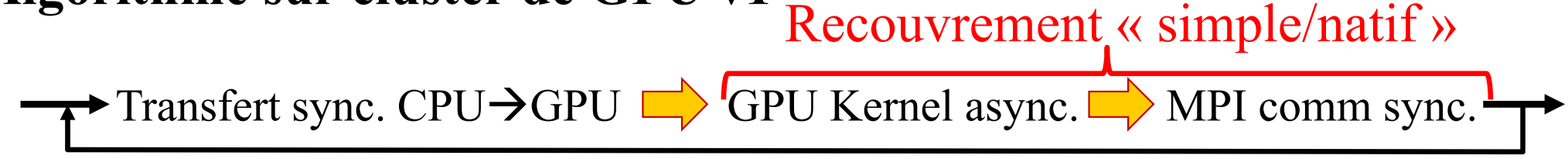


Kernel sérialisé avec le transfert, car même *stream*
→ *le nouveau transfert ne peut pas écraser les données courantes du kernel*

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

Algorithme sur cluster de GPU – v1

Algorithme sur cluster de GPU v1




```
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();
```

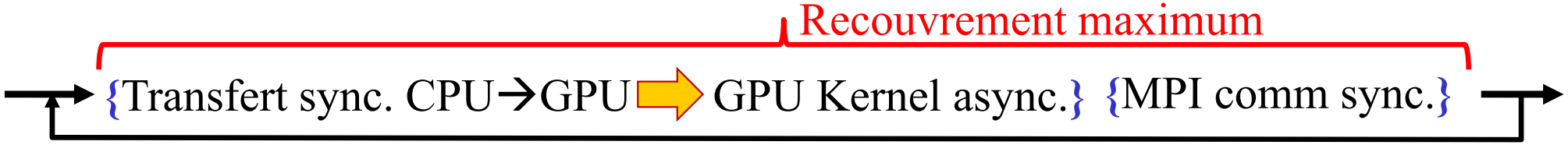
Code : MPI + CUDA

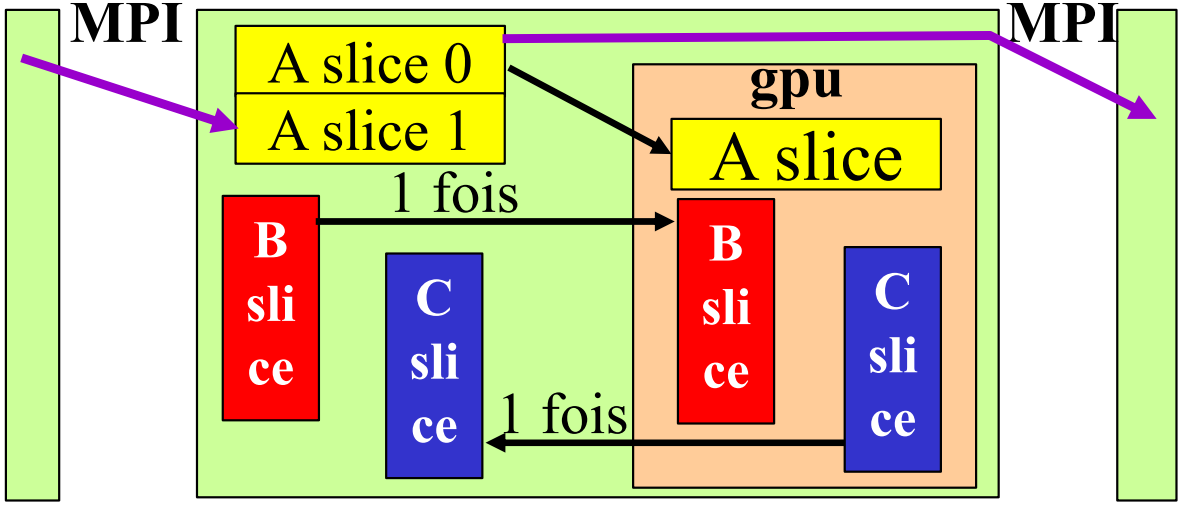
Algorithme sur cluster de GPU – v2

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





Les 2 buffers *A slice* sont nécessaires sur CPU

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

POLYTECH PARIS-SACLAY

Principe algorithmique V2 :

Recouvrement des comm MPI avec {kernel + transferts}

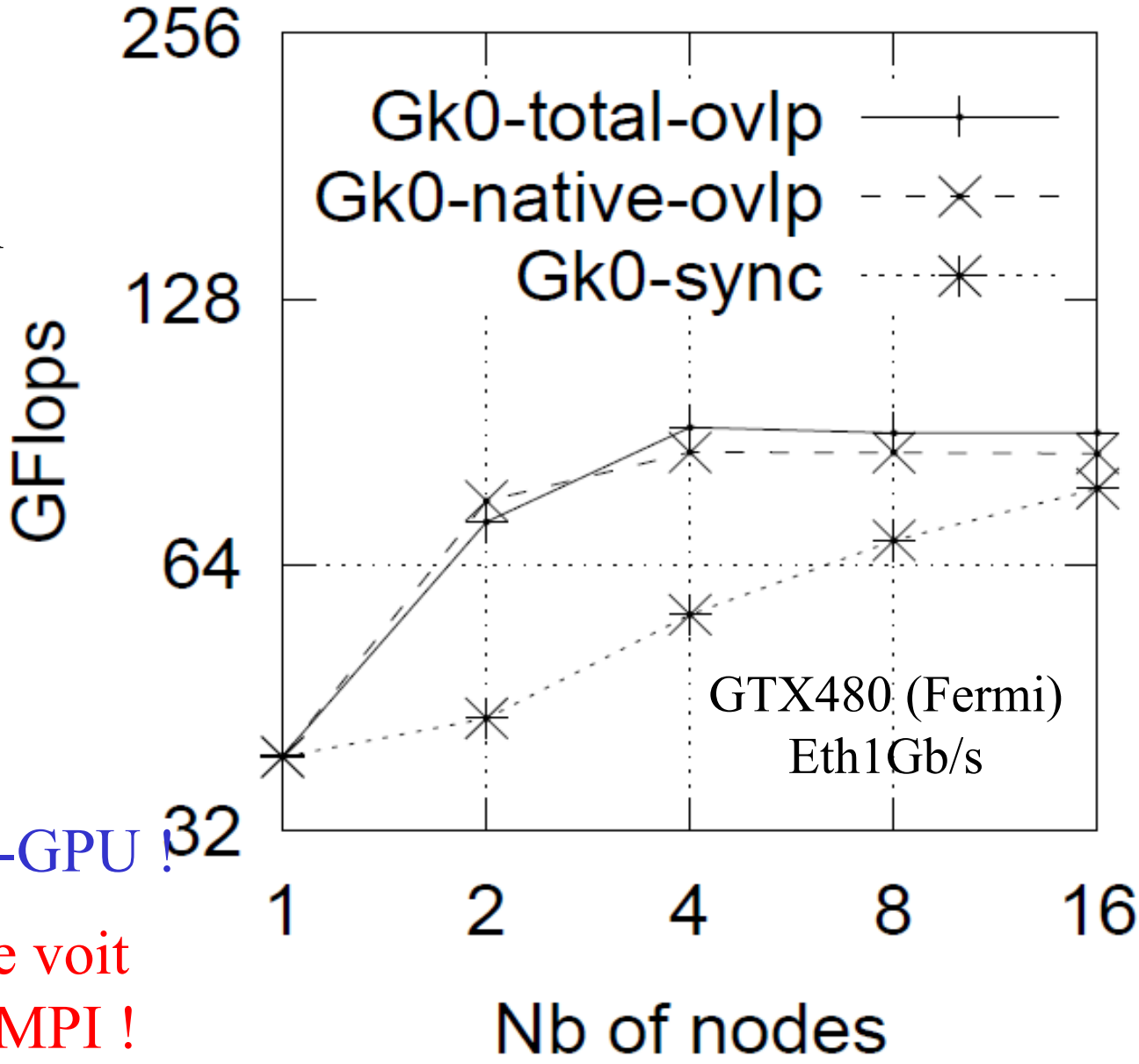
Code :
MPI +
OpenMP +
CUDA

```
#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↩
            , &A[1-idx][0][0], LOCAL_SIZE*SIZE, MPI_DOUBLE, (Me-1+NbPE)%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
```

Performances sur cluster de GPU

Performances avec réseau Eth-1Gb/s

- Gk0 : kernel simple (sans *shared memory*)
- Recouvrement des comm MPI :
  - total : (k + tr) // comm
  - natif : (k) // comm
  - aucun : *synchro forcée en fin d'exec de kernel*



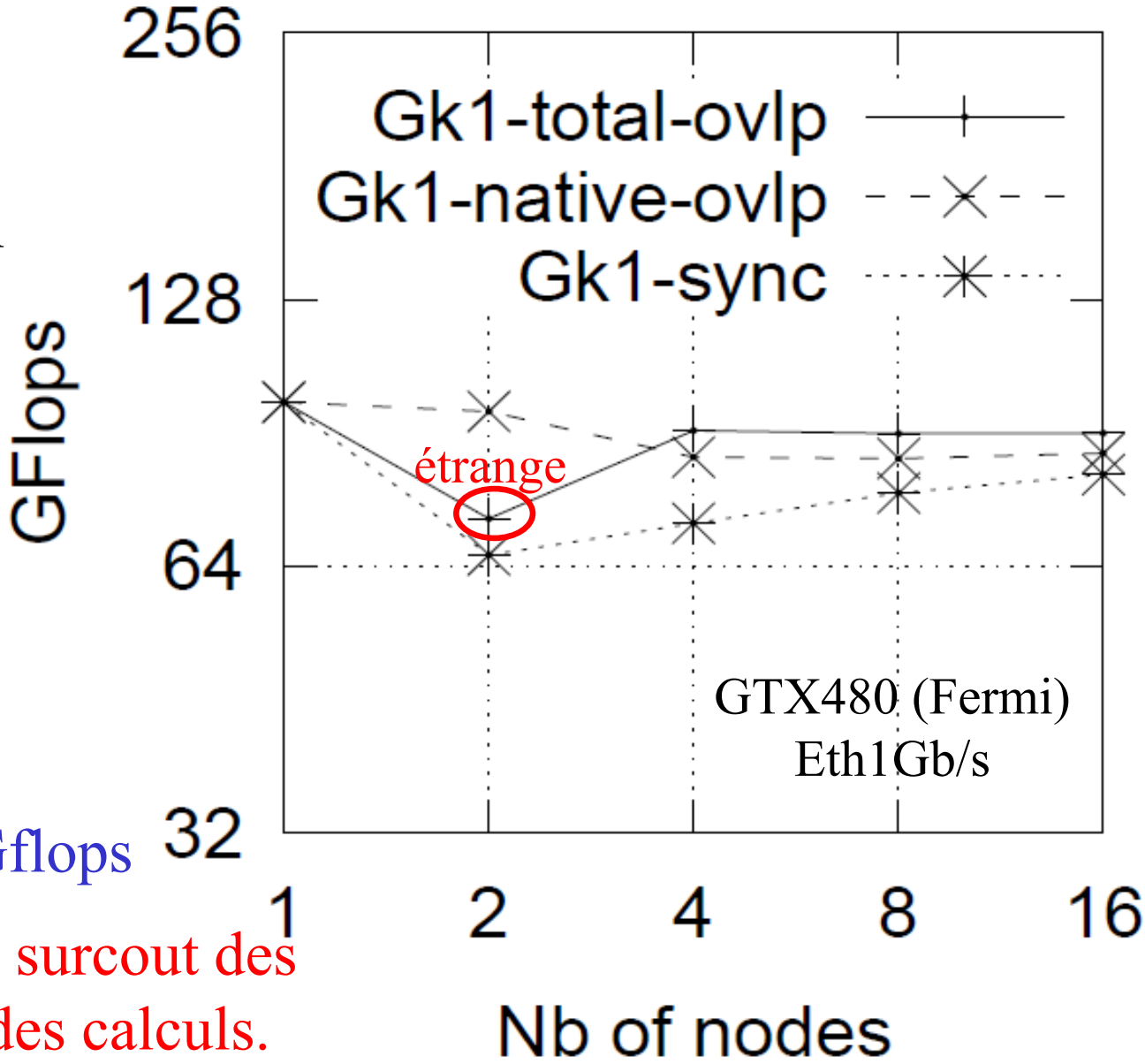
→ BESOIN de recouvrir au moins les transferts CPU-GPU !

→ A partir de 4 nœuds on ne voit que les communications MPI !

Performances sur cluster de GPU

Performances avec réseau Eth-1Gb/s

- Gk1 : kernel rapide avec la *shared memory*
- Recouvrement des comm MPI :
  - total : (k + tr) // comm
  - natif : (k) // comm
  - aucun : *synchro forcée en fin d'exec de kernel*



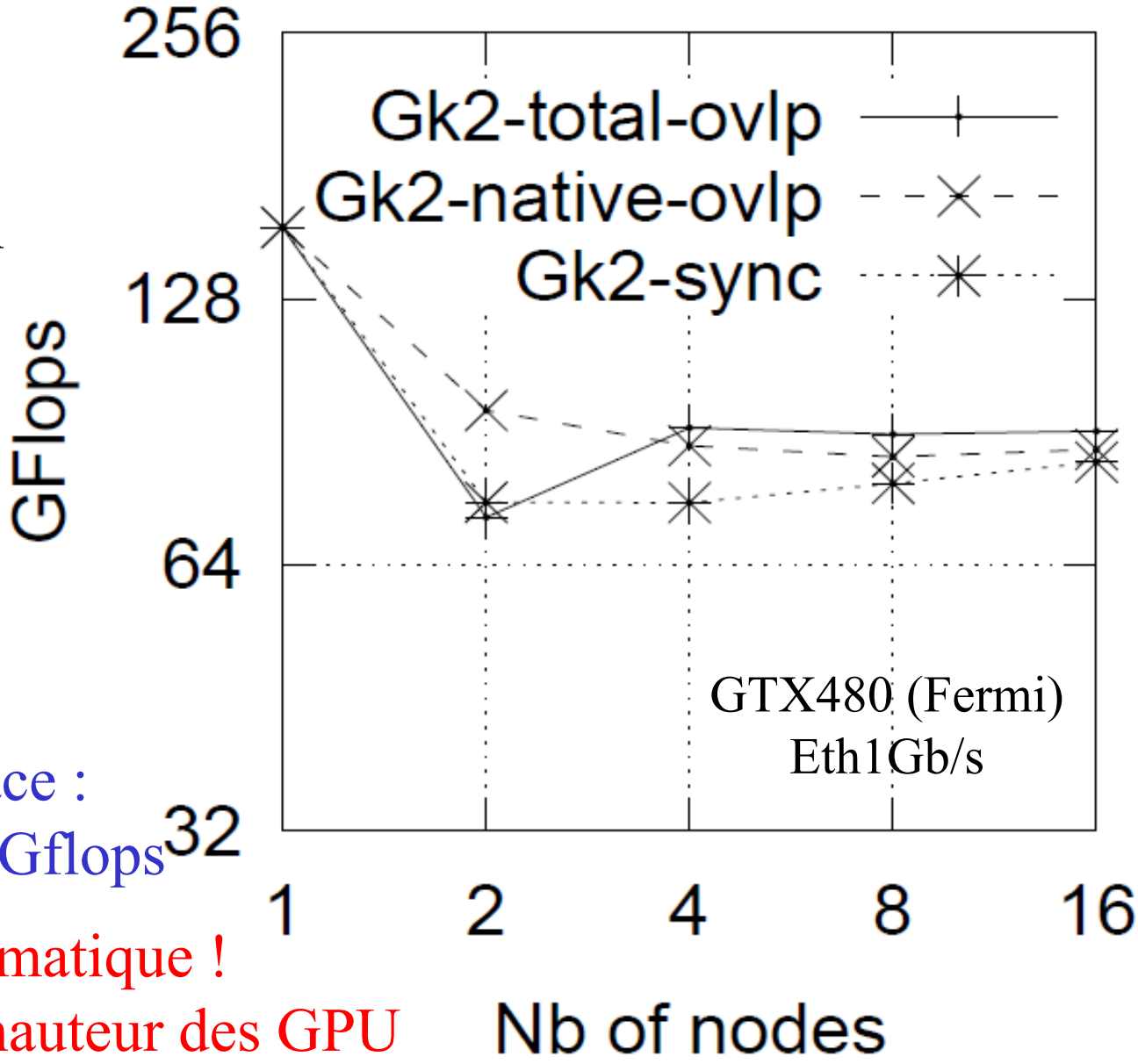
→ Kernel plus efficace :
98 Gflops au lieu de 39 Gflops

→ Aucune accélération ! Le surcout des comm compense le gain des calculs.

Performances sur cluster de GPU

Performances avec réseau Eth-1Gb/s

- Gk2 : kernel très rapide (avec *lib. CUBLAS*)
- Recouvrement des comm MPI :
  - total : (k + tr) // comm
  - natif : (k) // comm
  - aucun : *synchro forcée en fin d'exec de kernel*



→ Kernel encore plus efficace : 154 Gflops au lieu de 98 Gflops

→ Fort ralentissement systématique ! Le réseau n'est pas à la hauteur des GPU

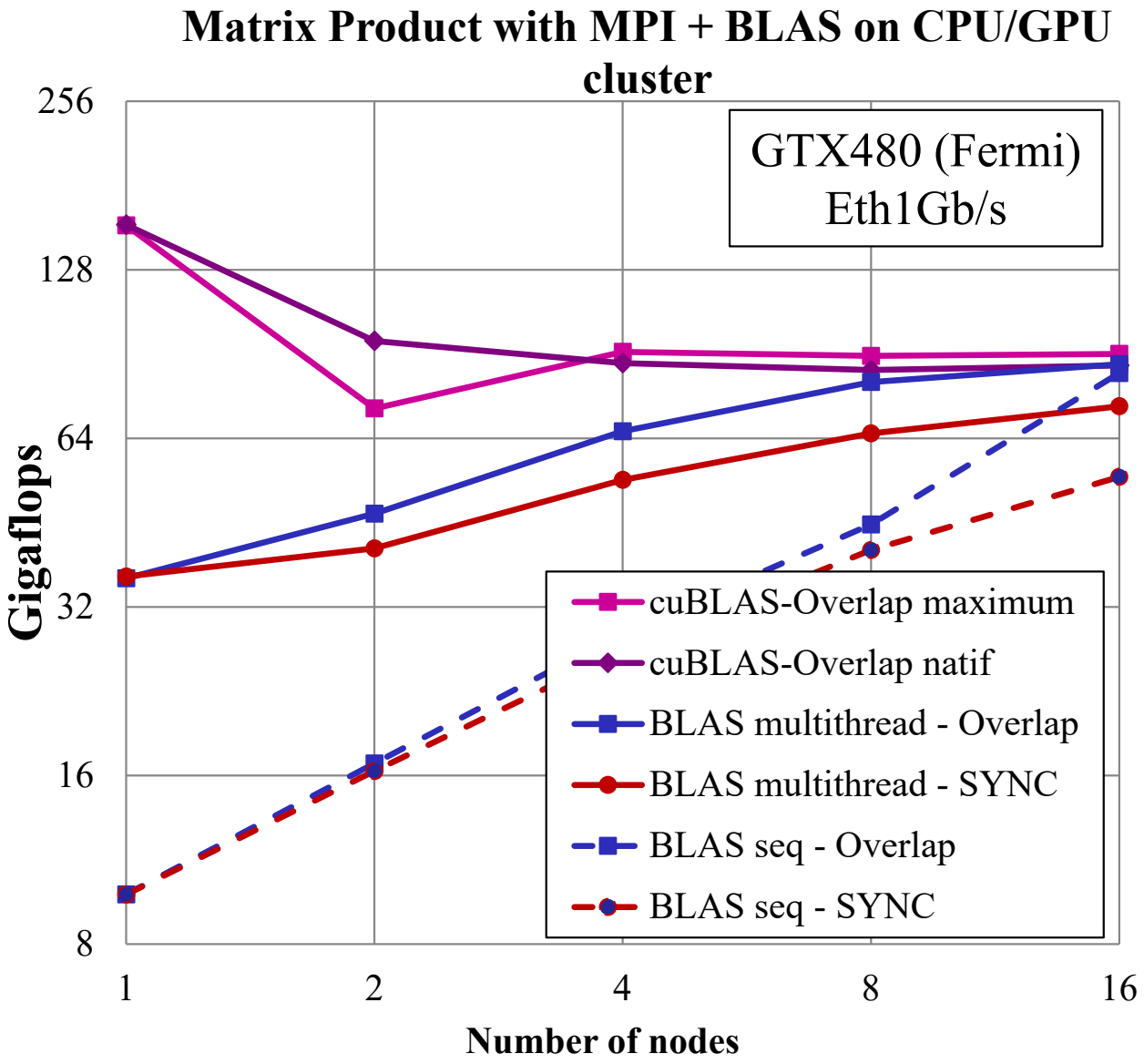
Performances sur cluster de GPU

Cluster de GPU *vs* cluster de CPU, avec réseau Eth-1Gb/s :

3 - GPU utilisé
→ réseau 1GigabitEth totalement insuffisant

2 - Tous cœurs CPU utilisés
→ réseau 1GigabitEth critique, recouvrement indispensable

1 - Un cœur CPU utilisé
→ réseau 1GigabitEth suffisant



Performances sur cluster de GPU

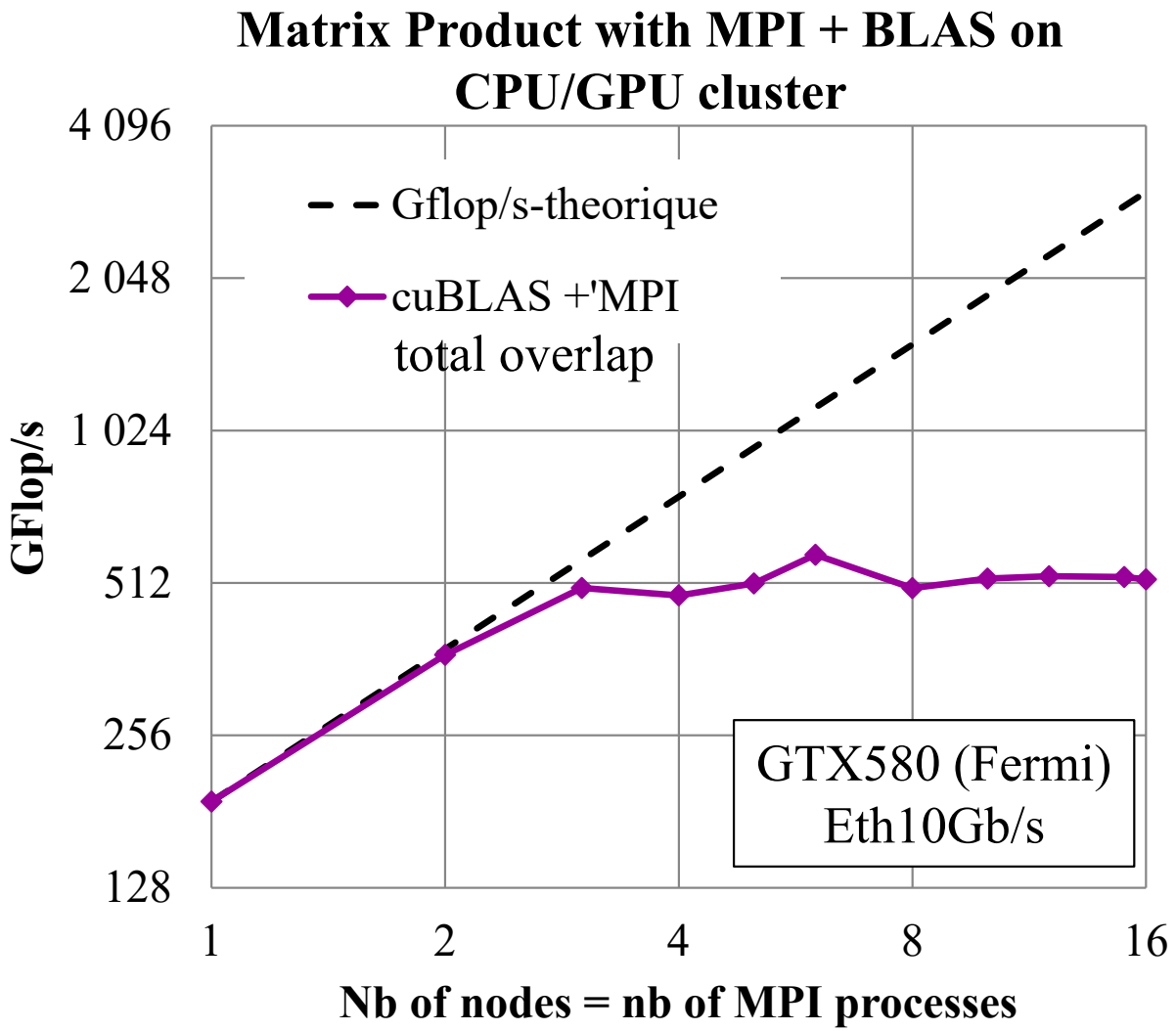
Performances avec réseau Eth-10Gb/s

- Gk2 : kernel très rapide (avec *lib. CUBLAS*)
- Recouvrement total des comm MPI :
  - total : (k + tr) // comm

→ Accélère jusqu'à 4 nœuds.

→ Au delà de 4 nœuds : les perf. saturent

→ Les clusters de GPU ont besoin de réseaux TRES rapides



Performances sur cluster de GPU

Bilan de performances sur cluster de GPU :

Temps de communications :

- Tcomm de CPU à CPU : inchangé qd on utilise les unités AVX ou les GPUs,
- T de transfert de CPU à GPU en plus si on utilise les GPUs

Donc : Tcomm-cluster-hybride ≥ Tcomm-cluster-CPU

Temps de calculs :

- Tcalc diminue quand on utilise les GPU (sinon on ne les utilise pas)

Donc : Tcomm proportionnellement plus couteux sur cluster hybride

→ Les performances stagnent plus rapidement sur un cluster hybride

→ L'écart de performances entre le cluster hybride et le cluster de CPUs diminue quand le nombre de nœuds augmente : on ne mesure (presque) plus que des temps de communication!

POLYTECH
PARIS-SACLAY

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

Multithreading CPU et CUDA

Cas général :

- un nœud de calcul avec *Nc* cœurs CPU et *Ng* GPUs.

Objectifs :

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

1ère démarche possible (fréquente) :

- implanter un pgm CPU multithreads, avec *Nc* threads CPU
- dédier *Ng* (< *Nc*) 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**

Multithreading CPU et CUDA

Cas général :

- un nœud de calcul avec *Nc* cœurs CPU et *Ng* GPUs.

Objectifs :

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

2ème 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.

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'application

  OU
  - 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)

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 !

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)

POLYTECH
PARIS-SACLAY

Feedback on hybrid programming (CUDA + OpenMP/MPI)

END