



GP-GPU

CUDA: Fast transfers and overlap

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

CUDA : Transferts rapides et recouvrements

- 1 – Utilisation de pages mémoires CPU verrouillées
 - Principes de la mémoire verrouillée
 - Allocation et conversion de mémoire verrouillée
 - Accélération des transferts CPU/GPU
- 2 – Transferts asynchrones CPU/GPU
- 3 – Utilisation de multiples *Streams*

2

Principes de la mémoire verrouillée



- Mémoire paginée standard (`malloc (...)`)

Ou bien :

- **Mémoire verrouillée (« *pinned* », « *locked* ») :**

- La mémoire paginée peut être envoyée sur disque, alors que la mémoire verrouillée reste en RAM
- Il ne faut pas verrouiller trop de gros volumes de données sinon il n'y a plus de RAM disponible !



- **Les transferts CPU/GPU seront + rapides**
- **Les transferts CPU/GPU pourront être asynchrones (voir plus loin)**

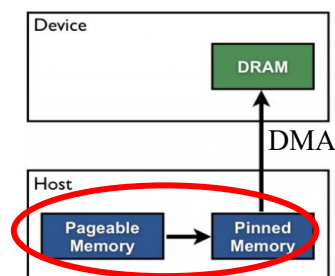


3

Principes de la mémoire verrouillée

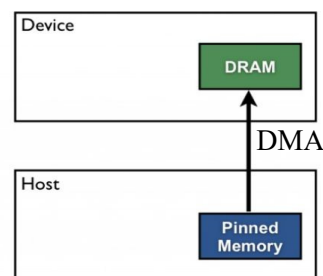
Le transfert CPU/GPU nécessite des pages mémoires verrouillées

Pageable Data Transfer



1. Allocation de mémoire verrouillée
 2. Recopie des données à transférer
 3. Puis transfert des données
- op longue avec synchronisations

Pinned Data Transfer



1. Transfert des données
- allouer dès le début de l'espace mémoire verrouillé !

4

Allocation de mémoire verrouillée

Allocation de mémoire verrouillée

Allocation de mémoire verrouillée sur le CPU pour une application :

```
cudaHostAlloc(AdrPtHost, size, cudaHostAllocDefault)
```

Allocation de mémoire verrouillée accessible à plusieurs applications
(plusieurs contextes CUDA) :

```
cudaHostAlloc(AdrPtHost, size, cudaHostAllocPortable)
```

Libération de la mémoire verrouillée allouée dans la RAM du CPU

```
cudaFreeHost(PtHost)
```

*Rmq : on peut allouer de la mémoire verrouillée avec les bibliothèques
systèmes CPU...mais avec CUDA c'est très simple à faire !*

5

Conversion de mémoire verrouillée

Conversion dynamique de mémoire paginée en mémoire verrouillée

- On transforme une zone mémoire standard en mémoire verrouillée :

```
cudaHostRegister(PtHost, size, cudaHostRegisterDefault)
```

ou

```
cudaHostRegister(PtHost, size, cudaHostRegisterPortable)
```

- On retransforme une zone mémoire verrouillée en mémoire paginée dès qu'on n'a plus besoin de la transférer vers/depuis le GPU:

```
cudaHostUnregister(PtHost)
```

évite de garder trop d'espace mémoire verrouillé et de manquer de RAM!

→ *La conversion dynamique facilite la transformation de legacy codes*

Inutile de modifier la totalité d'un ancien code CPU complexe !

6

Accélération des transferts CPU/GPU

Possibilité d'accélération sans autres changement dans le code

```
cudaMemcpy (PtDst, PtSrc, size, cudaMemcpyHostToDevice)
MyKernel<<< Dg, Db >>> (.....)
cudaMemcpy (PtDst, PtSrc, size, cudaMemcpyDeviceToHost)
```

Les transferts CPU/GPU depuis/vers de la mémoire CPU verrouillée doivent être plus rapides ...

... mais on n'observe pas toujours de différence!

Les principaux intérêts à utiliser de la mémoire verrouillée sont :

- l'exploitation de routines de transferts asynchrones
- le recouvrement entre les calculs CPU, les calculs GPU et les transferts CPU/GPU.



7

CUDA : Transferts rapides et recouvrements

- 1 – Utilisation de pages mémoires CPU verrouillées
- 2 – **Transferts asynchrones CPU/GPU**
 - **Routines CUDA**
 - **Fonctionnement d'un *stream* CUDA**
 - **Recouvrement de calculs CPU et GPU**
- 3 – Utilisation de multiples *Streams*

8

Routines CUDA

La mémoire CPU verrouillée autorise les transferts asynchrones

```
1 cudaMemcpyAsync(PtGPU, PtCPU, size, cudaMemcpy...To...);
2 kernel<<<Dg,Db>>>(...);
3 cudaMemcpyAsync(PtCPU, PtGPU, size, cudaMemcpy...To...);
4 ..... // next instructions of the CPU
```

- Les trois opérations 1, 2, et 3 sont **non bloquantes** pour le CPU qui fait ces appels : elles sont réalisées en asynchrone vis-à-vis du CPU
- Le CPU se retrouve instantanément à l'instruction 4 et peut entamer d'autres calculs pendant que le GPU travaille (voir plus loin)
- Mais les instructions 1, 2 et 3 sont bien exécutées dans l'ordre par le GPU, sur le *stream par défaut* (ou *stream 0*)



9

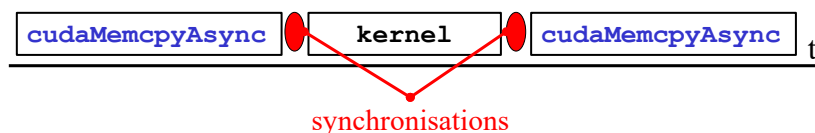
Fonctionnement d'un *stream* CUDA

Un *stream* fonctionne comme une file d'attente FIFO

On peut préciser explicitement le *stream* à utiliser (0 = *stream* par défaut) :

```
1 cudaMemcpyAsync(PtGPU, PtCPU, size, cudaMemcpy...To..., 0);
2 kernel<<<Dg,Db,0,0>>>(...);
3 cudaMemcpyAsync(PtCPU, PtGPU, size, cudaMemcpy...To..., 0);
```

Le *stream* séquentialise (dans l'ordre) les opérations qui lui sont envoyées :



→ Il y a sérialisation et synchronisation implicites des opérations lancées sur un même *stream*

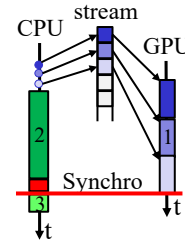
10

Recouvrement de calculs CPU et GPU

Mémoire verrouillée + Asynchronisme + Synchronisation :

```

1  cudaHostRegister (PtInCPU, ...);
2  cudaHostRegister (PtOutCPU, ...);
-----
3  cudaMemcpyAsync (PtInGPU, PtInCPU, size,
                    cudaMemcpyHostToDevice);
4  k1_GPU<<< Dg, Db >>> (...);
5  cudaMemcpyAsync (PtOutCPU, PtOutGPU, size,
                    cudaMemcpyDeviceToHost);
-----
6  k2_CPU (...);
7  cudaDeviceSynchronize (); ← Attend la fin de toutes
8  k3_CPU (PtOutCPU);         les ops. sur le GPU
-----
9  cudaFreeHost (PtInCPU);
10 cudaFreeHost (PtOutCPU);
    
```



→ Calculs sur CPU en parallèle des transferts et des calculs sur GPU

→ Besoin d'une **synchro. explicite** avant d'exploiter les résultats du GPU

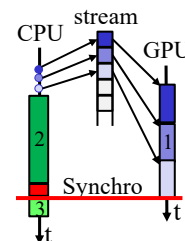
11

Recouvrement de calculs CPU et GPU

Mémoire verrouillée + Asynchronisme + Synchronisation :

```

1  cudaHostRegister (PtInCPU, ...);
2  cudaHostRegister (PtOutCPU, ...);
-----
3  cudaMemcpyAsync (PtInGPU, PtInCPU, size,
                    cudaMemcpyHostToDevice);
4  k1_GPU<<< Dg, Db >>> (...);
5  cudaMemcpyAsync (PtOutCPU, PtOutGPU, size,
                    cudaMemcpyDeviceToHost);
-----
6  k2_CPU (...);
7  cudaStreamSynchronize (0); ← Attend la fin de toutes les
8  k3_CPU (PtOutCPU);         ops. sur le stream par défaut
-----
9  cudaFreeHost (PtInCPU);
10 cudaFreeHost (PtOutCPU);
    
```



→ Calculs sur CPU en parallèle des transferts et des calculs sur GPU

→ Besoin d'une **synchro. explicite** avant d'exploiter les résultats du GPU

12

CUDA :

Transferts rapides et recouvrements

- 1 – Utilisation de pages mémoires CPU verrouillées
- 2 – Transferts asynchrones CPU/GPU
- 3 – Utilisation de multiples *Streams*
 - Création et suppression de *Streams*
 - Recouvrement de transferts et de calculs GPU

13

Création et suppression de *Streams*

Routines CUDA

CUDA permet de créer un grand nombre de *streams* (sans limites)
 Mais un nombre limité de *streams* peuvent fonctionner en parallèle
 (ex : 16 sur les anciennes architectures Fermi....)

```

1  cudaStream_t Stream[Ns];
2  for (int i = 0; i < Ns; i++)
    cudaStreamCreate(&Stream[i]);
3  ..... // Transfers and computations on all streams
4  for (int i = 0; i < Ns; i++)
    cudaStreamDestroy(Stream[i]);
    
```

Chaque *stream* créé gère une file FIFO d'opérations, et sérialise et synchronise les opérations de sa file.

14

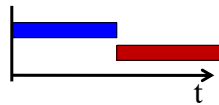
Recouvrement de transferts et de calculs GPU

Basic approach

```
// Synchronous basic data transfer
1 cudaMemcpy(PtGPU, PtCPU, N*sizeof(float),
             cudaMemcpyHostToDevice);

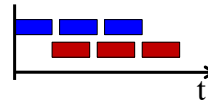
// Run a 1D Grid of 1D blocks of threads
2 kernel<<< {N/BSX,1,1}, {BSX,1,1} >>>(PtGPU);
```

Chronogramme
de l'exécution :



→ Le GPU et le bus PCIe ne travaillent pas en parallèle

On cherche à obtenir un **recouvrement**
des calculs et des transferts :

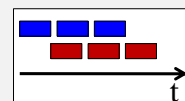


15

Recouvrement de transferts et de calculs GPU

Démarche NVIDIA : Lancer tous les transferts de données sur chaque *stream*
Puis, lancer toutes les petites grilles de blocs de threads

```
..... // Create Nst streams
1 size = N*sizeof(float)/Nst; //Size processed on 1 stream
// Insert a transfer op in each stream FIFO
2 for (i = 0; i < Nst; i++) {
3   offset = i * N/Nst;
4   cudaMemcpyAsync(PtGPU + offset, PtCPU + offset, size,
                   cudaMemcpyHostToDevice, Stream[i]);
}
// Insert a small grid launch op in each stream FIFO
5 for (i = 0; i < Nst; i++) {
6   offset = i * N/Nst;
7   kernel<<< {N/Nst/BSX,1,1}, {BSX,1,1},
              0, Stream[i] >>>(PtGPU + offset);
}
```



16

Recouvrement de transferts et de calculs GPU

Démarche NVIDIA : Puis, lancer tous les transferts de résultats
Puis, resynchroniser avec le CPU (si besoin) – v1

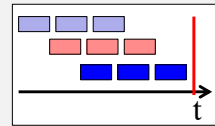
```

// Insert a transfer op in each stream FIFO
8 for (i = 0; i < Nst; i++) {
9   offset = i * N/Nst;
10  cudaMemcpyAsync(PtResCPU+offset, PtResGPU+offset, size,
                  cudaMemcpyDeviceToHost, Stream[i]);
   }

// Synchronize the CPU with end of ops. on each stream
11 for (i = 0; i < Nst; i++) {
12  cudaStreamSynchronize(Stream[i]);
   }

..... // Destroy the Nst streams

```



17

Recouvrement de transferts et de calculs GPU

Démarche NVIDIA : Puis, lancer tous les transferts de résultats
Puis, resynchroniser avec le CPU (si besoin) – v2

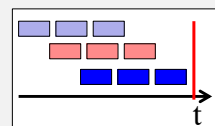
```

// Insert a transfer op in each stream FIFO
8 for (i = 0; i < Nst; i++) {
9   offset = i * N/Nst;
10  cudaMemcpyAsync(PtResCPU+offset, PtResGPU+offset, size,
                  cudaMemcpyDeviceToHost, Stream[i]);
   }

// Synchronize the CPU with end of ops. on the GPU
11 cudaDeviceSynchronize();

..... // Destroy the Nst streams

```



18

CUDA: Fast transfers and overlap

Fin