



GP-GPU

Advanced CUDA programming Part 1

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



Advanced CUDA programming Part 1

1. Principles of the *Shared Memory*

- **Basic concepts**
- Scheme of a basic ShM 2D-kernel
- Scheme of a ShM 2D-kernel with loop

2. Examples of the use of *Shared Memory*

3. Atomic operations

4. Dynamic parallelism

5. Conclusion on CUDA programming

2

Principles of the *shared memory* Basic concepts

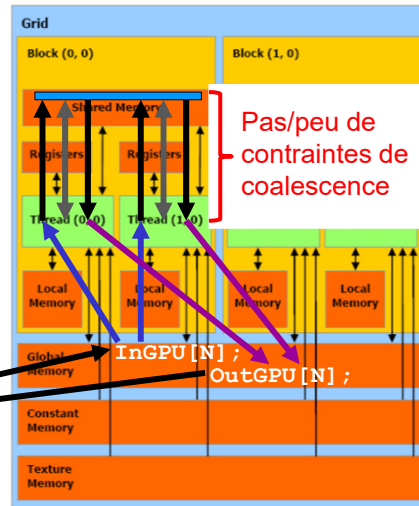
Avantages des kernels utilisant la mémoire globale et la mémoire *shared* :

Motivations/Problèmes :

- besoin que les threads d'un bloc puissent partager des données
- besoin de plus de mémoire (rapide) que celle des registres,
- besoin de diminuer le nombre d'accès à la mémoire globale

Assurer la coalescence
(en **lecture** et **écriture**)

CPU



3

Principles of the *shared memory* Basic concepts

Avantages des kernels utilisant la mémoire globale et la mémoire *shared* :

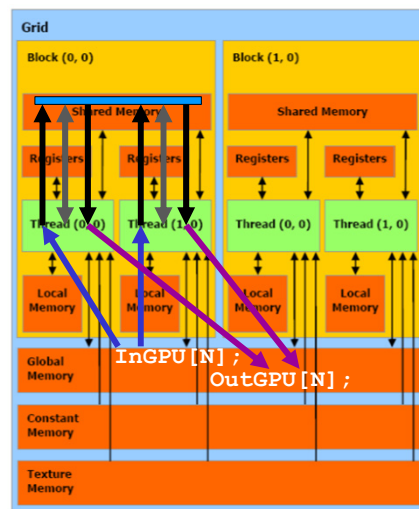
Motivations/Problèmes :

- besoin que les threads d'un bloc puissent partager des données
- besoin de plus de mémoire (rapide) que celle des registres,
- besoin de diminuer le nombre d'accès à la mémoire globale

Shared memory d'un multiprocesseur :

- 164 KB max par StreamMultiprocessor
- 2x48KB en static sur archi Turing ou Ampere (48KB par bloc)
- même technologie que le cache L1 (un peu plus lent que les registres)

- **partagée par tous les threads du bloc**
- **accès rapide sans contrainte**



4

Basic concepts

Kernel utilisant la *shared memory* ... sans partager de données (!)

Calcul : $res[i] = data[i]*data[i];$

Objectif : disposer de plus de mémoire qu'avec uniquement les registres.

Principe : les *threads* utilisent des tables partagées,
...mais différents *thread* accèdent à des cases différentes.

Hyp : $Nd = k.BLOCK_SIZE_X$

$Db = \{BLOCK_SIZE_X, 1, 1\}$

$Dg = \{Nd/BLOCK_SIZE_X, 1, 1\}$

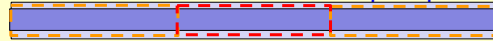
```

__global__ void k1D(void)
{
    // Collective definition of table in the shared memory
    __shared__ float shdata[BLOCK_SIZE_X];
    // Local computation result
    float res;
    // Compute data idx of the thread
    int idx = threadIdx.x + blockIdx.x*BLOCK_SIZE_X;
    // Read data from the global memory, store in the shared memory
    shdata[threadIdx.x] = InGPU[idx];
    // Compute result (just a computation example ...)
    res = shdata[threadIdx.x]*shdata[threadIdx.x];
    // Write result in the global memory
    OutGPU[idx] = res;
}

```

Le programmeur
« remplace l'algo
de cache » !

Les blocs sont juxtaposés



5

Advanced CUDA programming Part 1

1. Principles of the *Shared Memory*

- Basic concepts
- **Scheme of a basic ShM 2D-kernel**
- Scheme of a ShM 2D-kernel with loop

2. Examples of the use of *Shared Memory*

3. Atomic operations

4. Dynamic parallelism

5. Conclusion on CUDA programming

6

Scheme of a basic ShM 2D-kernel

```

__global__ void k2D(void)
{
  // Collective definition of table in shared memory
  __shared__ float shdata[BLOCK_SIZE_Y][BLOCK_SIZE_X];
  // Local computation result
  float res;
  // Local definition and computation of indexes in global memory
  int idxX = f(threadIdx.x, blockIdx.x, BLOCK_SIZE_X);
  int idxY = g(threadIdx.y, blockIdx.y, BLOCK_SIZE_Y);

  // Loading input data into the ShM, respecting certain index
  // limits in the global memory
  if (...) { No constraint Ensure coalescence
    shdata[threadIdx.y][threadIdx.x] = Input[idxY][idxX];
  }

  __syncthreads(); // REQUIRED: wait for all data loaded in ShM

  // Calculations using any data in the shared memory, but
  // respecting certain limits (boundaries)
  if (...) { No constraint
    res = ... shdata[...] [...] ... ;
    Output[idxY][idxX] = res;
  } Ensure coalescence
}
  
```

7

Advanced CUDA programming Part 1

1. Principles of the *Shared Memory*

- Basic concepts
- Scheme of a basic ShM 2D-kernel
- **Scheme of a ShM 2D-kernel with loop**

2. Examples of the use of *Shared Memory*

3. Atomic operations

4. Dynamic parallelism

5. Conclusion on CUDA programming

8

Scheme of a ShM 2D-kernel with loop

```

__global__ void k2D(void)
{
    // Collective definition of table in shared memory
    __shared__ float shdata[BLOCK_SIZE_Y][BLOCK_SIZE_X];
    // Local computation result
    float res;
    // Local definition and computation of indexes in global memory
    int idxX = f(threadIdx.x, blockIdx.x, BLOCK_SIZE_X);
    int idxY = g(threadIdx.y, blockIdx.y, BLOCK_SIZE_Y);
    for (int step = 0; step < ...; step++) {
        // - shared memory update
        ...
        __syncthreads();
        // - local computation
        res += ...
        __syncthreads();
    }
    if (...) {
        Output[idxY][idxX] = res;
    }
}
    
```

9

Scheme of a ShM 2D-kernel with loop

```

__global__ void k2D(void)
{
    .....
    for (int step = 0; step < ...; step++) {
        // - shared memory update: loading input data into the ShM
        if (...) {
            shdata[threadIdx.y][threadIdx.x] =
                Input[gg(idxY,step)][ff(idxX,step)];
        }
        __syncthreads(); // REQUIRED: wait for all ShM update are done
        // - local computation: using any data into the shared memory,
        if (...) {
            res += ... shdata[...][...] ... ;
        }
        __syncthreads(); // REQUIRED: wait for all ShM read are done
    }
    .....
}
    
```

10

Advanced CUDA programming Part 1

1. Principles of the *Shared Memory*
2. Examples of the use of *Shared Memory*
 - **Ex 1: Filtering & juxtaposed blocks**
 - Ex 2: Filtering & overlapping blocks
 - Ex 3: Matrix transposition & juxtaposed blocks
3. Atomic operations
4. Dynamic parallelism
5. Conclusion on CUDA programming

11

Examples of the use of *Shared Memory*

Ex 1: Filtering & juxtaposed blocks

Kernel utilisant la *shared memory* et partageant des données – v1

Calcul : `if (i > 0 && i < Nd-1)`
`res[i] = data[i-1]/4+data[i]/2+data[i+1]/4;`

Objectif : accélérer les accès répétés à une même donnée,
 éviter de lire plusieurs fois une même donnée en mémoire globale

Principe : table partagée, et accès à une même case par plusieurs *threads*

Hyp : $Nd = k * BLOCK_SIZE_X$

```
__global__ void k1D(void)
{
    // Collective definition of table in the shared memory
    __shared__ float shdata[BLOCK_SIZE_X];
    // Local computation result
    float res;
    // Compute data idx of the thread, read one element and sync.
    int idx = threadIdx.x + blockIdx.x*BLOCK_SIZE_X;
    shdata[threadIdx.x] = InGPU[idx];
    __syncthreads(); // REQUIRED !!
    .....
}
```

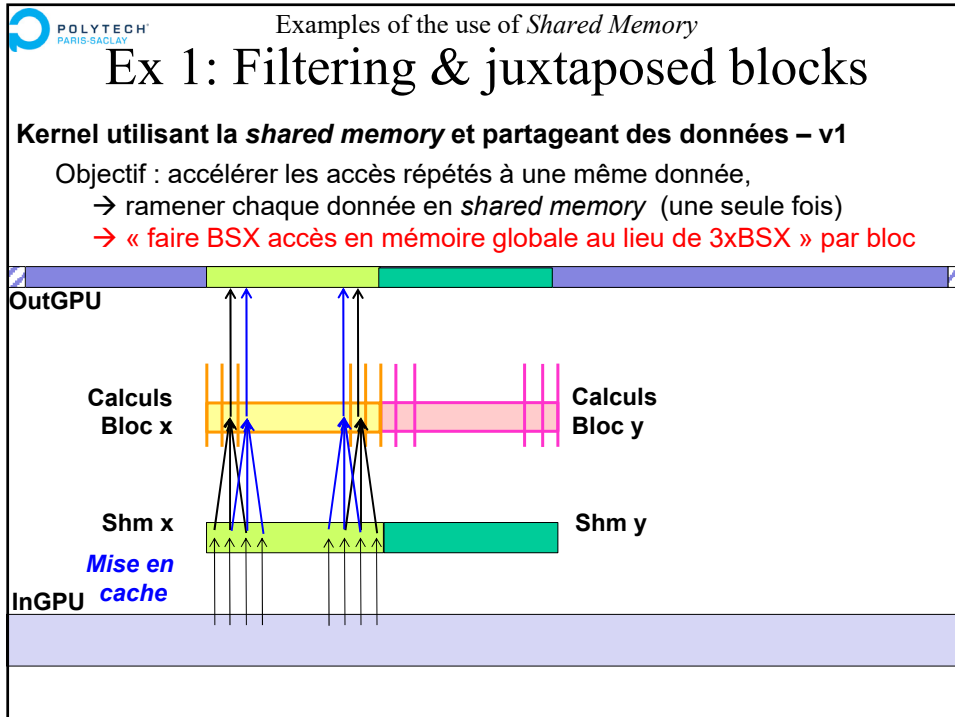
Db = {BLOCK_SIZE_X,1,1}
 Dg = {Nd/(BLOCK_SIZE_X),1,1}

Chaque thread du bloc a fini de charger une donnée en *shm*

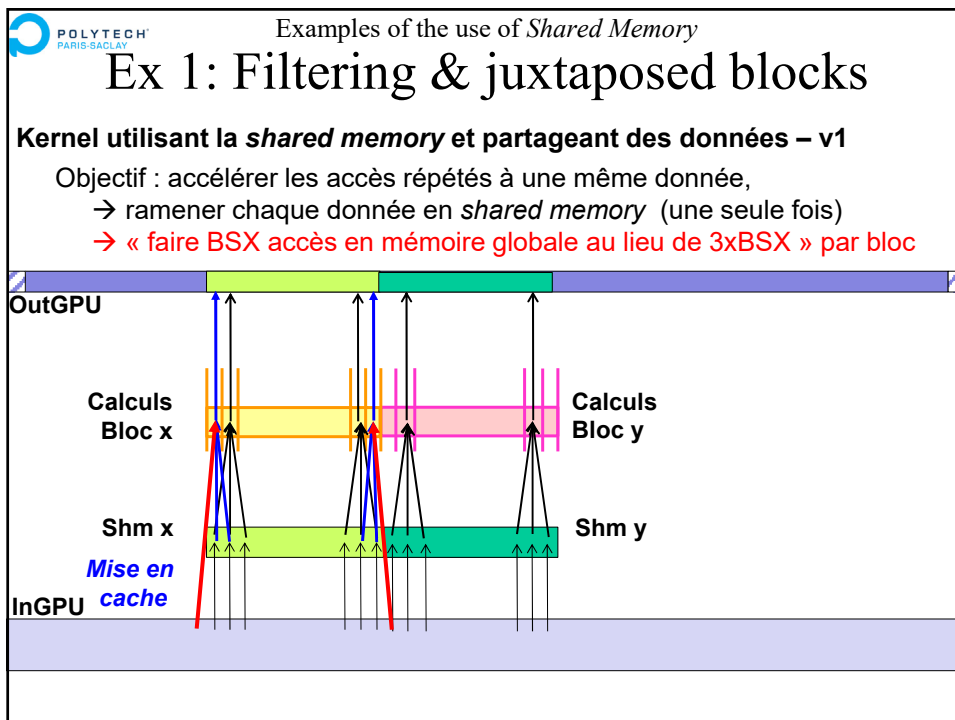
Les blocs sont juxtaposés



12



13



14

Ex 1: Filtering & juxtaposed blocks

Kernel utilisant la *shared memory* et partageant des données – v1

Principe : table partagée, et accès à une même case par plusieurs *threads*

Hyp : $Nd = k * BLOCK_SIZE_X$

Db = {BLOCK_SIZE_X, 1, 1}
Dg = {Nd / (BLOCK_SIZE_X), 1, 1}

```

... ..
if (idx > 0 && idx < Nd-1) {
    // Compute the left and right values
    float left, right;
    if (threadIdx.x == 0)
        left = InGPU[idx-1];
    else
        left = shdata[threadIdx.x-1];
    if (threadIdx.x == BLOCK_SIZE_X-1)
        right = InGPU[idx+1];
    else
        right = shdata[threadIdx.x+1];
    // Compute result
    res = left*0.25f + shdata[threadIdx.x]*0.5f + right*0.25f;
    // Write result in the global memory
    OutGPU[idx] = res;
}

```

Accès à des données non chargées dans la *shm* par les *threads* du bloc

Exploitation de données dans la *shm*

Les blocs sont juxtaposés

15

Ex 1: Filtering & juxtaposed blocks

Kernel utilisant la *shared memory* et partageant des données – v2

Objectif : **ne ramener chaque donnée qu'une seule fois** en mémoire locale

Principe : tables partagées, et accès aux mêmes cases

Hyp : $Nd \neq k * BLOCK_SIZE_X$

Db = {BLOCK_SIZE_X, 1, 1}
Dg = {Nd / (BLOCK_SIZE_X) +
(Nd % BLOCK_SIZE_X ? 1 : 0), 1, 1}

```

__global__ void k1D(void)
{
    // Collective definition of table in the shared memory
    __shared__ float shdata[BLOCK_SIZE_X];
    // Local computation result
    float res;
    // Compute data idx of the thread, read one element and sync.
    idx = threadIdx.x + blockIdx.x * BLOCK_SIZE_X;
    if (idx < Nd) {
        shdata[threadIdx.x] = InGPU[idx];
    }
    __syncthreads(); // REQUIRED !!
    .....
}

```

Les blocs sont juxtaposés
mais le dernier bloc déborde

16

Ex 1: Filtering & juxtaposed blocks

Kernel utilisant la *shared memory* et partageant des données – v2

Objectif : **ne ramener chaque donnée qu'une seule fois** en mémoire locale


Principe : tables partagées, et accès aux même cases

Hyp : $Nd \neq k \cdot BLOCK_SIZE_X$

```
Db = {BLOCK_SIZE_X,1,1}
Dg = {Nd/(BLOCK_SIZE_X)+
      (Nd%BLOCK_SIZE_X ? 1 : 0),1,1}
```

```
.....
if (idx > 0 && idx < Nd-1 /* && idx < Nd */) {
    // Compute the left and right values
    float left, right;
    if (threadIdx.x == 0)
        left = InGPU[idx-1];
    else
        left = shdata[threadIdx.x-1];
    if (threadIdx.x == BLOCK_SIZE_X-1)
        right = InGPU[idx+1];
    else
        right = shdata[threadIdx.x+1];
    // Compute result
    res = left*0.25f + shdata[threadIdx.x]*0.5f + right*0.25f;
    // Write result in the global memory
    OutGPU[idx] = res;
}
}
```

Les blocs sont juxtaposés
mais le dernier bloc déborde



17

Advanced CUDA programming Part 1

1. Principles of the *Shared Memory*
2. Examples of the use of *Shared Memory*
 - Ex 1: Filtering & juxtaposed blocks
 - **Ex 2: Filtering & overlapping blocks**
 - Ex 3: Matrix transposition & juxtaposed blocks
3. Atomic operations
4. Dynamic parallelism
5. Conclusion on CUDA programming

18

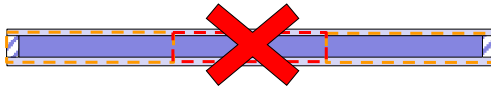
Ex 2: Filtering & overlapping blocks

Kernel utilisant la *shared memory* et partageant des données – v3

Objectif : **toutes** les données des calculs de chaque bloc cachées en *shm*

→ afin de pouvoir écrire le code suivant :

```
__global__ void k1D(void)
{
    .....
    if (...) {
        // Compute result (another computation example...)
        res = shdata[threadIdx.x-1]*0.25f +
            shdata[threadIdx.x]*0.50f +
            shdata[threadIdx.x+1]*0.25f;
        // Write result in the global memory
        OutGPU[idx] = res;
    }
}
```



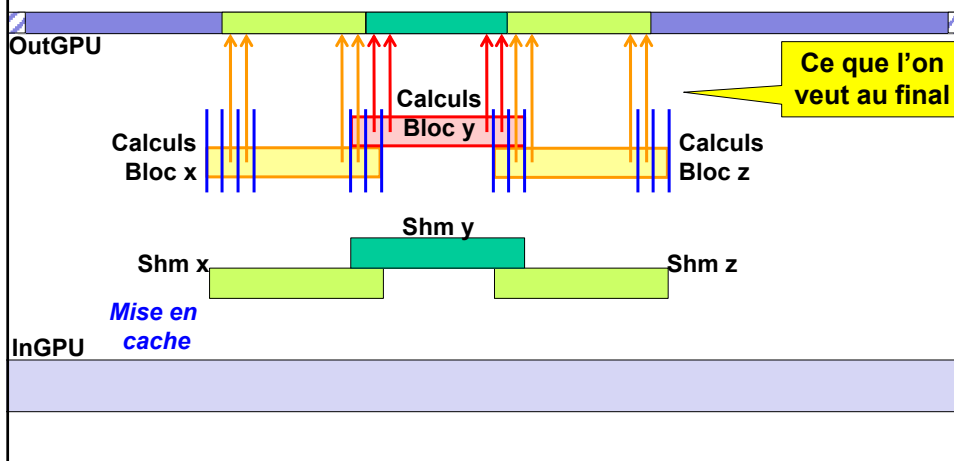
→ Des blocs juxtaposés ne suffisent plus

19

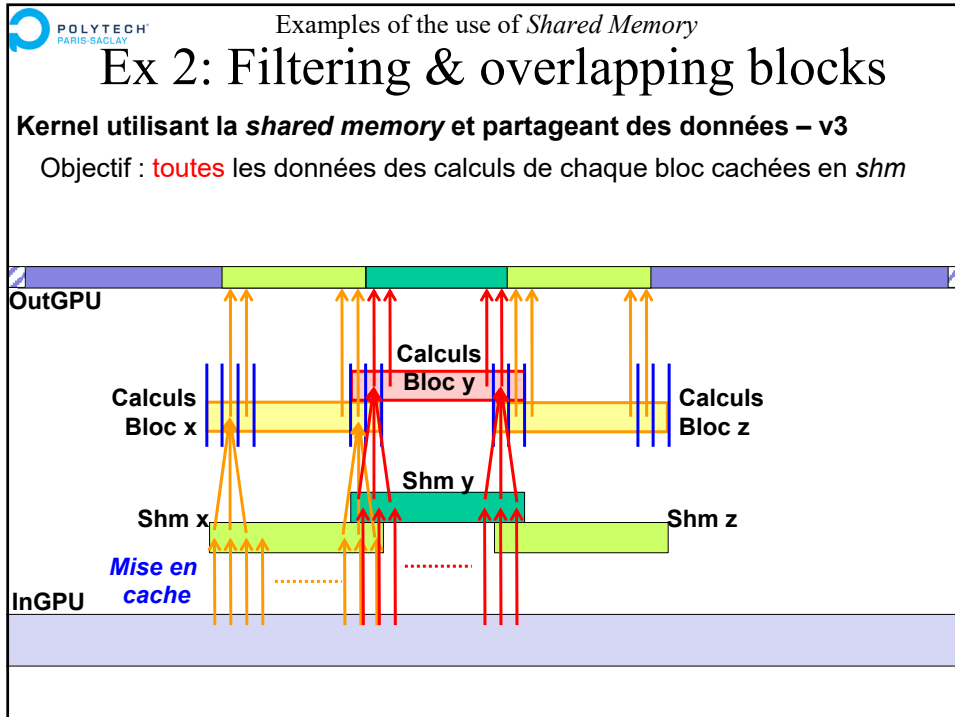
Ex 2: Filtering & overlapping blocks

Kernel utilisant la *shared memory* et partageant des données – v3

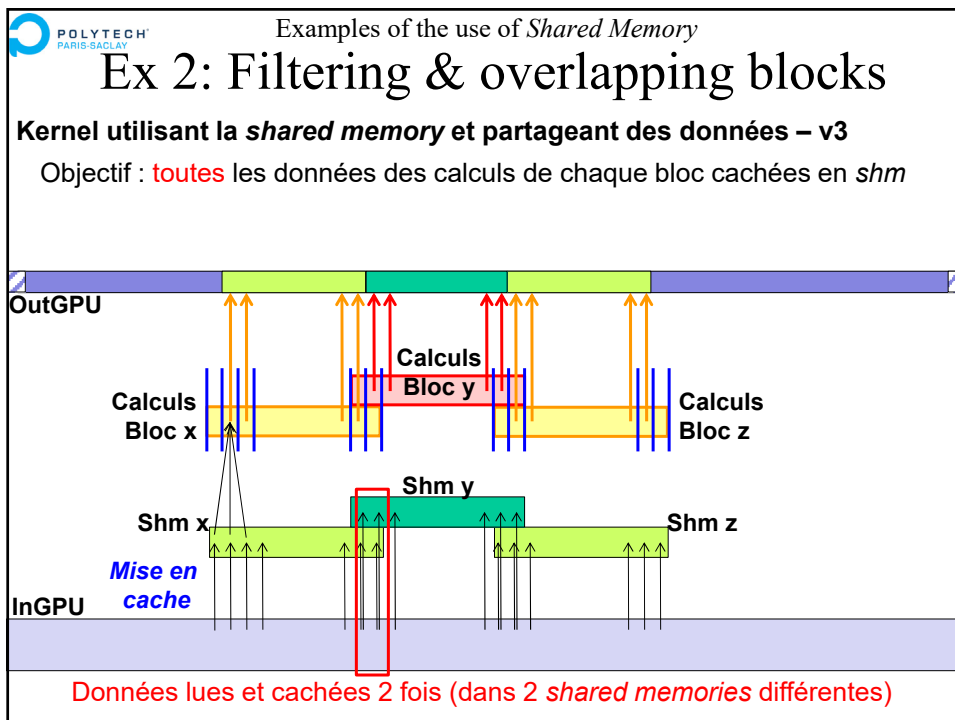
Objectif : **toutes** les données des calculs de chaque bloc cachées en *shm*



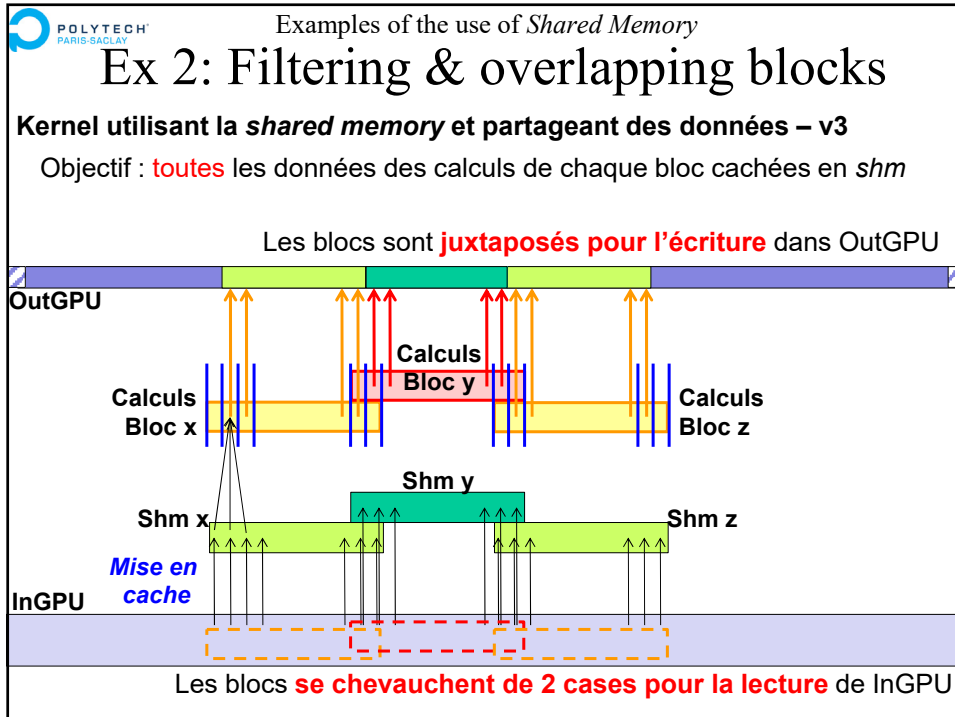
20



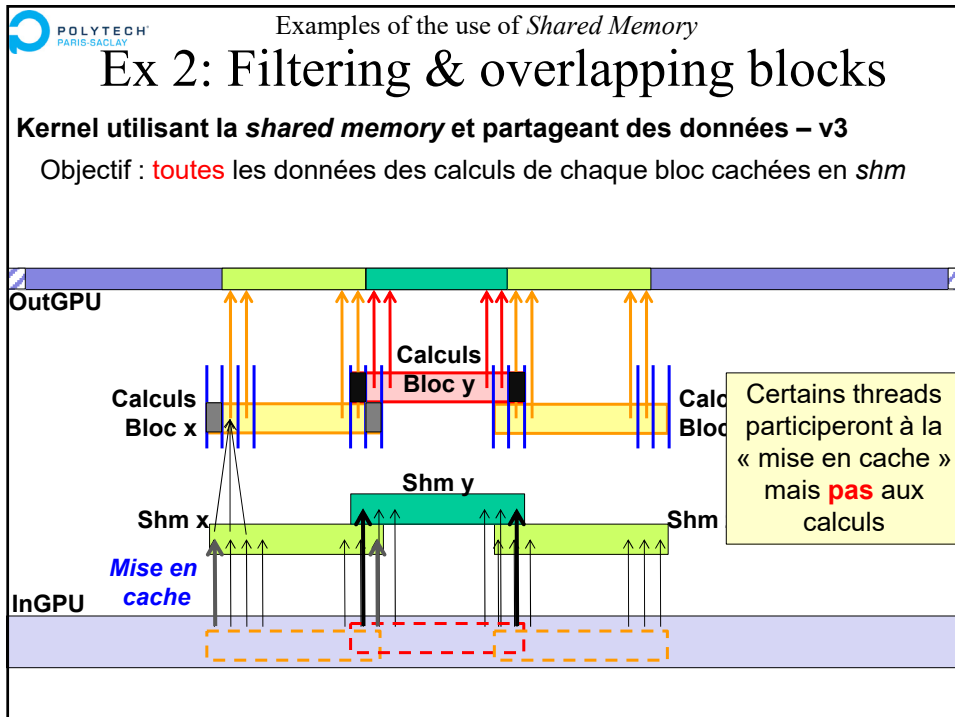
21



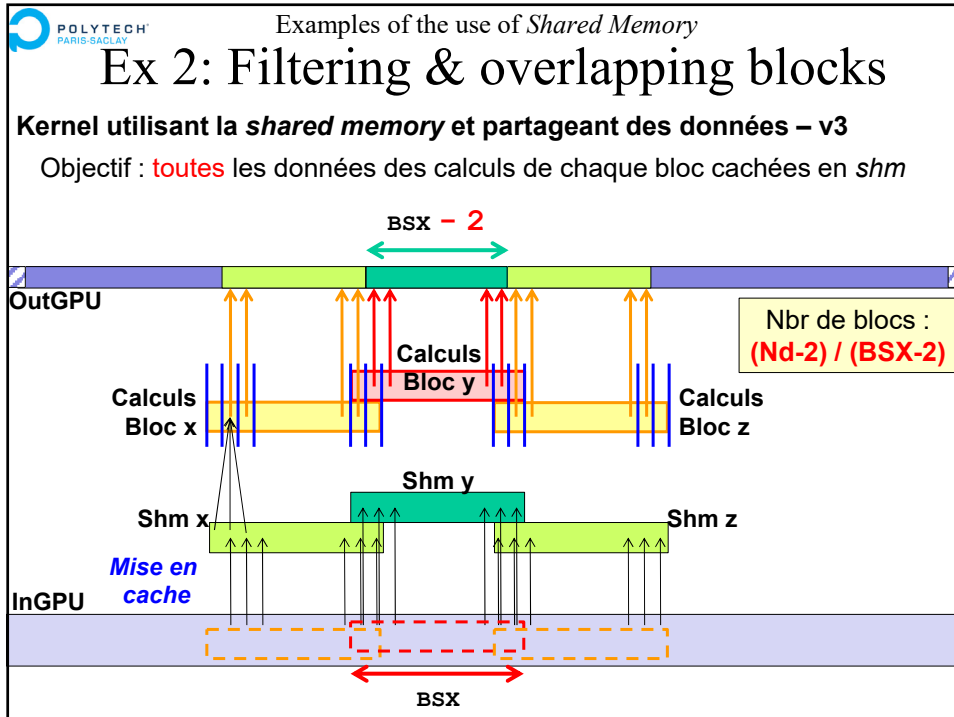
22



23



24



25

POLYTECH PARIS-SACLAY

Ex 2: Filtering & overlapping blocks

Kernel utilisant la *shared memory* et partageant des données – v3

Objectif : **toutes** les données des calculs de chaque bloc cachées en *shm*

Principe : table partagée, et accès aux même cases depuis plusieurs *threads*

Hyp : $Nd-2 = k*(BSX-2)$

```

global __ void k1D(void)
{
    __shared__ float shdata[BSX]; // Collective shm definition
    float res; // Local variable (register)

    // Compute data idx of the thread, read one element and sync.
    int idx = threadIdx.x + blockIdx.x*(BSX-2);
    shdata[threadIdx.x] = InGPU[idx];

    __syncthreads(); // REQUIRED !!

    // Computation
    if (threadIdx.x > 0 && threadIdx.x < BSX-1
        /* && idx > 0 && idx < Nd-1 */) {
        res = shdata[threadIdx.x-1]*0.25f + // Computation example
              shdata[threadIdx.x]*0.50f +
              shdata[threadIdx.x+1]*0.25f;
        OutGPU[idx] = res; // Result storage
    }
}

```

Db = {BSX, 1, 1}

Dg = { (Nd-2) / (BSX-2), 1, 1 }

Les blocs doivent se chevaucher

26

Ex 2: Filtering & overlapping blocks

Kernel utilisant la *shared memory* et partageant des données – v4

Objectif : **toutes** les données des calculs de chaque bloc cachées en *shm*

Principe : table partagée, et accès aux même cases depuis plusieurs *threads*

Hyp : $Nd-2 \neq k*(BSX-2)$

```

global __void k1D(void)
{
    __shared__ float shdata[BSX]; //
    float res; // Local variable (register)


    // Compute data idx of the thread, read one element and sync.
    int idx = threadIdx.x + blockIdx.x*(BSX-2);
    if (idx < Nd) {shdata[threadIdx.x] = InGPU[idx];}
    __syncthreads(); // REQUIRED !!

    // Computation
    if (threadIdx.x > 0 && threadIdx.x < BSX-1
        /* && idx > 0 */&& idx < Nd-1) {
        res = shdata[threadIdx.x-1]*0.25f + // Computation example
              shdata[threadIdx.x]*0.50f +
              shdata[threadIdx.x+1]*0.25f;
        OutGPU[idx] = res;
    }
}

```

new!
new!
new!

Les blocs doivent se chevaucher, et le dernier déborde



27

Advanced CUDA programming Part 1

1. Principles of the *Shared Memory*
2. Examples of the use of *Shared Memory*
 - Ex 1: Filtering & juxtaposed blocks
 - Ex 2: Filtering & overlapping blocks
 - **Ex 3: Matrix transposition & juxtaposed blocks**
3. Atomic operations
4. Dynamic parallelism
5. Conclusion on CUDA programming

28

Ex 3: Matrix transposition & juxtaposed blocks

Kernel *naïf* de transposition d'une matrice

```

__global__ void Transpose_v0(float *MT, float *M,
                             int nbLigM, int nbColM)
{
    int ligM = threadIdx.y + blockIdx.y*BSIZE_XY_KT0;
    int colM = threadIdx.x + blockIdx.x*BSIZE_XY_KT0;
    if (ligM < nbLigM && colM < nbColM)
        MT[colM*nbLigM + ligM] = M[ligM*nbColM + colM];
    //MT[colM][ligM] = M[ligM][colM]
}
    
```

Accès NON coalescent

Accès coalescent

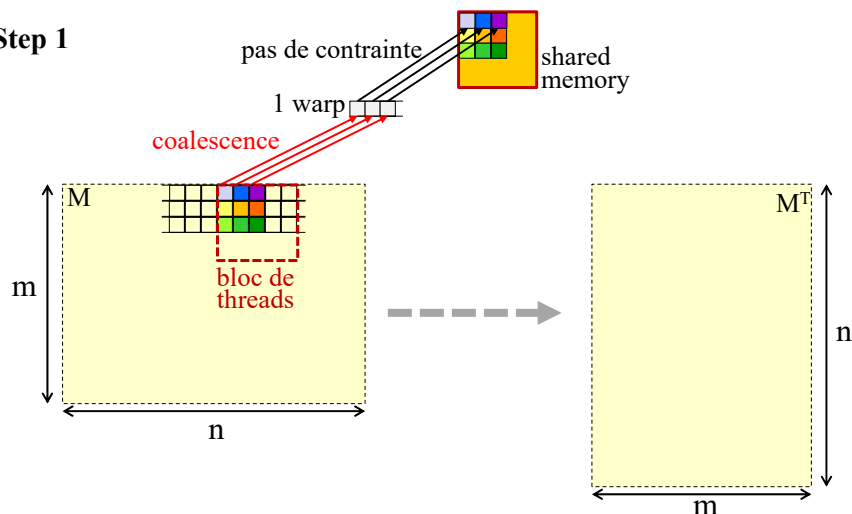
→ Utiliser la *shared memory* pour être coalescent à la lecture et à l'écriture...

29

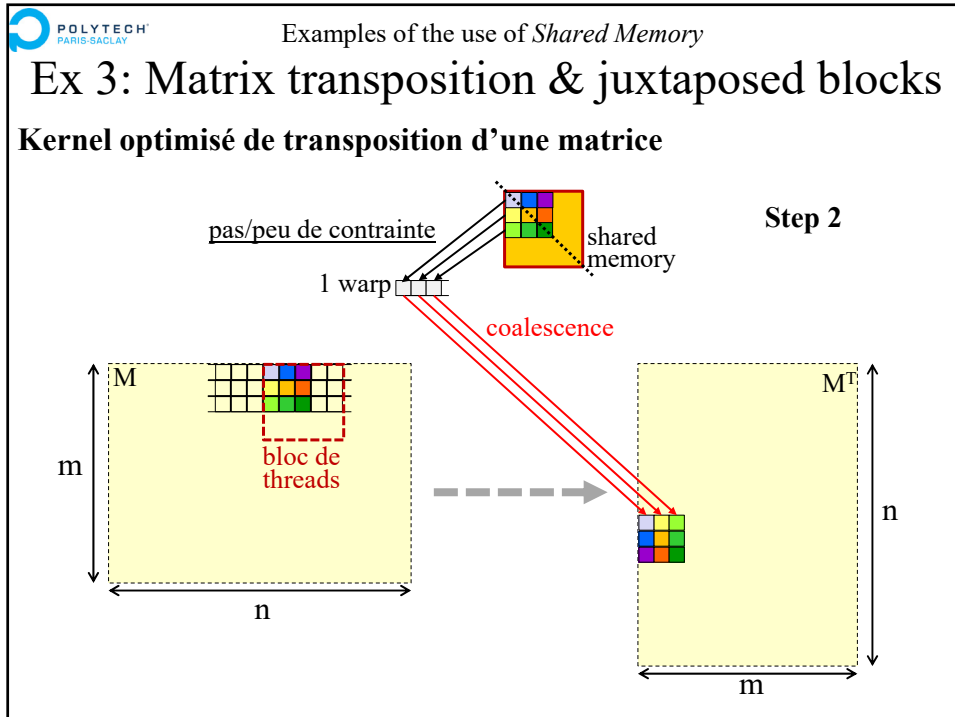
Ex 3: Matrix transposition & juxtaposed blocks

Kernel optimisé de transposition d'une matrice

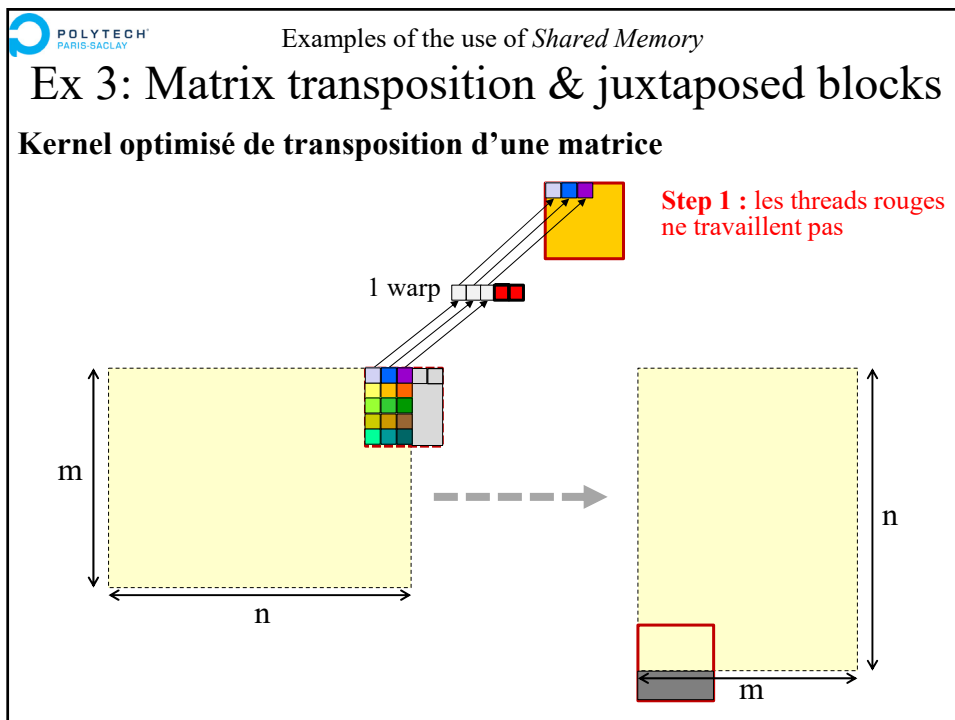
Step 1



30



31



32

POLYTECH PARIS-SACLAY

Examples of the use of *Shared Memory*

Ex 3: Matrix transposition & juxtaposed blocks

Kernel optimisé de transposition d'une matrice

1 warp

Step 1 : les threads rouges ne travaillent pas

m

n

m

→ Deux conditions différentes par thread
→ Deux « boundaries » par threads

33

POLYTECH PARIS-SACLAY

Examples of the use of *Shared Memory*

Ex 3: Matrix transposition & juxtaposed blocks

Kernel optimisé de transposition d'une matrice

1 warp

Step 1 : les threads rouges ne travaillent pas

Step 2 : les threads verts travaillent !
Ce sont les mêmes threads!

m

n

m

→ Deux conditions différentes par thread
→ Deux « boundaries » par threads

34

Ex 3: Matrix transposition & juxtaposed blocks

Kernel optimisé de transposition d'une matrice

```

__global__ void Transpose_v1(float *MT, float *M,
                             int nbLigM, int nbColM)
{
    int firstLigBlock = blockIdx.y*BSIZE_XY_KT1;
    int firstColBlock = blockIdx.x*BSIZE_XY_KT1;
    int ligM = firstLigBlock + threadIdx.y;
    int colM = firstColBlock + threadIdx.x;
    int ligMT = firstColBlock + threadIdx.y;
    int colMT = firstLigBlock + threadIdx.x;
    __shared__ float shM[BSIZE_XY_KT1][BSIZE_XY_KT1];

    if (ligM < nbLigM && colM < nbColM) // Load data in cache
        shM[threadIdx.y][threadIdx.x] = M[ligM*nbColM + colM];

    __syncthreads(); // Wait for all data in cache

    if (ligMT < nbColM && colMT < nbLigM) // Write back the cache
        MT[ligMT*nbLigM + colMT] = shM[threadIdx.x][threadIdx.y];
}

```

35

Advanced CUDA programming Part 1

1. Principles of the *Shared Memory*
2. Examples of the use of *Shared Memory*
- 3. Atomic operations**
4. Dynamic parallelism
5. Conclusion on CUDA programming

36

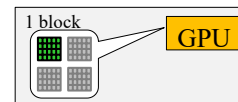
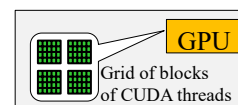
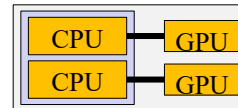
3 kinds of atomic functions

Principes :

- Des fonctions qui doivent s'exécuter sur le GPU
- Des fonctions qui apportent une section critique et une « mutex »

Mais 3 types de « mutex » :

- `atomicXxx_system(...)` : exclusion avec tous les threads du même programme sur le(s) GPU et sur le(s) CPU
- `atomicXxx(...)` : exclusion avec tous les threads CUDA du même programme sur le même GPU
- `atomicXxx_block(...)` : exclusion avec tous les threads CUDA du même bloc de threads

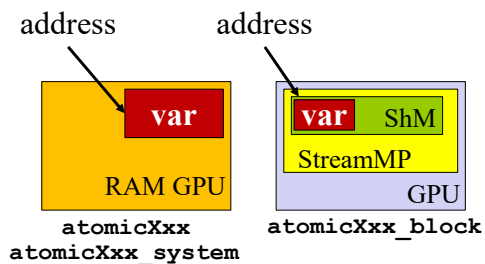


37

Classical atomic functions

Les atomicAdd :

```
int atomicAdd(int* address, int val);
unsigned int atomicAdd(unsigned int* address, unsigned int val);
unsigned long long int atomicAdd(unsigned long long int* address,
                                unsigned long long int val);
float atomicAdd(float* address, float val);
double atomicAdd(double* address, double val);
```



Entrée en mutex

Lecture de `var` (`old_var`)
Ajout de `val` à `old_var`
Ecriture dans `var`

Sortie de mutex

Retournent l'ancienne valeur de `var` (`old_var`)

38

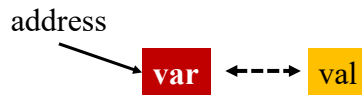
Classical atomic functions

Les atomicSub :

```
int atomicSub(int* address, int val);
unsigned int atomicSub(unsigned int* address, unsigned int val);
```

Les atomicExch :

```
int atomicExch(int* address, int val);
unsigned int atomicExch(unsigned int* address, unsigned int val);
unsigned long long int atomicExch(unsigned long long int* address,
                                unsigned long long int val);
float atomicExch(float* address, float val);
```



Retourne l'ancienne valeur de var (*old_var*)

39

Classical atomic functions

Les atomicMin :

```
int atomicMin(int* address, int val);
unsigned int atomicMin(unsigned int* address, unsigned int val);
unsigned long long int atomicMin(unsigned long long int* address,
                                unsigned long long int val);
```



Les atomicMax :

```
int atomicMax(int* address, int val);
unsigned int atomicMax(unsigned int* address, unsigned int val);
unsigned long long int atomicMax(unsigned long long int* address,
                                unsigned long long int val);
```

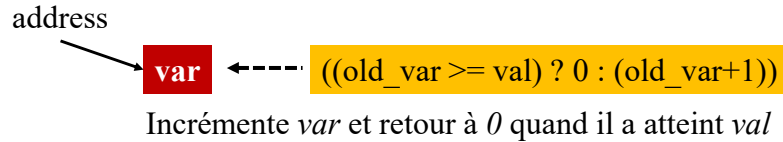
Retourne l'ancienne valeur de var (*old_var*)

40

Classical atomic functions

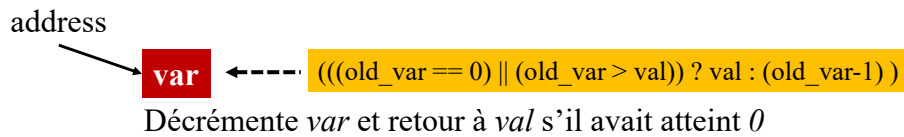
Les atomicInc :

```
unsigned int atomicInc(unsigned int* address, unsigned int val);
```



Les atomicDec :

```
unsigned int atomicDec(unsigned int* address, unsigned int val);
```



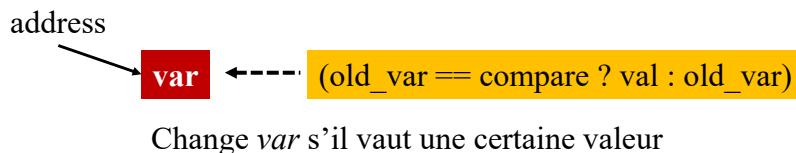
Retournent l'ancienne valeur de *var* (*old_var*)

41

Classical atomic functions

Les atomicCAS (*compare and set*) :

```
int atomicCAS(int* address, int compare, int val);
unsigned int atomicCAS(unsigned int* address,
    unsigned int compare, unsigned int val);
unsigned long long int atomicCAS(unsigned long long int* address,
    unsigned long long int compare,
    unsigned long long int val);
unsigned short int atomicCAS(unsigned short int *address,
    unsigned short int compare,
    unsigned short int val);
```



Retournent l'ancienne valeur de *var* (*old_var*)

42

Classical atomic functions

Les fct atomiques « bitwise » :

```
int atomicAnd(int* address, int val);
unsigned int atomicAnd(unsigned int* address, unsigned int val);
unsigned long long int atomicAnd(unsigned long long int* address,
                                unsigned long long int val);
```

Opération « and » sur chaque bit de var

`var ← (old_var & val)`

```
int atomicOr(int* address, int val);
unsigned int atomicOr(unsigned int* address, unsigned int val);
unsigned long long int atomicOr(unsigned long long int* address,
                                unsigned long long int val);
```

```
int atomicXor(int* address, int val);
unsigned int atomicXor(unsigned int* address, unsigned int val);
unsigned long long int atomicXor(unsigned long long int* address,
                                unsigned long long int val);
```

43

« System » and « block » atomic functions

L'exclusion des fonctions atomique précédentes porte :

- sur les threads CUDA du même programme
- sur le même GPU

→ Des versions « _system » et « _block » existent

Toutes les fct atomiques ne sont pas disponibles sur tous les GPU :

- compute capability < 6.0 only `atomicXxx` only (device wide)
- compute capability < 7.2 no `atomicXxx_system` (system-wide)
- Autres restrictions sur les fct portant sur les doubles ou sur les half

44

Advanced CUDA programming Part 1

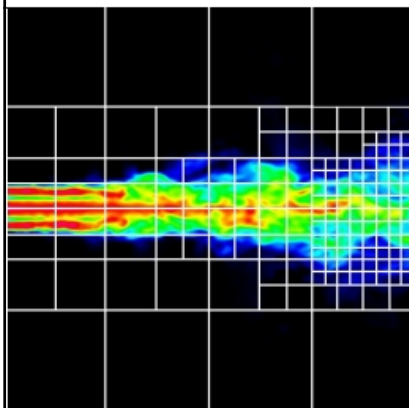
1. Principles of the *Shared Memory*
2. Examples of the use of *Shared Memory*
3. Atomic operations
4. **Dynamic parallelism**
5. Conclusion on CUDA programming

45

Dynamic parallelism

Un thread GPU peut lancer d'autres threads GPU

- Un thread définit et lance lui-même une grille de blocs de threads
- Très utile pour des maillages adaptatifs

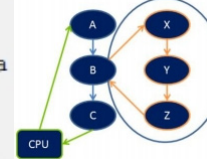


```

global ChildKernel(void* data){
    //Operate on data
}

global ParentKernel(void *data){
    if (threadIdx.x == 0) {
        ChildKernel<<<1, 32>>>(data);
        cudaThreadSynchronize();
    }
    syncthreads();
    //Operate on data
}

// In Host Code
ParentKernel<<<8, 32>>>(data);
    
```



46

Dynamic parallelism

Un thread GPU peut lancer d'autres threads GPU

Remplacer une grosse grille de blocs par :

- une petite grille mère dont les threads créent des grilles filles (même de tailles identiques)
- avec un *stream* différent par grille fille

→ Permet parfois de gagner du temps à l'exécution et d'augmenter les performances !
(permet de mieux saturer les ressources d'un GPU)

47

Advanced CUDA programming Part 1

1. Principles of the *Shared Memory*
2. Examples of the use of *Shared Memory*
3. Atomic operations
4. Dynamic parallelism
- 5. Conclusion on CUDA programming**

48

Conclusion on CUDA programming

Une nouvelle façon de programmer :

- Remplace la programmation *vectorielle* sur les architectures *vectérielles* disparues
- Partage des concepts avec la programmation *threads+vectorisation* des architectures CPU *multicores+SIMD*
- Demande une période d'apprentissage et debug difficile...

Démarche d'apprentissage conseillée :

- Apprendre les bases de CUDA puis les optimisations principales (*CUDA C Best Practices Guide*)
- Guider les développements par la mesure de la performance atteinte
- Apprendre à identifier si un algorithme est adapté au GPU
... avant de le développer sur GPU

49

Conclusion on CUDA programming

Les bonnes pratiques :

- Ecrire des kernels coalescents et non-divergents
- Utiliser la *shared memory* avec un « algo de cache dédié au pb »
... ne pas oublier de resynchroniser les threads si nécessaire !
- Sur des structures de données 2D ou plus : implanter des grilles et des blocs 2D (ou 3D) et chercher les bonnes tailles de blocs

Performances :

- Parfois des gains *spectaculaires* vis-à-vis d'un cœur CPU et en ne considérant que les temps de calculs
- Mais souvent un gain de 2 à 10 seulement en considérant les transferts de données et vis-à-vis d'un code parallèle et optimisé sur un serveur dual-CPU (serveur standard) !

50

Advanced CUDA programming Part 1

End