

GP-GPU

CUDA programming with Shared Memory

St phane Vialle

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

CUDA programming with Shared Memory

1. Principles of the *Shared Memory*

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

2. Algorithm and code examples

3. Optimized reduction

Principles of the *shared memory*

Basic concepts

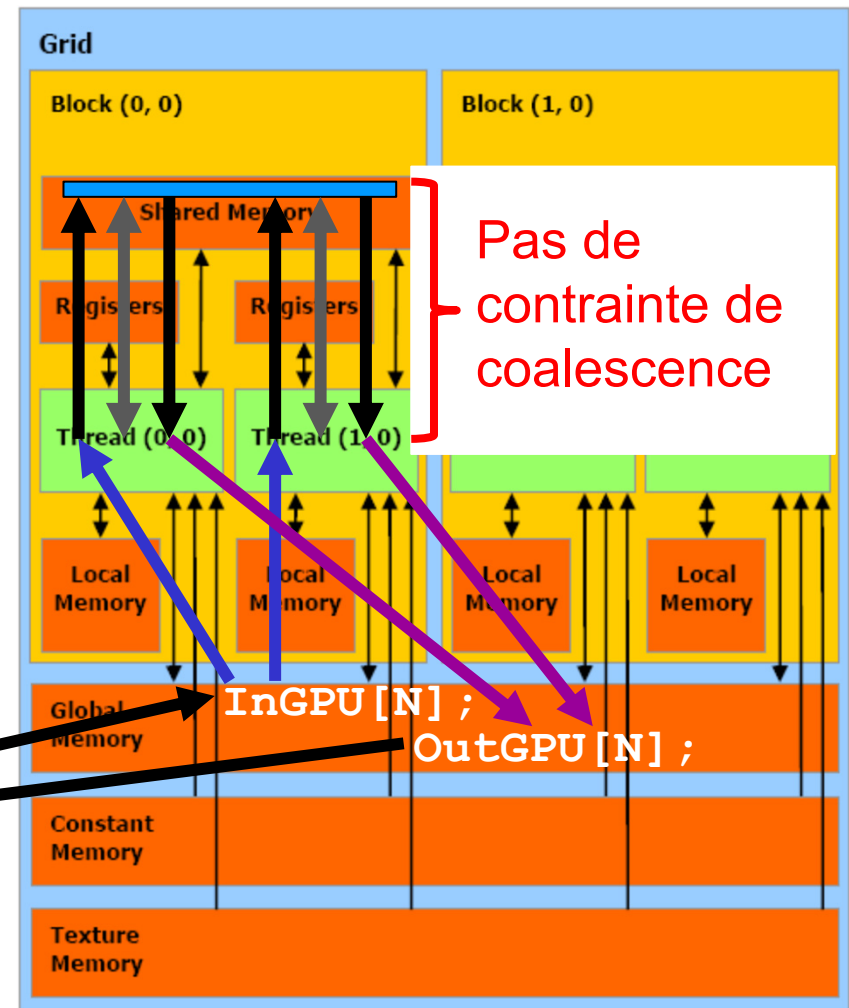
Avantages des kernels utilisant la mémoire globale et la mémoire *shared* :

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



Principles of the *shared memory*

Basic concepts

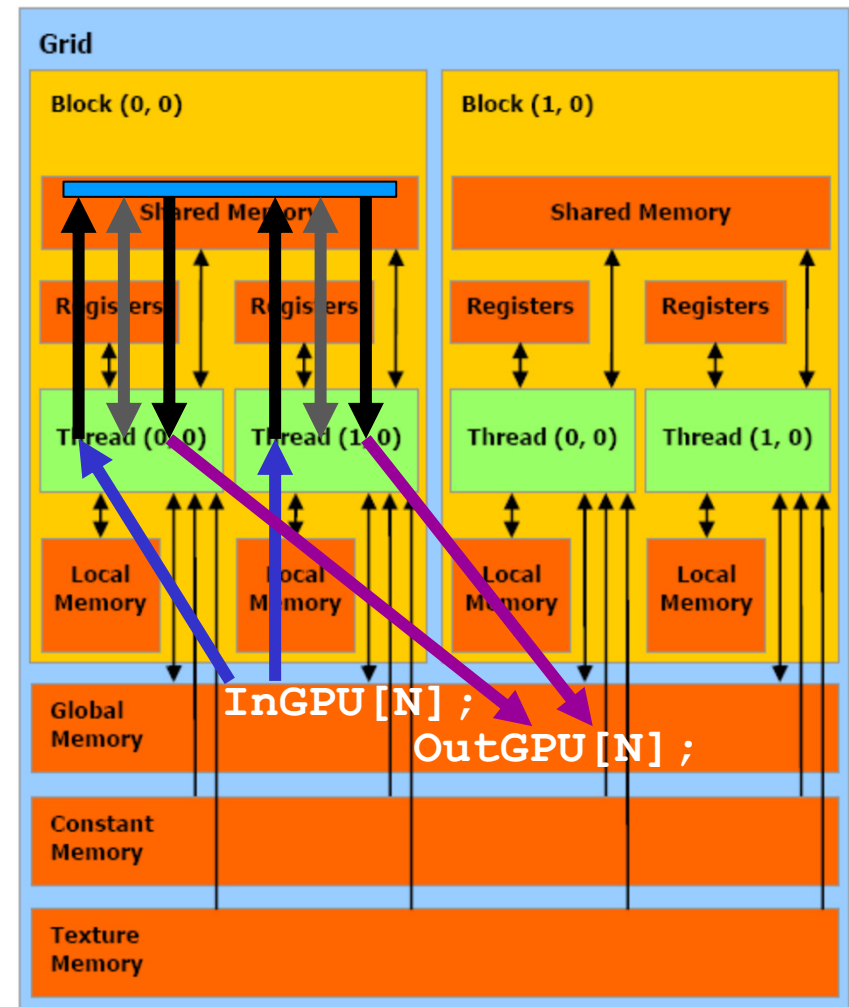
Avantages des kernels utilisant la mémoire globale et la mémoire *shared* :

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 :

- 64KB/multiproc sur archi Pascal.
- 2x48KB sur archi Turing (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**



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 : $N_d = k \cdot \text{BLOCK_SIZE_X}$

```
__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
```

```
    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;
```

```
}
```

`Db = {BLOCK_SIZE_X,1,1}`

`Dg = {Nd/BLOCK_SIZE_X,1,1}`

Le programmeur
« remplace l'algo
de cache » !

Les blocs sont juxtaposés



CUDA programming with Shared Memory

1. Principles of the *Shared Memory*

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

2. Algorithm and code examples

3. Optimized reduction

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
```

CUDA programming with Shared Memory

1. Principles of the *Shared Memory*

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

2. Algorithm and code examples

3. Optimized reduction

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;
    }
}
```

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
    }

    .....
}
```

CUDA programming with Shared Memory

1. Principles of the *Shared Memory*
2. **Algorithm and code examples**
 - **Ex 1: Filtering & juxtaposed blocks**
 - Ex 2: Filtering & overlapping blocks
 - Ex 3: Matrix transposition & juxtaposed blocks
3. Optimized reduction

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 : $Nd = k \cdot BLOCK_SIZE_X$

```
__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*

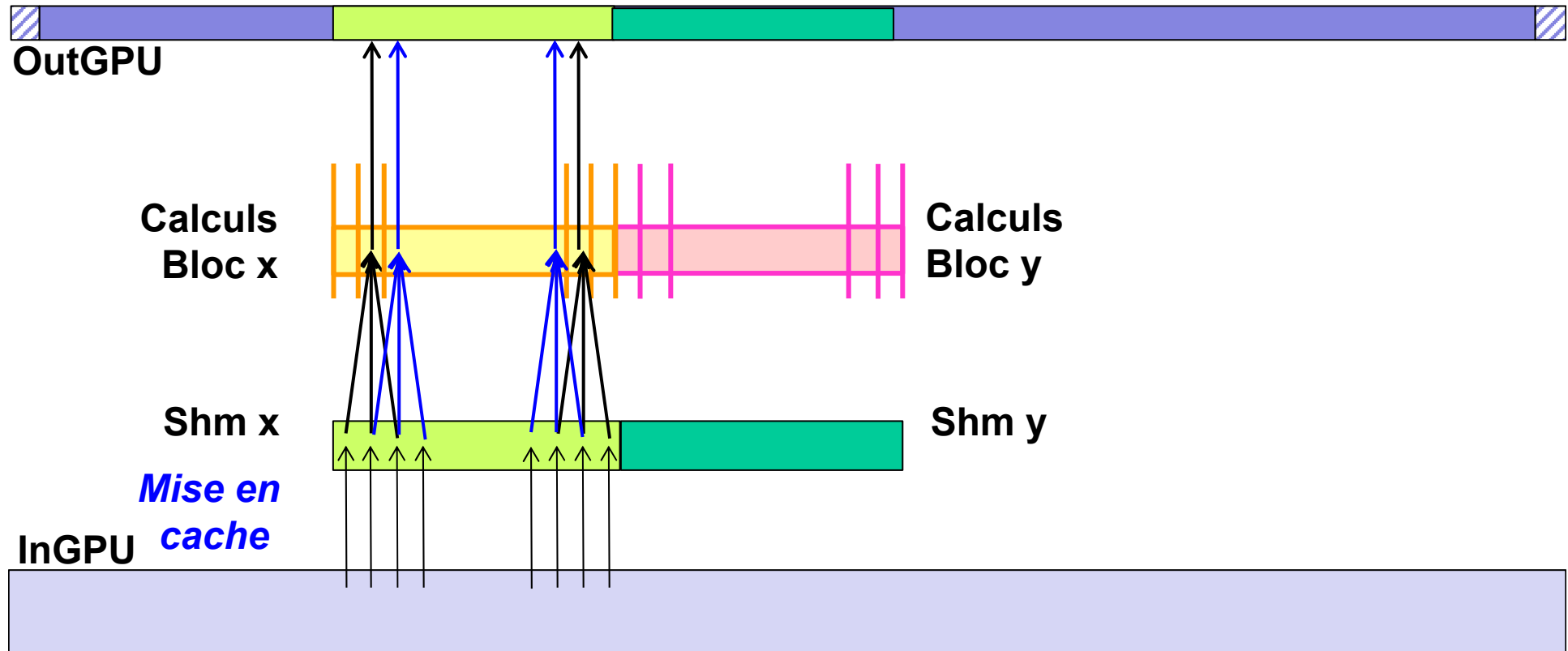
Les blocs sont juxtaposés



Ex 1: Filtering & juxtaposed blocks

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

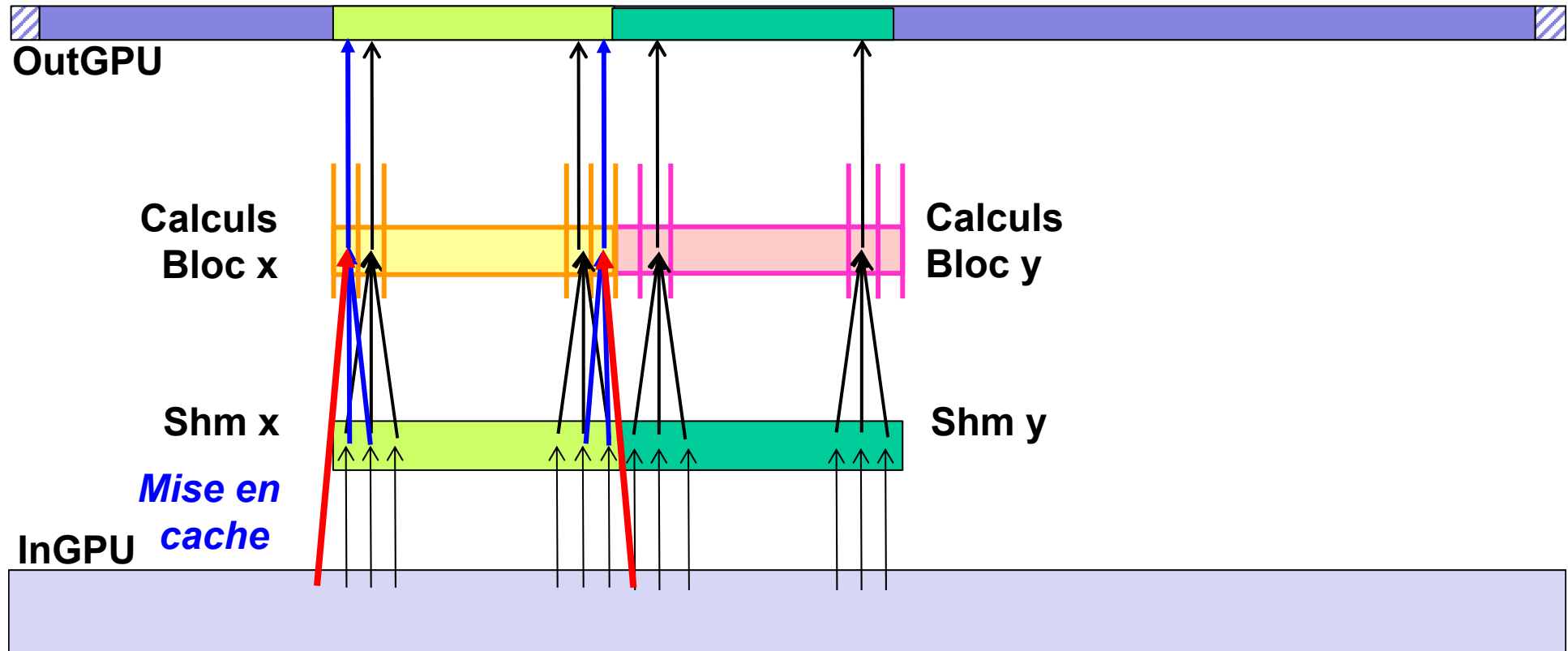
Objectif : accélérer les accès répétés à une même donnée,
 → ramener chaque donnée en *shared memory* (une seule fois)
 → « faire BSX accès en mémoire globale au lieu de 3xBSX »



Ex 1: Filtering & juxtaposed blocks

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

Objectif : accélérer les accès répétés à une même donnée,
 → ramener chaque donnée en *shared memory* (une seule fois)
 → « faire BSX accès en mémoire globale au lieu de 3xBSX »



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 : $N_d = k \cdot \text{BLOCK_SIZE_X}$

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

$\text{Dg} = \{N_d / (\text{BLOCK_SIZE_X}), 1, 1\}$

... ..

```
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



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 : $N_d \neq k \cdot \text{BLOCK_SIZE_X}$

```
__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.
```

```
    idx = threadIdx.x + blockIdx.x * BLOCK_SIZE_X;
```

```
    if (idx < Nd) {
```

```
        shdata[threadIdx.x] = InGPU[idx];
```

```
    }
```

```
    __syncthreads(); // REQUIRED !!
```

```
    .....
```

```
Db = {BLOCK_SIZE_X, 1, 1}
```

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

Les blocs sont juxtaposés
mais le dernier bloc déborde



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 : $N_d \neq k \cdot \text{BLOCK_SIZE_X}$

```
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



CUDA programming with Shared Memory

1. Principles of the *Shared Memory*
2. **Algorithm and code examples**
 - Ex 1: Filtering & juxtaposed blocks
 - **Ex 2: Filtering & overlapping blocks**
 - Ex 3: Matrix transposition & juxtaposed blocks
3. Optimized reduction

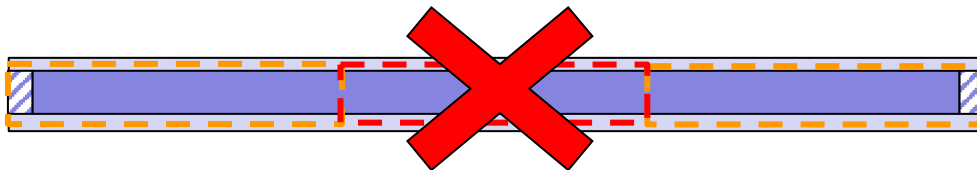
Ex 2: Filtering & overlapping blocks

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

Objectif : **toutes** les données cachées en *shared memory*.

→ 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;
    }
}
```

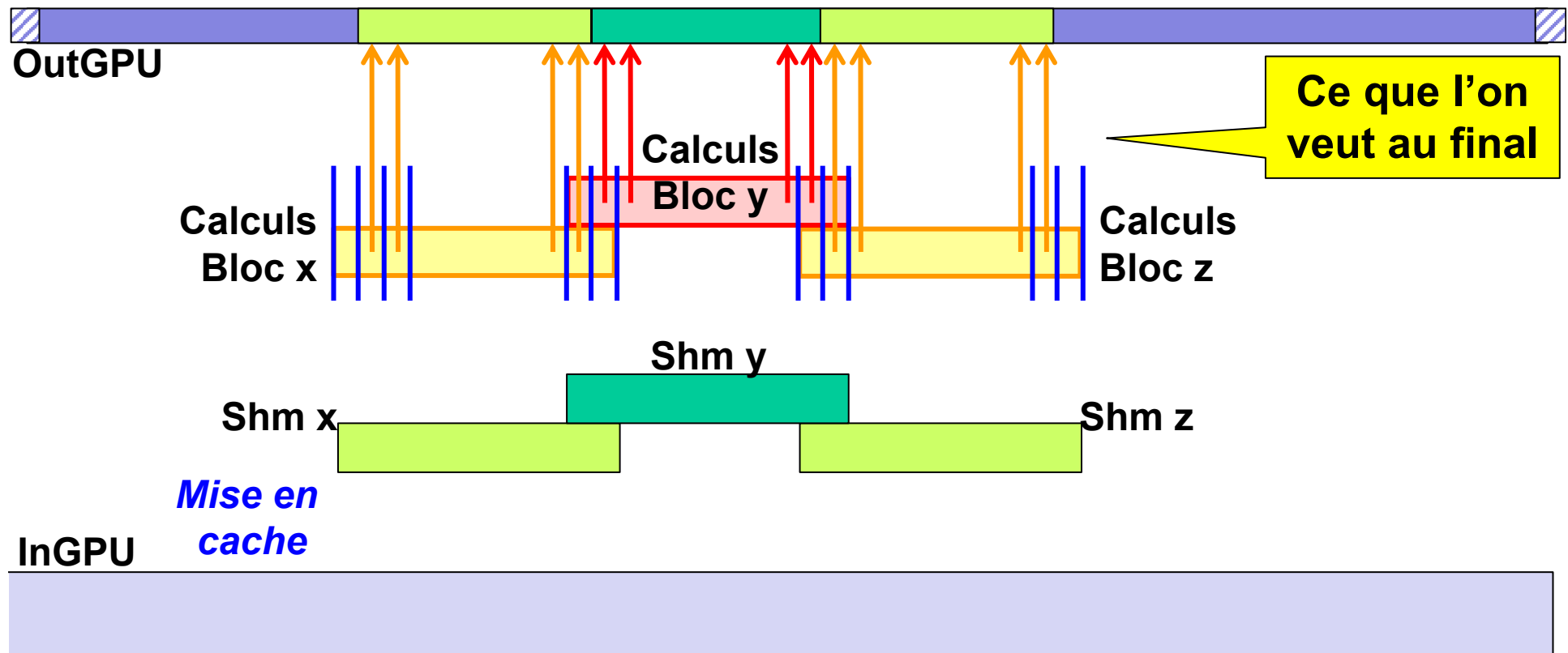


→ Des blocs juxtaposés ne suffisent plus

Ex 2: Filtering & overlapping blocks

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

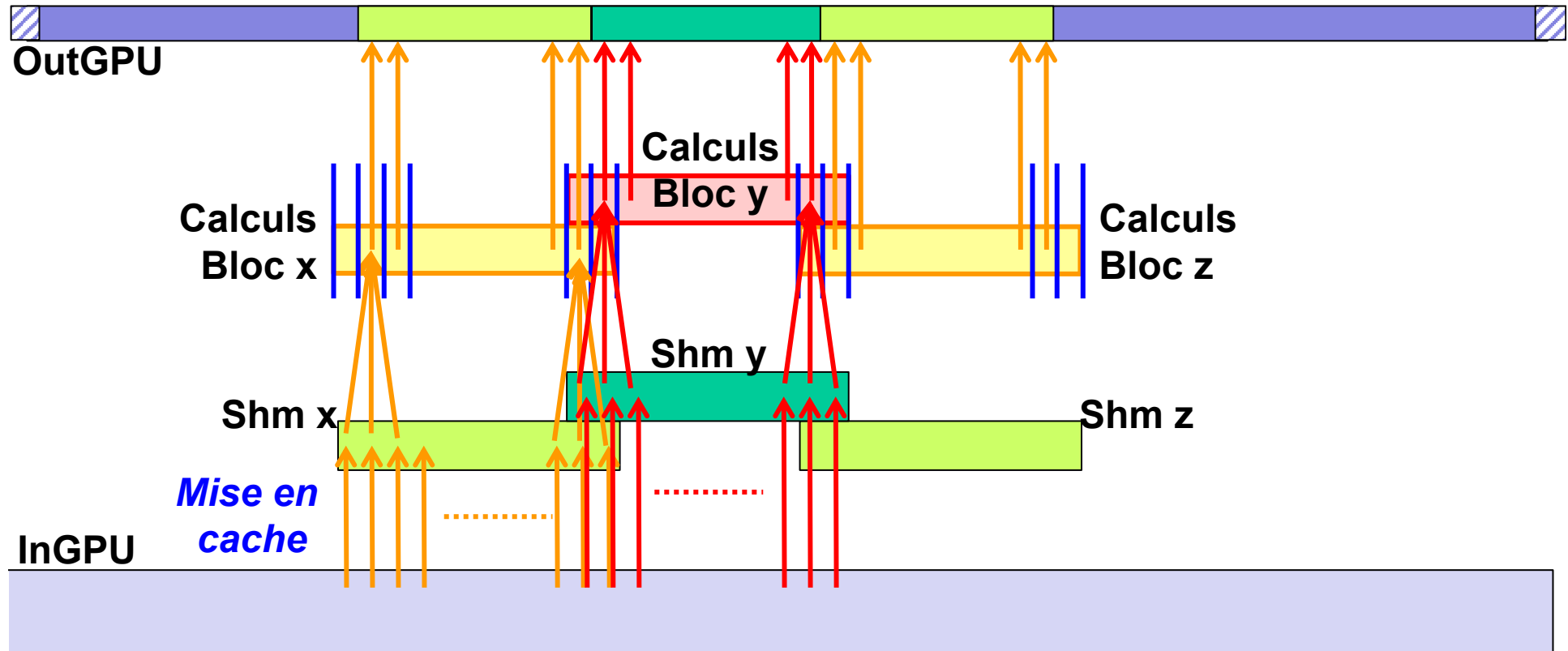
Objectif : **toutes** les données cachées en *shared memory*.



Ex 2: Filtering & overlapping blocks

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

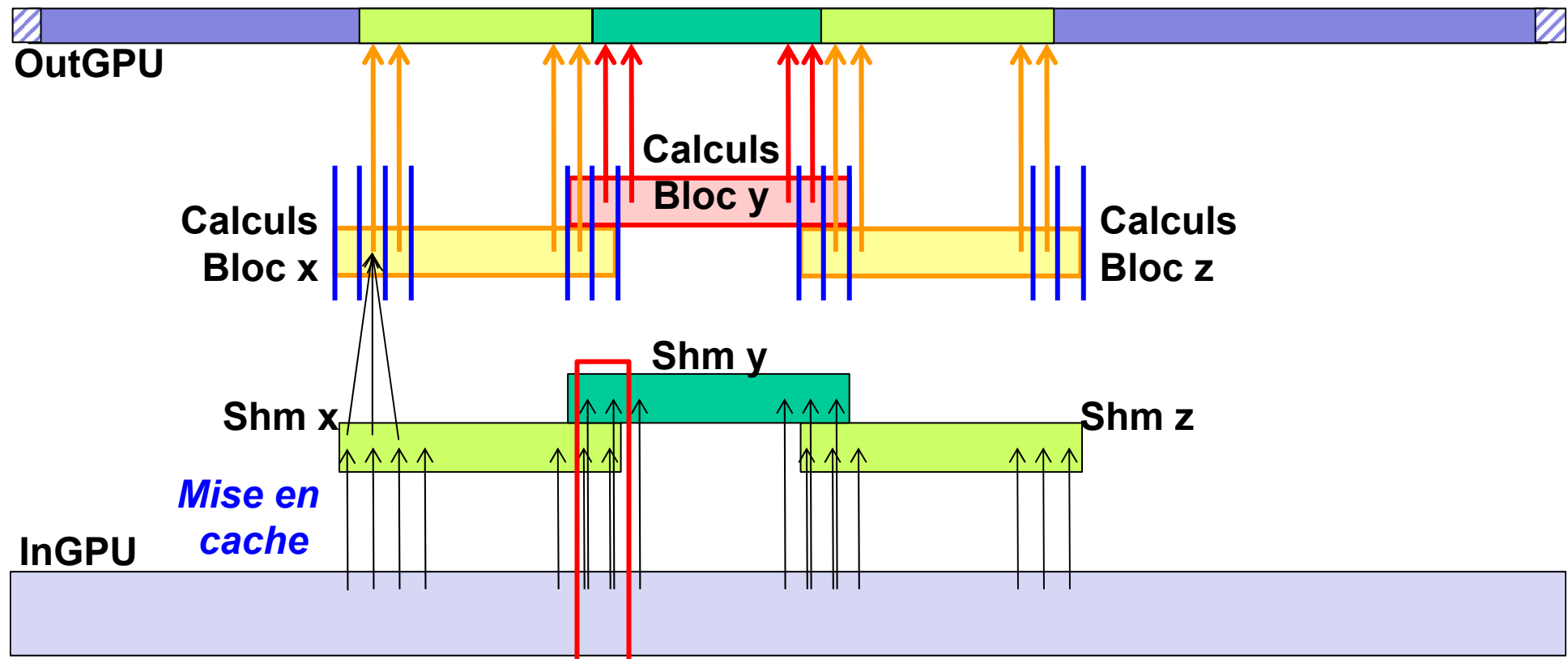
Objectif : **toutes** les données cachées en *shared memory*.



Ex 2: Filtering & overlapping blocks

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

Objectif : **toutes** les données cachées en *shared memory*.

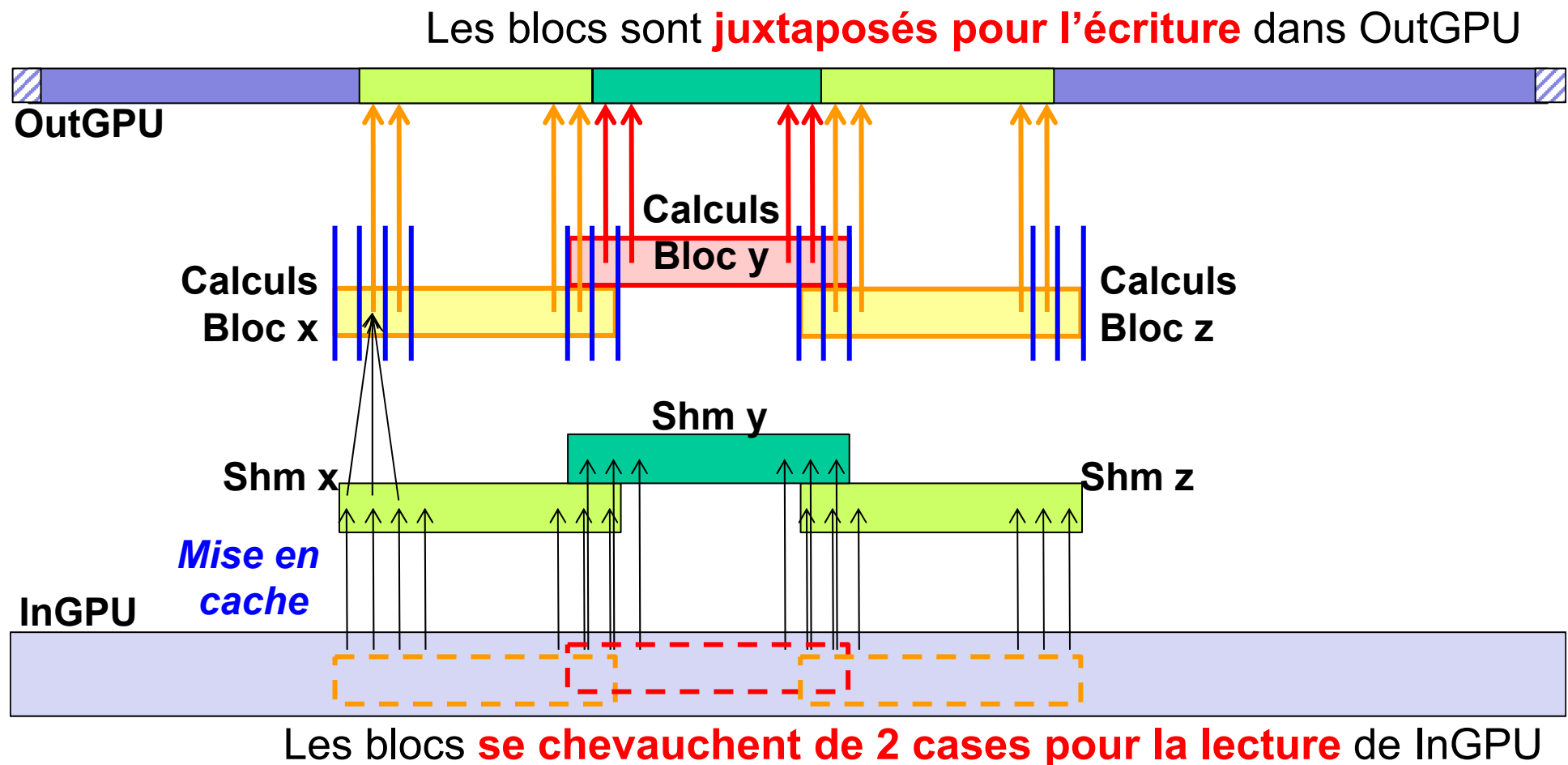


Données lues et cachées 2 fois (dans 2 *shared memories* différentes)

Ex 2: Filtering & overlapping blocks

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

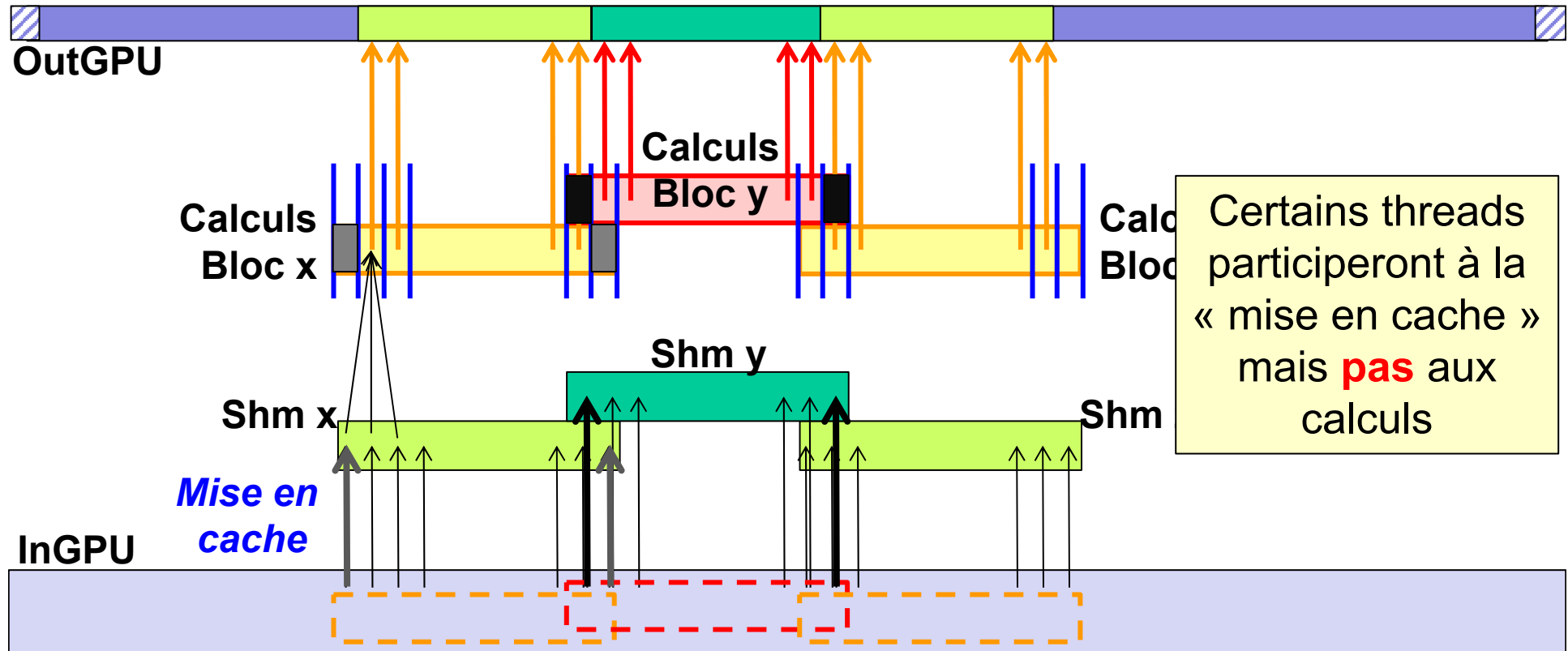
Objectif : **toutes** les données cachées en *shared memory*.



Ex 2: Filtering & overlapping blocks

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

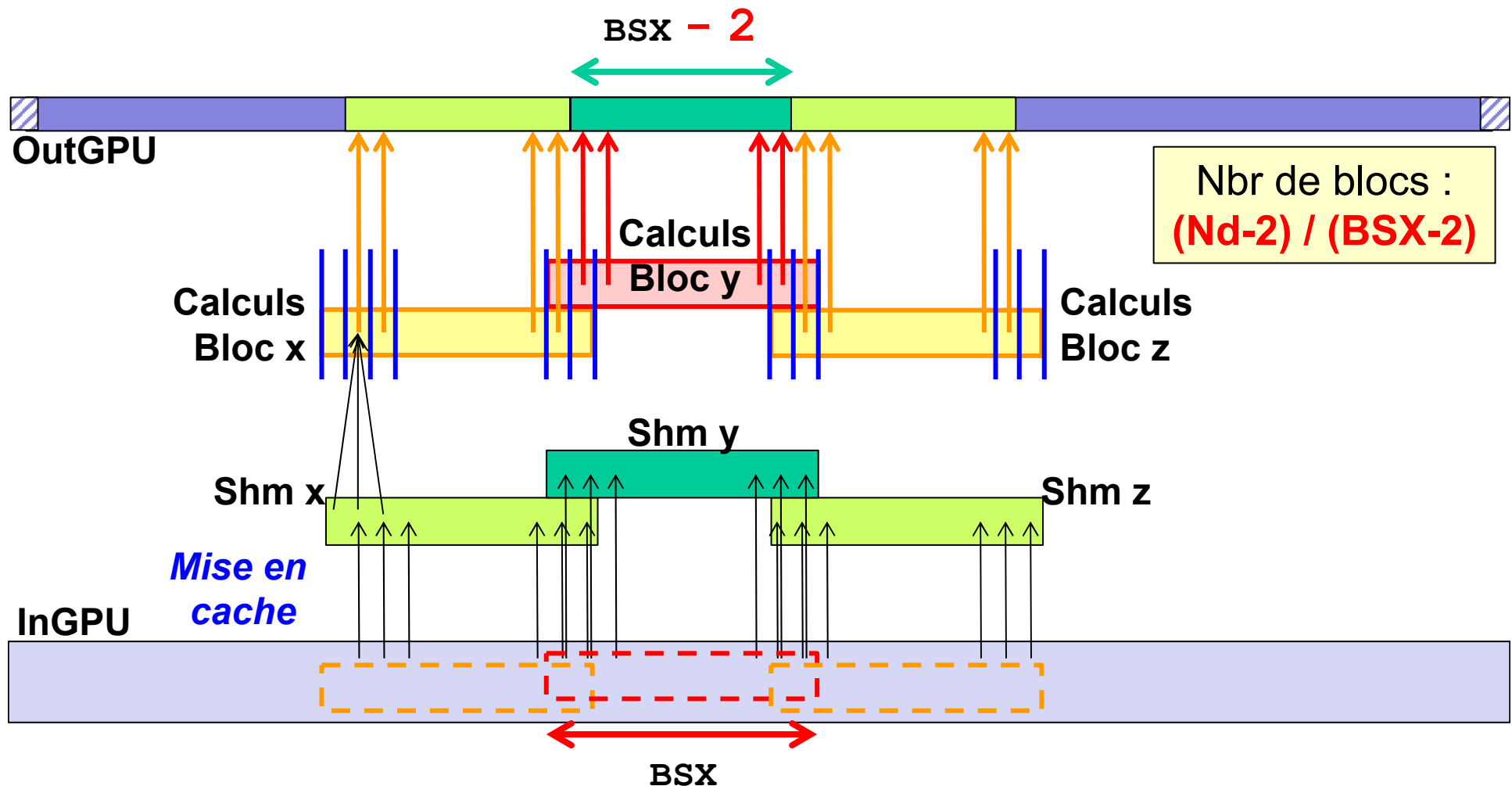
Objectif : **toutes** les données cachées en *shared memory*.



Ex 2: Filtering & overlapping blocks

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

Objectif : **toutes** les données des calculs d'un bloc cachées en *shm*



Ex 2: Filtering & overlapping blocks

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

Objectif : **toutes** les données cachées en *shared memory*

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

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

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

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

```
__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
    }
}
```

Les blocs doivent se chevaucher



Ex 2: Filtering & overlapping blocks

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

Objectif : **toutes** les données cachées en *shared memory*

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 +
```

```
        shdata[threadIdx.x]*0.50f +
```

```
        shdata[threadIdx.x+1]*0.25f;
```

```
    OutGPU[idx] = res;
```

```
}
```

```
}
```

Db = {BSX,1,1}

Dg = { (Nd-2) / (BSX-2) +
((Nd-2) % (BSX-2) ? 1 : 0) , 1, 1 }

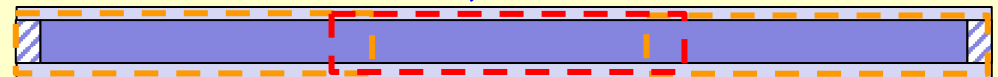
new !

new !

new !

// Computation example

Les blocs doivent se chevaucher, et le dernier déborde



CUDA programming with Shared Memory

1. Principles of the *Shared Memory*
2. **Algorithm and code examples**
 - Ex 1: Filtering & juxtaposed blocks
 - Ex 2: Filtering & overlapping blocks
 - **Ex 3: Matrix transposition & juxtaposed blocks**
3. Optimized reduction

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

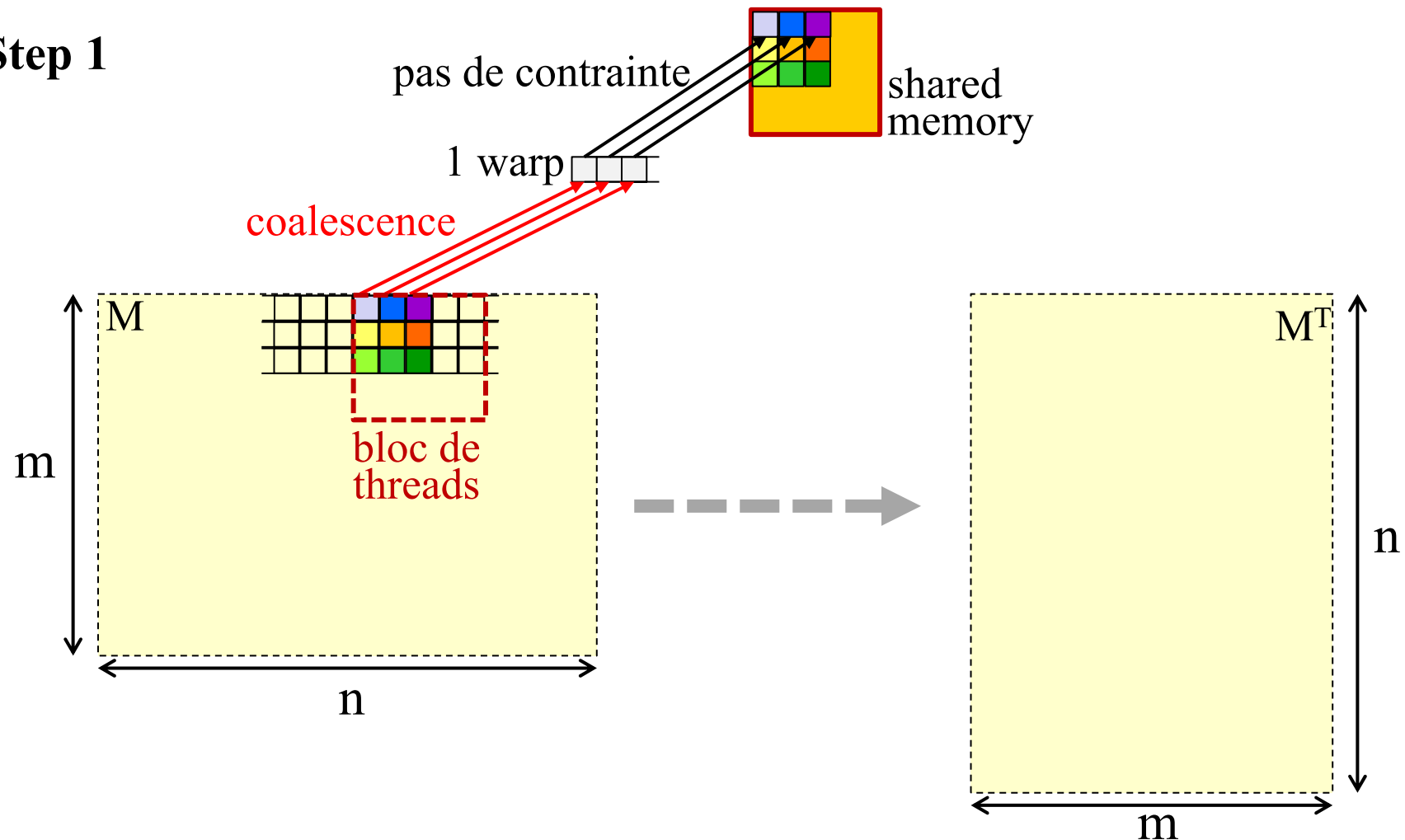
Accès coalescent

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

Ex 3: Matrix transposition & juxtaposed blocks

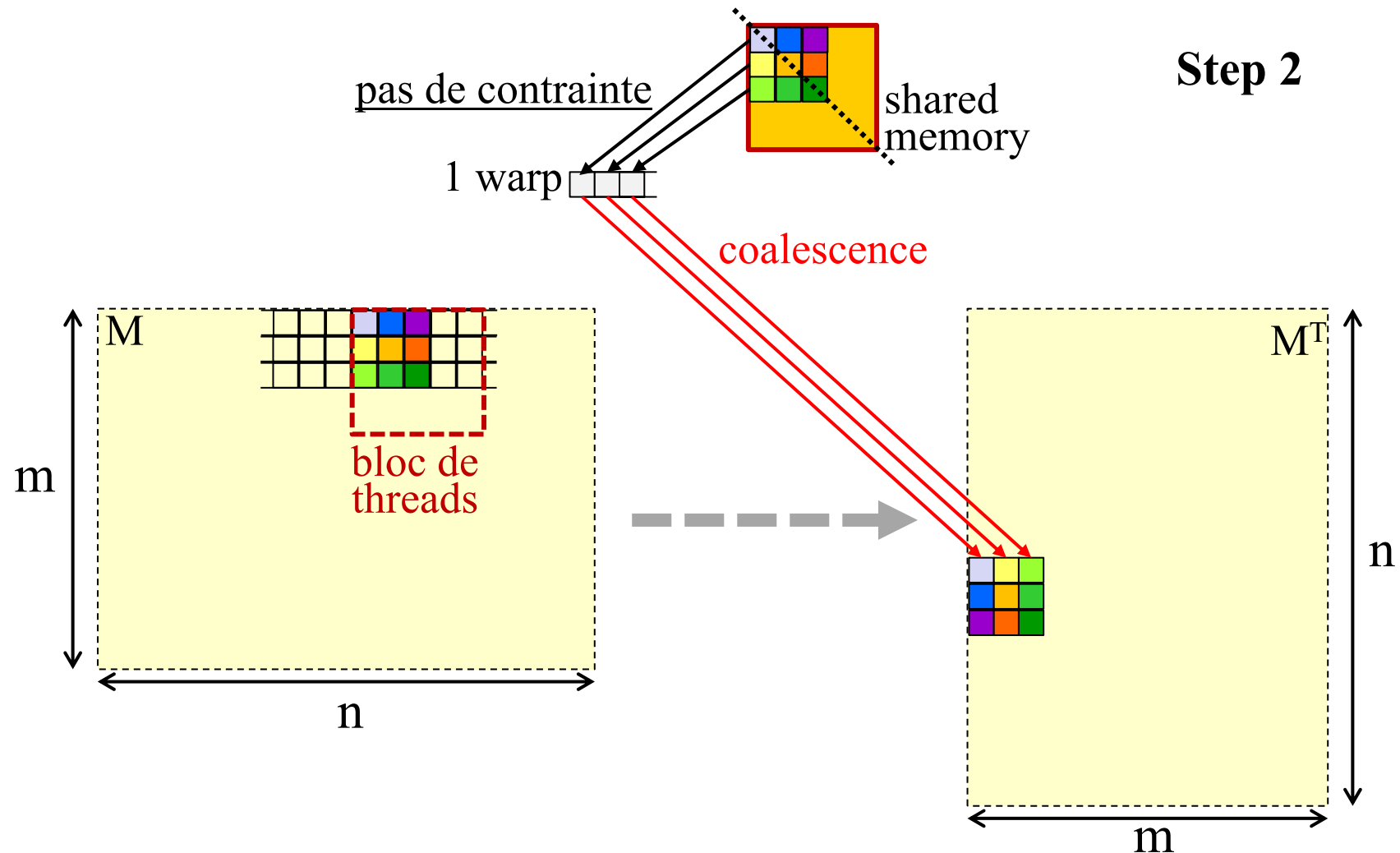
Kernel optimisé de transposition d'une matrice

Step 1



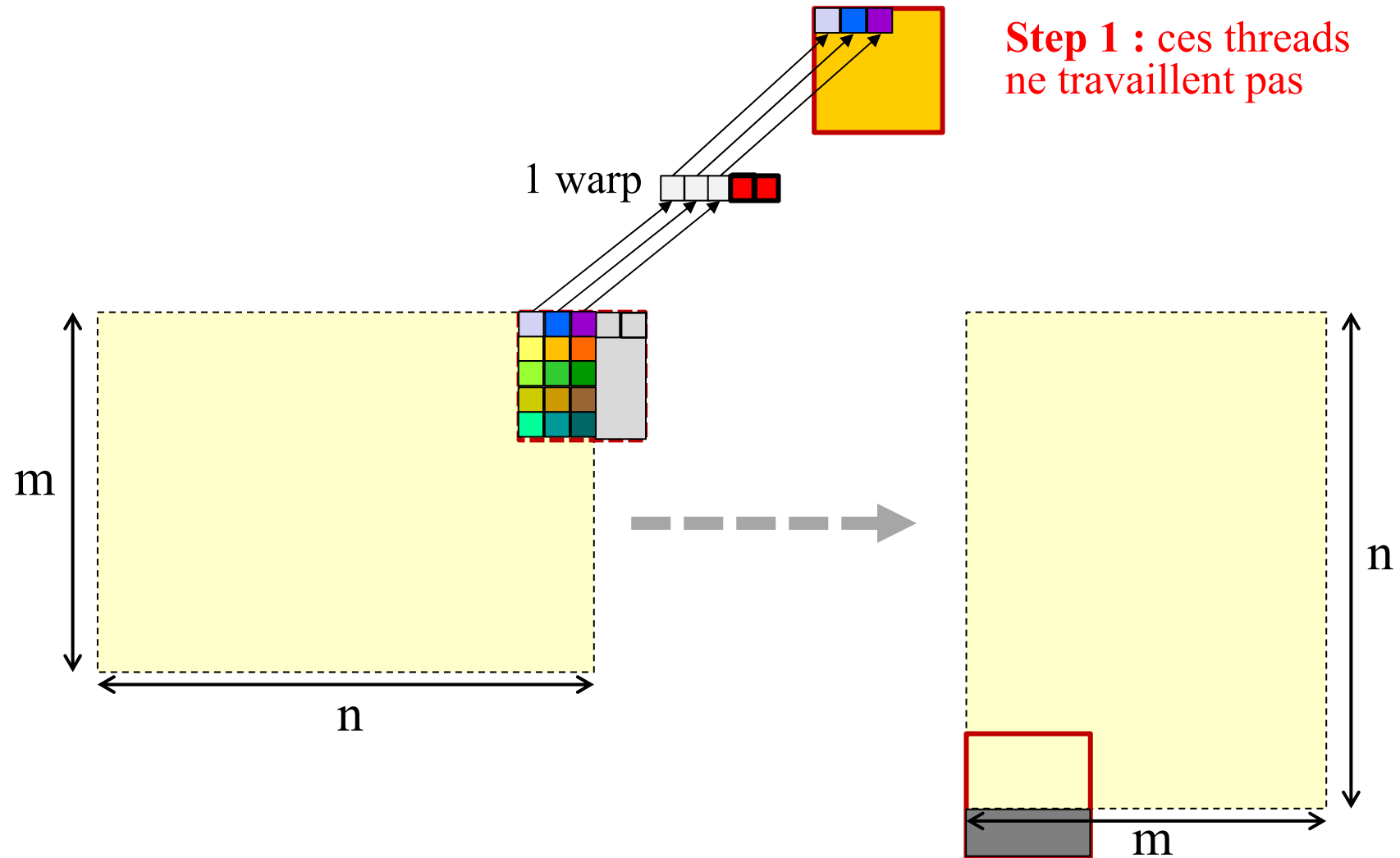
Ex 3: Matrix transposition & juxtaposed blocks

Kernel optimisé de transposition d'une matrice



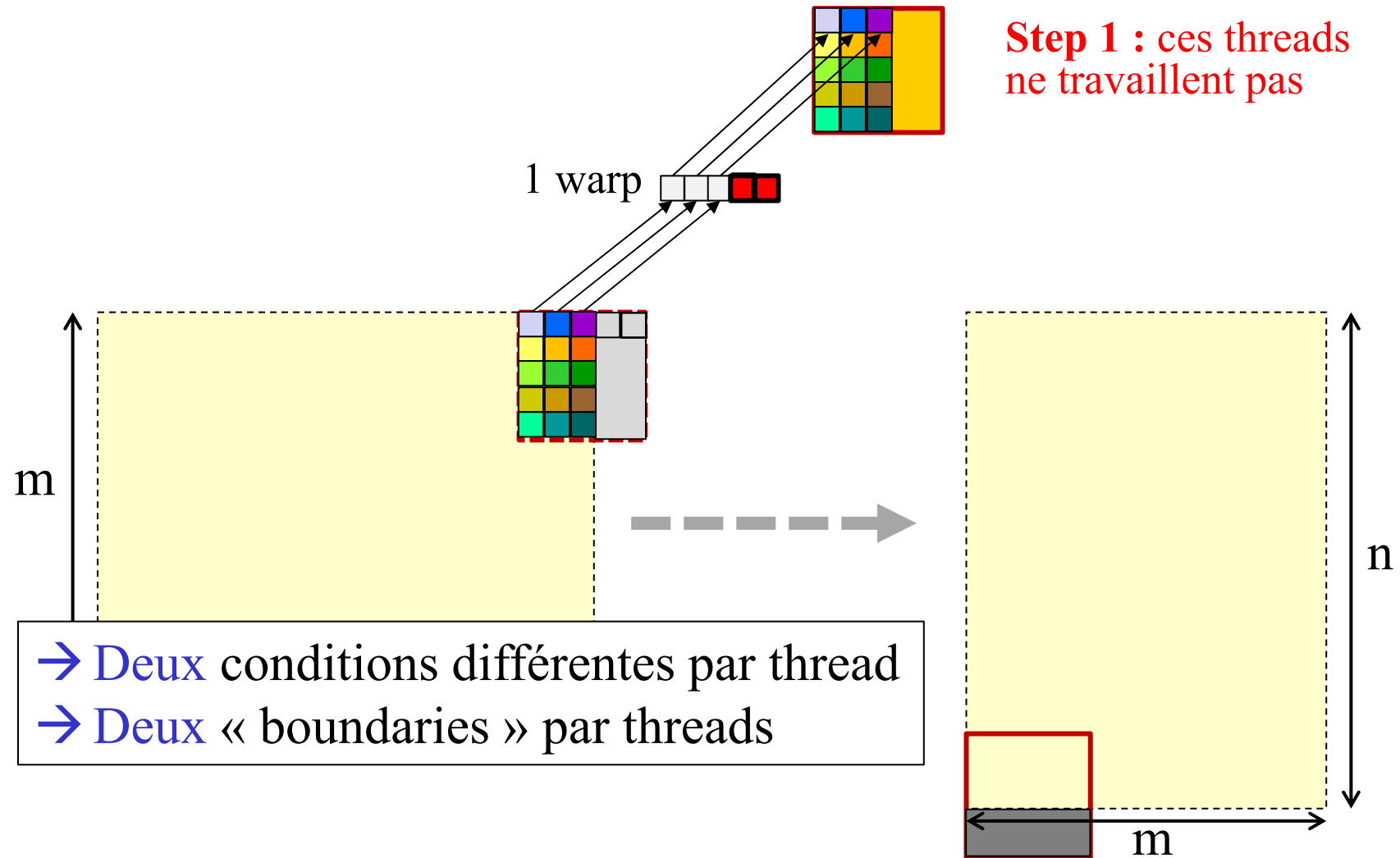
Ex 3: Matrix transposition & juxtaposed blocks

Kernel optimisé de transposition d'une matrice



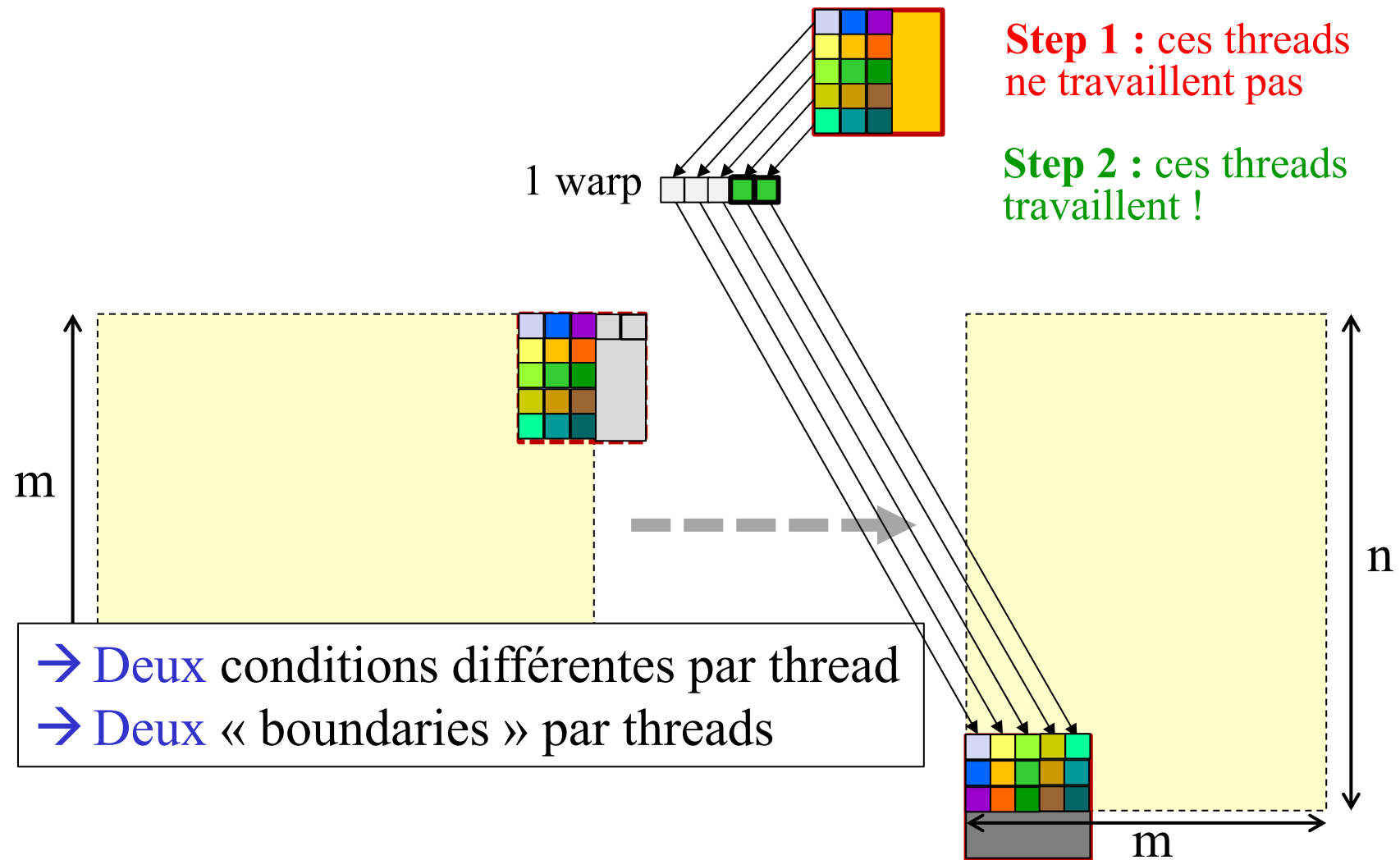
Ex 3: Matrix transposition & juxtaposed blocks

Kernel optimisé de transposition d'une matrice



Ex 3: Matrix transposition & juxtaposed blocks

Kernel optimisé de transposition d'une matrice



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];
}

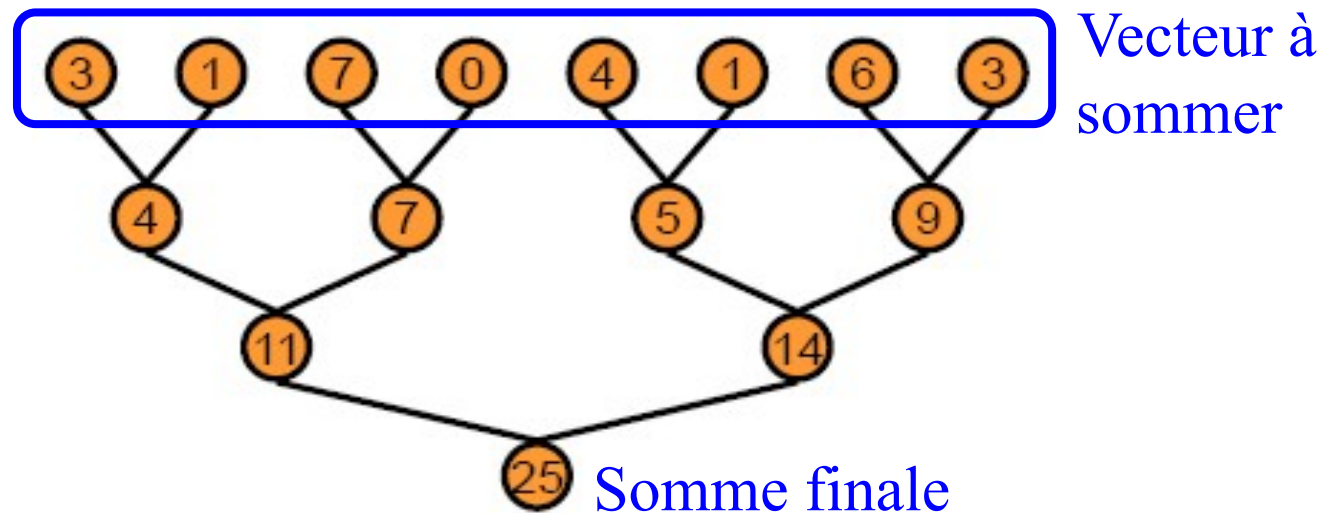
```

CUDA programming with Shared Memory

1. Principles of the *Shared Memory*
2. Algorithm and code examples
3. **Optimized reduction**
 - **Reduction algorithm with coalescence**
 - Code with coalescence & limited divergence
 - Optimized code with *Shared Memory*

Reduction algorithm with coalescence

Schéma de base :



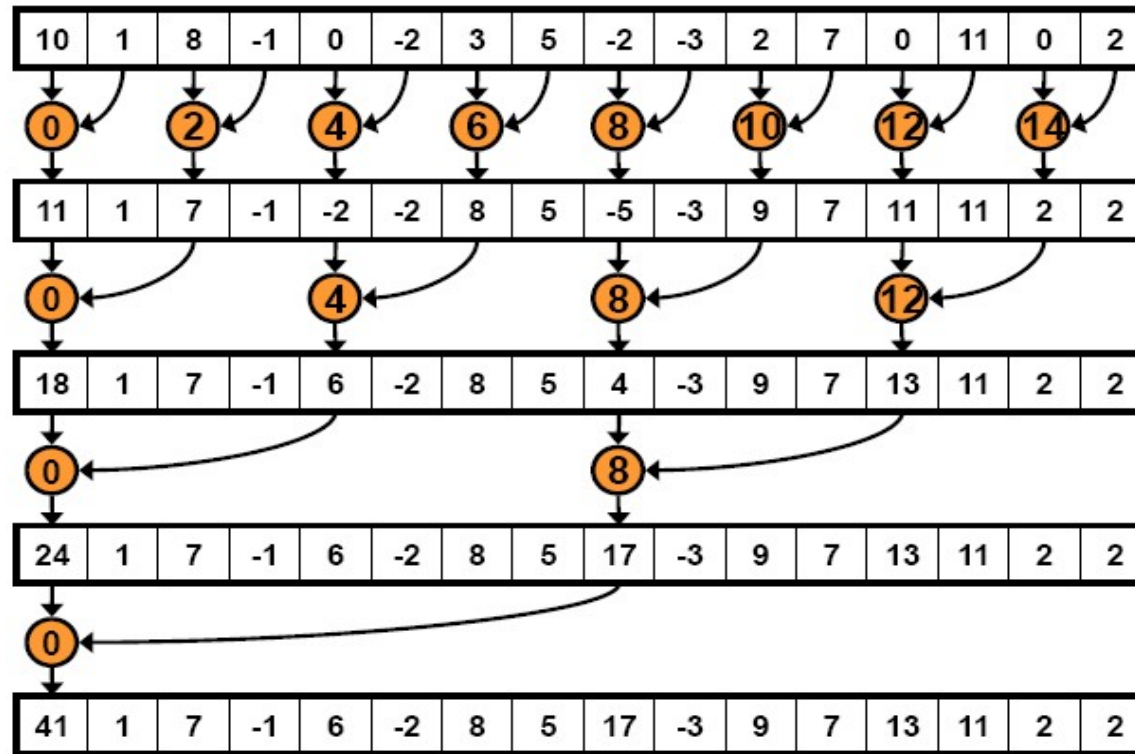
Une « réduction » contient du parallélisme difficile à exploiter :

- plus les calculs progressent et moins il y a de parallélisme,
- il y a bcp d'accès aux données et peu de calculs,
- et risque de *divergence* et de *non-coalescence*

Voir : *Optimizing Parallel Reduction in CUDA*, Mark Harris (NVIDIA)

Reduction algorithm with coalescence

Thread Id



Forte divergence, et pas de coalescence.
Très mauvaise stratégie sur GPU !

Données à réduire de + en + dispersées

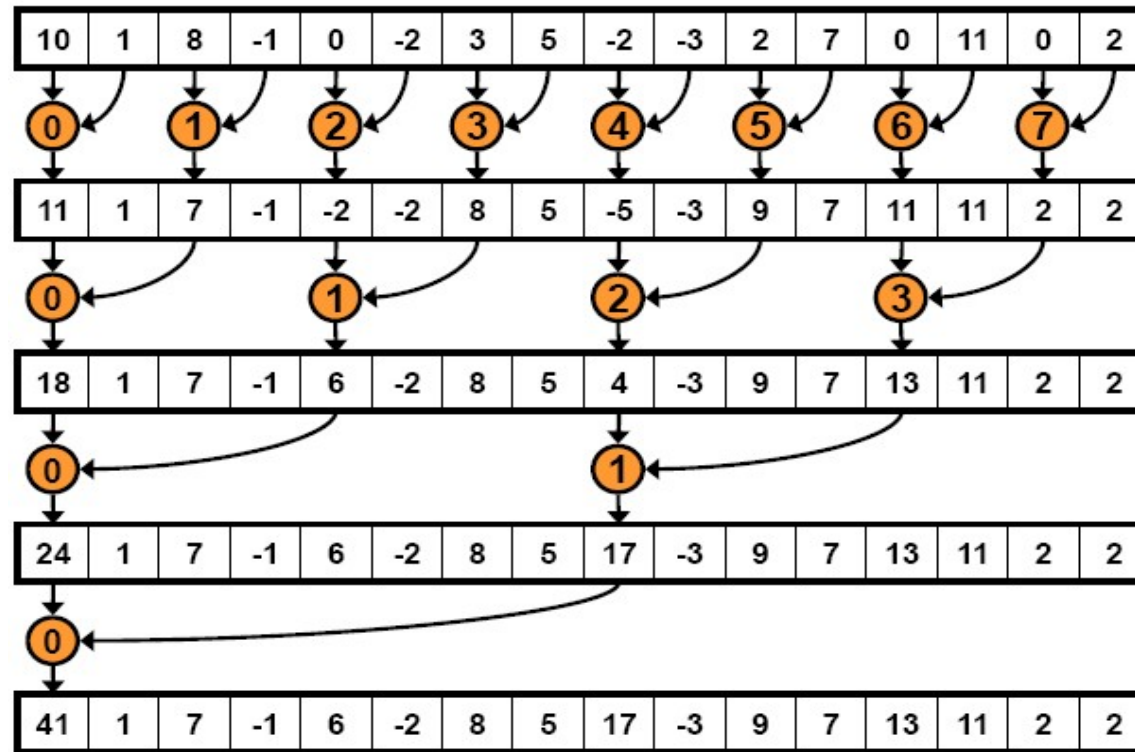
→ Accès mémoire de moins en moins « coalescents » !

Thread actifs de + en + dispersés

→ Activations de « warps » très pauvres en threads actifs

Reduction algorithm with coalescence

Thread Id



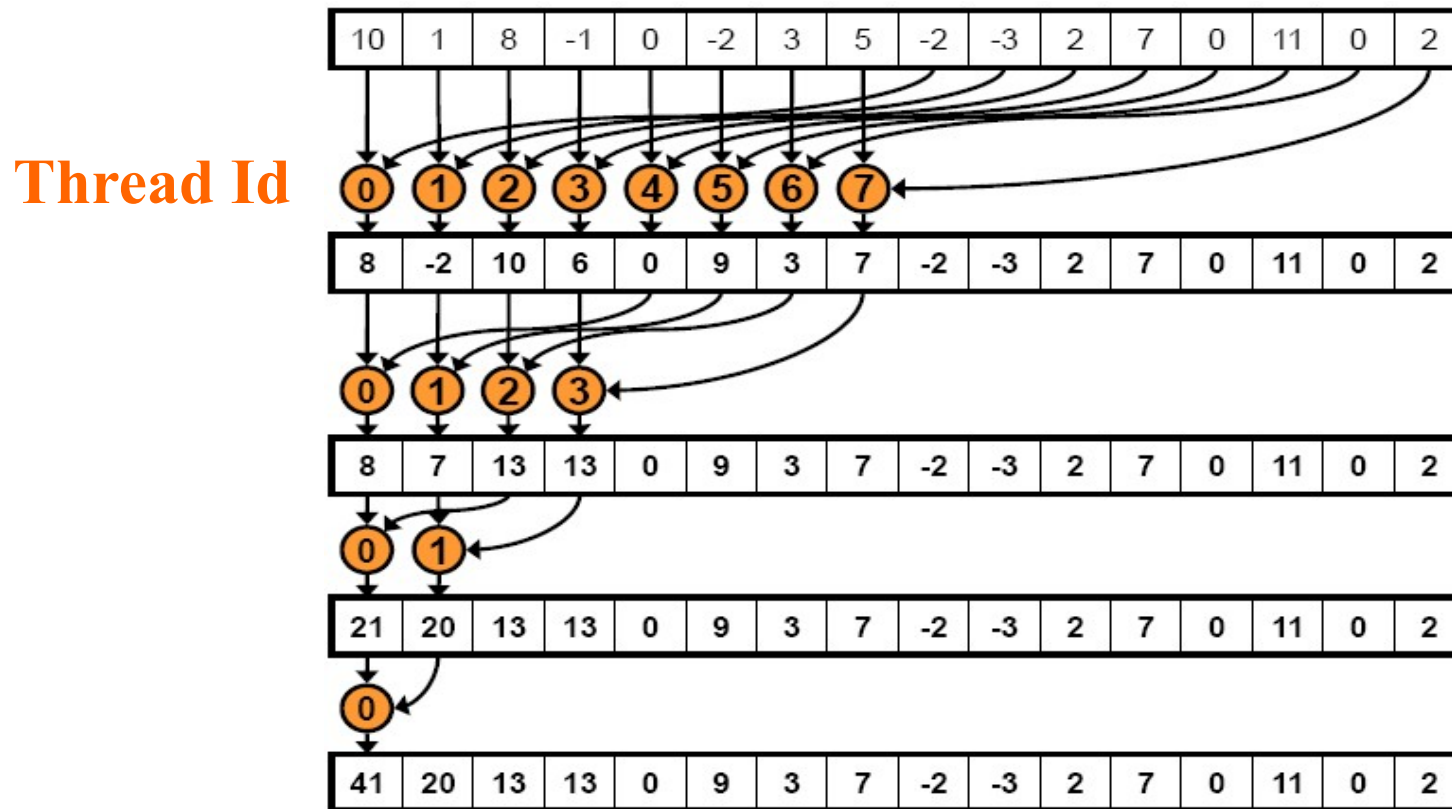
Divergence
maîtrisée,
mais **pas de
coalescence**.
Mauvaise
stratégie sur
GPU !

Sous-ensembles de threads actifs « contiguës depuis le thread 0 ».

Mais données à réduire de + en + dispersées

→ Accès mémoire toujours de moins en moins « coalescents » !

Reduction algorithm with coalescence



Divergence
maîtrisée,
et accès
coalescents.

**Bonne
stratégie sur
GPU !**

Sous-ensembles de threads actifs « contiguës depuis le thread 0 ».

Accès mémoires qui restent coalescents.

→ Stratégie efficace sur GPU **comment l'implanter ?**

CUDA programming with Shared Memory

1. Principles of the *Shared Memory*
2. Algorithm and code examples
3. **Optimized reduction**
 - Reduction algorithm with coalescence
 - **Code with coalescence & limited divergence**
 - Optimized code with *Shared Memory*

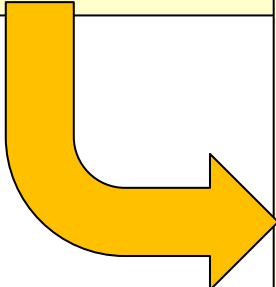
Code with coalescence & limited divergence

On garde actif un sous-ensemble $[0;n]$ de threads

```
int idx = ...;
// 1ère partie du kernel: tous les th actifs
A[idx] = ...

// 2nd partie du kernel: la moitié des th actifs
if (idx%2 == 0) {
    A[idx] = A[idx] + A[idx+1];
}
if (idx%4 == 0) { // Puis le quart des th actifs
    .....
}
...
```

Mauvais



```
int idx = ...;
// 1ère partie du kernel: tous les th actifs
A[idx] = ...

// 2nd partie du kernel: la moitié des th actifs
if (threadIdx.x < BLOCKSIZE_X/2) {
    A[idx] = A[idx] + A[idx + BLOCKSIZE_X/2];
}
if (threadIdx.x < BLOCKSIZE_X/4) .....
.....
```

Code with coalescence & limited divergence

On peut même terminer explicitement les threads inutiles

```
int idx = ...;
// 1ère partie du kernel: tous les th actifs
A[idx] = ...

// 2nd partie du kernel: la moitié des th actifs
if (threadIdx.x < BLOCKSIZE_X/2) {
    A[idx] = A[idx] + A[idx + BLOCKSIZE_X/2];
} else {
    return;
}
// Puis le quart des th actifs
if (threadIdx.x < BLOCKSIZE_X/4) {
    A[idx] = A[idx] + A[idx + BLOCKSIZE_X/4];
} else {
    return;
}
.....
```

- De moins en moins de « warps » activés
- Des accès mémoire coalescents (plus rapides)

CUDA programming with Shared Memory

1. Principles of the *Shared Memory*
2. Algorithm and code examples
3. **Optimized reduction**
 - Reduction algorithm with coalescence
 - Code with coalescence & limited divergence
 - **Optimized code with *Shared Memory***

Optimized code with *Shared Memory*

```
__global__ void Reduce_kernel(float gtab[N], int l, float *AdrGRes)
{
    __shared__ float buff[BLOCK_SIZE]; // BLOCK_SIZE must be a power of 2
    int useful = BLOCK_SIZE;           // Nb of useful threads
    int idx = threadIdx.x + blockIdx.x*BLOCK_SIZE;

    // Coalescent global memory reading (all threads are active)
    if (idx < N)
        buff[threadIdx.x] = gtab[idx]; // load global data
    else
        buff[threadIdx.x] = 0.0;        // padding when necessary
    __syncthreads();                   // Required synchronization barrier

    // Reduction loop
    useful >>= 1;                       // u = u/2 : Only half of threads are now useful
    while (useful > 0) {
        if (threadIdx.x < useful) // Useful threads reduce data
            buff[threadIdx.x] += buff[threadIdx.x + useful];
        else
            return;               // Useless threads terminate
        useful >>= 1;             // Half of threads won't be useful at the next iter
        __syncthreads();          // Required synchronization barrier
    }

    // Accumulation in global memory by th 0 of the block
    atomicAdd(AdrGRes, buff[0]); // expensive op: not the only solution
}
```

Very efficient solution (today)

Optimized code with *Shared Memory*

```
__global__ void Reduce_kernel(float gtab[N], int l, float *AdrGRes)
{
    __shared__ float buff[BLOCK_SIZE]; // BLOCK_SIZE must be a power of 2
    int useful = BLOCK_SIZE;           // Nb of useful threads
    int idx = threadIdx.x + blockIdx.x*BLOCK_SIZE;

    // Coalescent global memory reading (all threads are active)
    if (idx < N)
        buff[threadIdx.x] = gtab[idx]; // load global data
    else
        buff[threadIdx.x] = 0.0;        // padding when necessary

    // Reduction loop
    useful >>= 1; // u = u/2 : Only half of threads are now useful
    while (useful > 0) {
        syncthreads(); // Required synchronization barrier
        if (threadIdx.x < useful) // Useful threads reduce data
            buff[threadIdx.x] += buff[threadIdx.x + useful];
        else
            return; // Useless threads terminate
        useful >>= 1; // Half of threads won't be useful at next iter
    }

    // Accumulation in global memory by th 0 of the block
    atomicAdd(AdrGRes, buff[0]); // expensive op: not the only solution
}
```

Avec 1 barrière
de synchro. de
moins 😊

CUDA programming with Shared Memory

End