



Mineure CalHau2

CUDA : optimized programming

Stéphane Vialle



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

1

CUDA : optimized programming

1 – Utilisation de la *shared memory*

- Principes de la *shared memory*
- Ex 1: Moyenne glissante & blocs juxtaposés
- First generic scheme of a shM 2D kernel
- Ex 2: Moyenne glissante & blocs recouvrant
- Ex 3: Transposition de matrice
- Second generic scheme of a ShM 2D kernel

2 – Réduction optimisée

3 – Kernels auto-adaptatifs

4 – Parallélisme dynamique sur GPU

5 – Bilan de la programmation CUDA

2

Principe de la *shared memory*

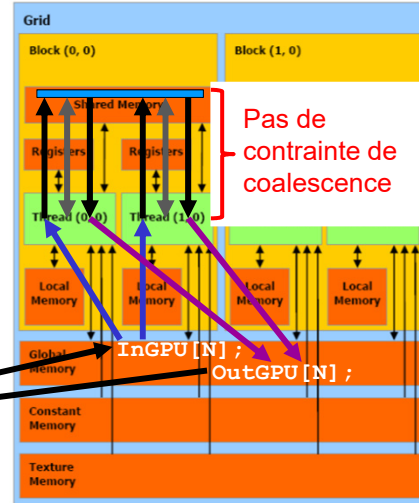
Avantages des kernels utilisant la mémoire globale et la mémoire *shared* :

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 global d'accès à la mémoire

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

CPU



3

Principe de la *shared memory*

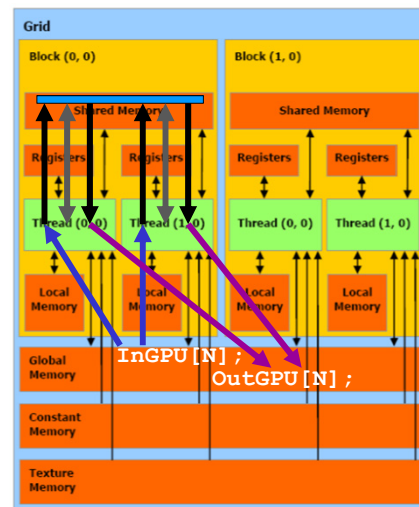
Avantages des kernels utilisant la mémoire globale et la mémoire *shared* :

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 global d'accès à la mémoire

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**



4

Principe de la *shared memory*

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

Calcul : `res[j] = 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
« fait le cache » !

Les blocs sont juxtaposés



5

CUDA : optimized programming

1 – Utilisation de la *shared memory*

- Principes de la *shared memory*
- **Ex 1: Moyenne glissante & blocs juxtaposés**
- First generic scheme of a shM 2D kernel
- Ex 2: Moyenne glissante & blocs recouvrant
- Ex 3: Transposition de matrice
- Second generic scheme of a ShM 2D kernel

2 – Réduction optimisée

3 – Kernels auto-adaptatifs

4 – Parallélisme dynamique sur GPU

5 – Bilan de la programmation CUDA

6

Ex 1: Moyenne glissante & blocs juxtaposés

Kernel utilisant la mémoire *shared* 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

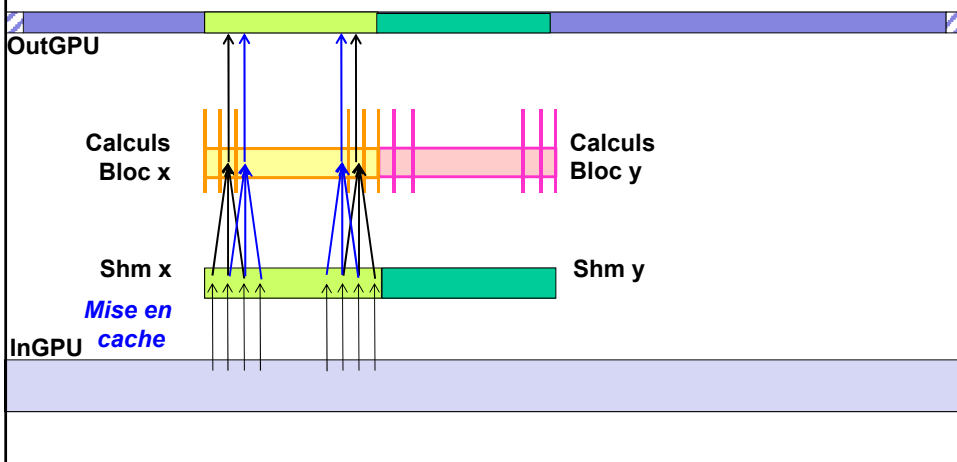


7

Ex 1: Moyenne glissante & blocs juxtaposés

Kernel utilisant la mémoire *shared* 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 »



8

Utilisation de la *shared memory*

Ex 1: Moyenne glissante & blocs juxtaposés

Kernel utilisant la mémoire *shared* 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 »

The diagram illustrates the data flow for two adjacent blocks, 'Bloc x' and 'Bloc y'. At the bottom, the 'InGPU' contains input data. For 'Bloc x', data is loaded into 'Shm x' (green bar) and then accessed by 'Calculs Bloc x' (yellow bar). For 'Bloc y', data is loaded into 'Shm y' (green bar) and then accessed by 'Calculs Bloc y' (pink bar). The 'OutGPU' at the top shows the resulting output. Blue arrows indicate the path from InGPU to Shm and then to Calculs. Red arrows show the path from InGPU to Calculs. Vertical arrows point from the Shm bars to the Calculs bars, indicating shared memory access.

9

Utilisation de la *shared memory*

Ex 1: Moyenne glissante & blocs juxtaposés

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

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

Hyp : $N_d = k * \text{BLOCK_SIZE_X}$

```

... ..
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

On exploite les données en *shm*

Les blocs sont juxtaposés

The diagram at the bottom shows two adjacent blocks in memory, represented by blue bars with dashed red outlines, indicating they are 'juxtaposés'.

10

Utilisation de la *shared memory*

Ex 1: Moyenne glissante & blocs juxtaposés

Kernel utilisant la mémoire *shared* 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}
__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

11

Utilisation de la *shared memory*

Ex 1: Moyenne glissante & blocs juxtaposés

Kernel utilisant la mémoire *shared* 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

12

CUDA : optimized programming

1 – Utilisation de la *shared memory*

- Principes de la *shared memory*
- Ex 1: Moyenne glissante & blocs juxtaposés
- **First generic scheme of a shM 2D kernel**
- Ex 2: Moyenne glissante & blocs recouvrant
- Ex 3: Transposition de matrice
- Second generic scheme of a ShM 2D kernel

2 – Réduction optimisée

3 – Kernels auto-adaptatifs

4 – Parallélisme dynamique sur GPU

5 – Bilan de la programmation CUDA

13

First generic scheme of a 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 some boundaries
    // on indexes in global memory
    if (...) { Ensure coalescence
        shdata[threadIdx.y][threadIdx.x] = Input[idxY][idxX];
    }
    __syncthreads(); // REQUIRED: wait for all data loaded in ShM
    // Computations using any data into the shared memory,
    // respecting others boundaries
    if (...) { No constraint
        res = ... shdata[...] [...] ... ;
        Output[idxY][idxX] = res;
    } Ensure coalescence
}

```

14

CUDA : optimized programming

1 – Utilisation de la *shared memory*

- Principes de la *shared memory*
- Ex 1: Moyenne glissante & blocs juxtaposés
- First generic scheme of a shM 2D kernel
- **Ex 2: Moyenne glissante & blocs recouvrant**
- Ex 3: Transposition de matrice
- Second generic scheme of a ShM 2D kernel

2 – Réduction optimisée

3 – Kernels auto-adaptatifs

4 – Parallélisme dynamique sur GPU

5 – Bilan de la programmation CUDA

15

Ex 2: Moyenne glissante & blocs recouvrant

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

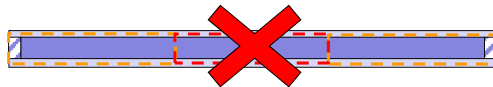
Objectif : **toutes** les données cachées en *shared memory*.

→ pouvoir écrire le code suivant :

```

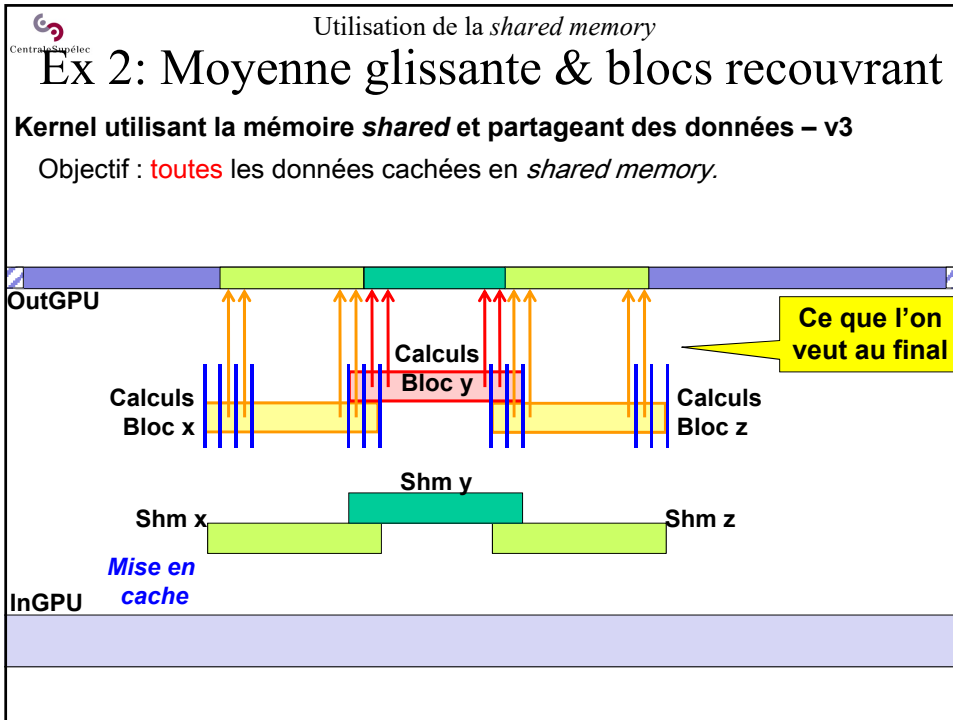
__global__ void k1D(void)
{
    .....
    if (...) {
        // Compute result (another computation example...)
        res = shdata[threadIdx.x-1]*0.25f +
              shdata[threadIdx.x]*0.50f   +
              shdata[threadIdx.x+1]*0.25f;
        // Write result in the global memory
        OutGPU[idx]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

```

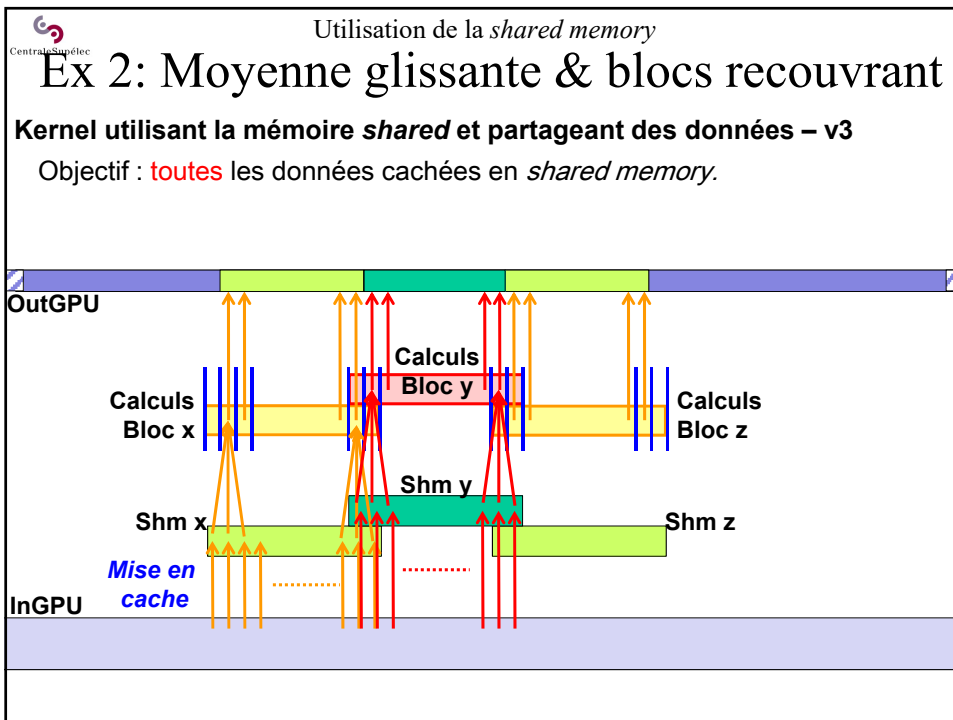


→ Des blocs juxtaposés ne suffisent plus

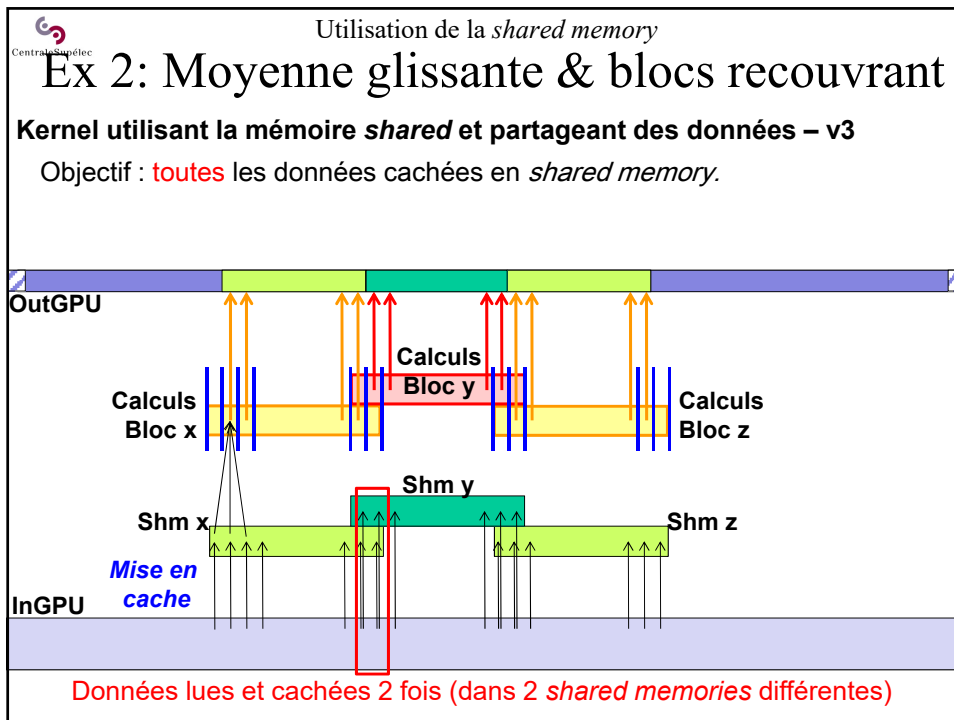
16



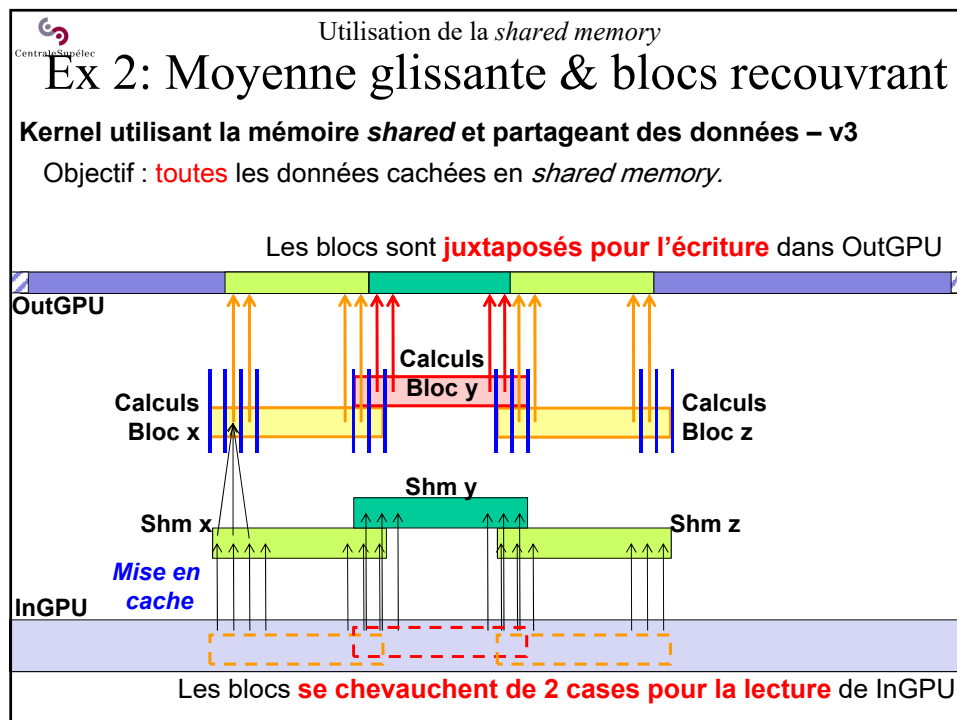
17



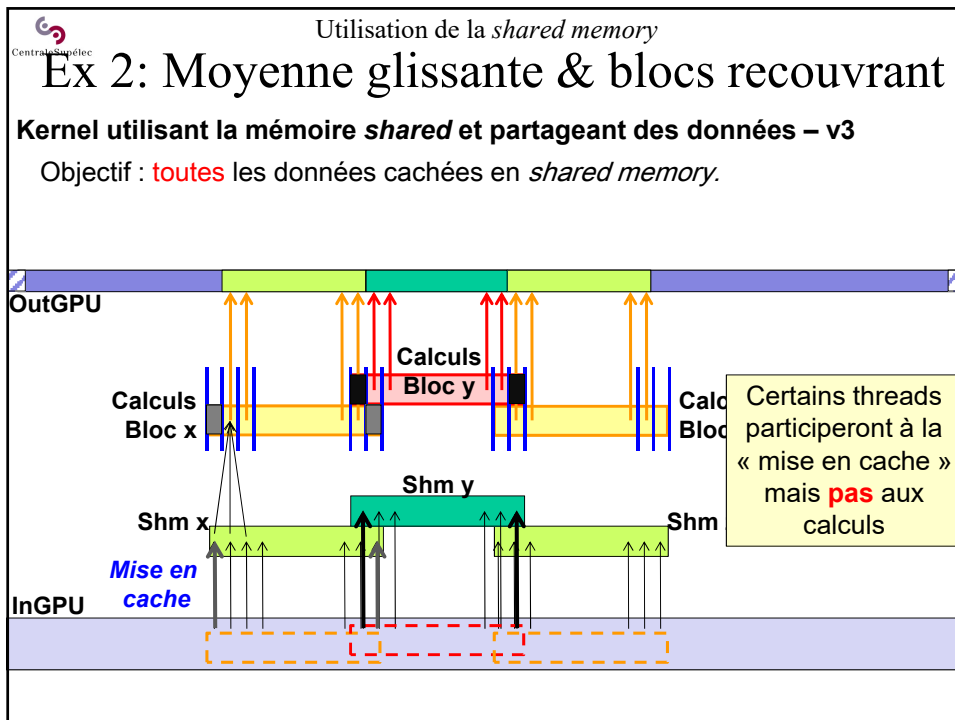
18



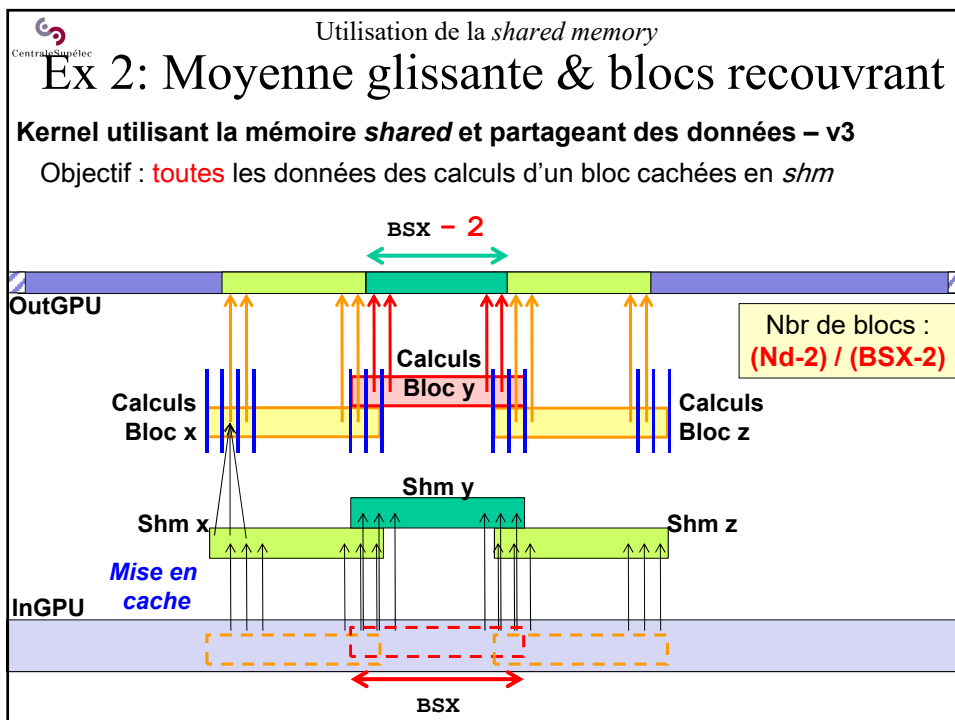
19



20



21



22

Utilisation de la *shared memory*

Ex 2: Moyenne glissante & blocs recouvrant

Kernel utilisant la mémoire *shared* 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 : $Nd-2 = k*(BSX-2)$

```

global__ void k1D(void)
{
    // Collective definition of table in the shared memory
    shared float shdata[BSX];
    // Local computation result
    float res;
    // 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 !!
    if (threadIdx.x > 0 && threadIdx.x < BSX-1
        /* && idx > 0 && idx < Nd-1 */) {
        // 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;
    }
}

```

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

Les blocs doivent se chevaucher

23

Utilisation de la *shared memory*

Ex 2: Moyenne glissante & blocs recouvrant

Kernel utilisant la mémoire *shared* 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 f1(void)
{
    // Collective definition of table
    shared float shdata[BSX];
    // Local computation result
    float res;
    // 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 !!
    if (threadIdx.x > 0 && threadIdx.x < BSX-1
        /* && idx > 0 */ && idx < Nd-1) {
        // 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;
    }
}

```

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

Les blocs doivent se chevaucher, et le dernier déborde

24

CUDA : optimized programming

1 – Utilisation de la *shared memory*

- Principes de la *shared memory*
- Ex 1: Moyenne glissante & blocs juxtaposés
- First generic scheme of a shM 2D kernel
- Ex 2: Moyenne glissante & blocs recouvrant
- **Ex 3: Transposition de matrice**
- Second generic scheme of a ShM 2D kernel

2 – Réduction optimisée

3 – Kernels auto-adaptatifs

4 – Parallélisme dynamique sur GPU

5 – Bilan de la programmation CUDA

25

Ex 3: Transposition de matrice

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];
}
```

Accès NON coalescent

Accès coalescent

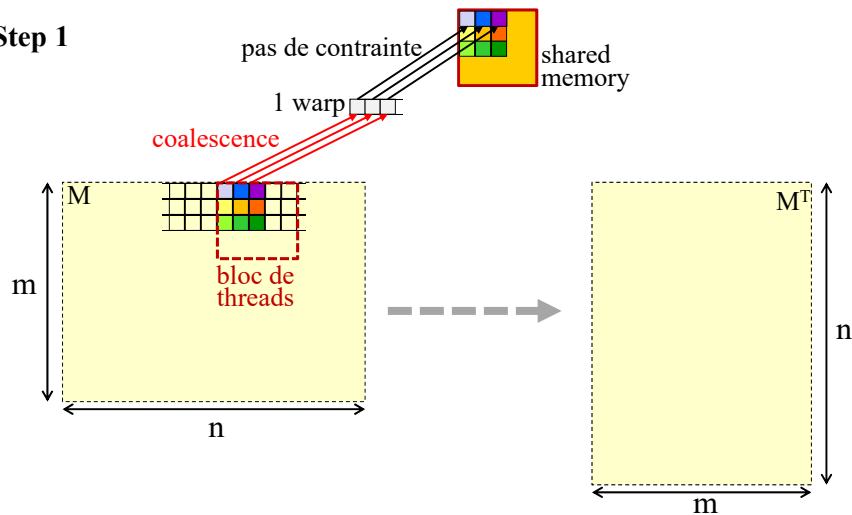
→ Utiliser la *shared memory* pour être coalescent à la lecture et à l'écriture...

26

Ex 3: Transposition de matrice

Kernel optimisé de transposition d'une matrice

Step 1

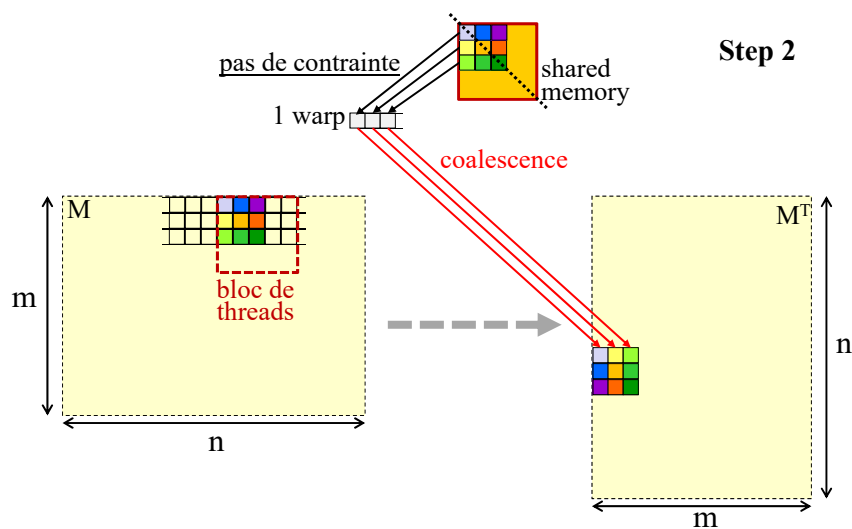


27

Ex 3: Transposition de matrice

Kernel optimisé de transposition d'une matrice

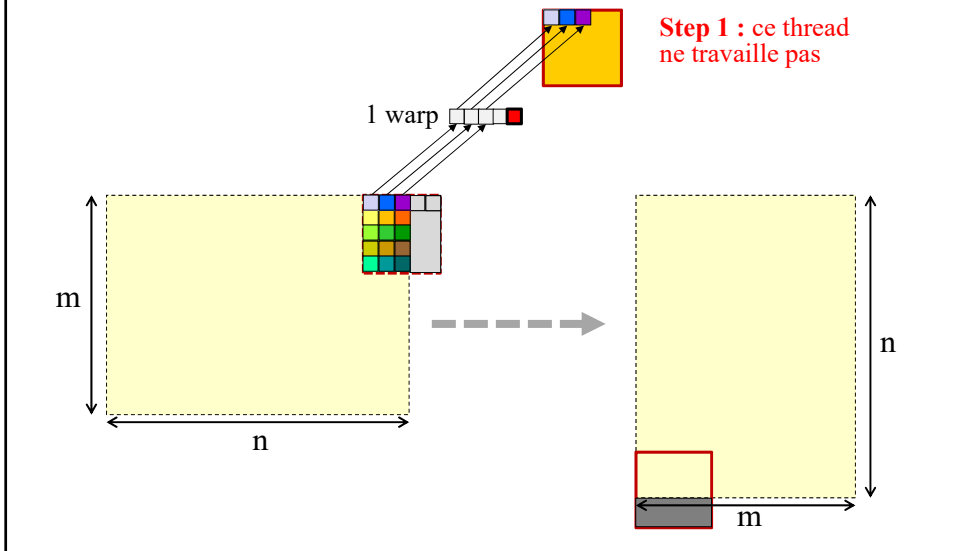
Step 2



28

Ex 3: Transposition de matrice

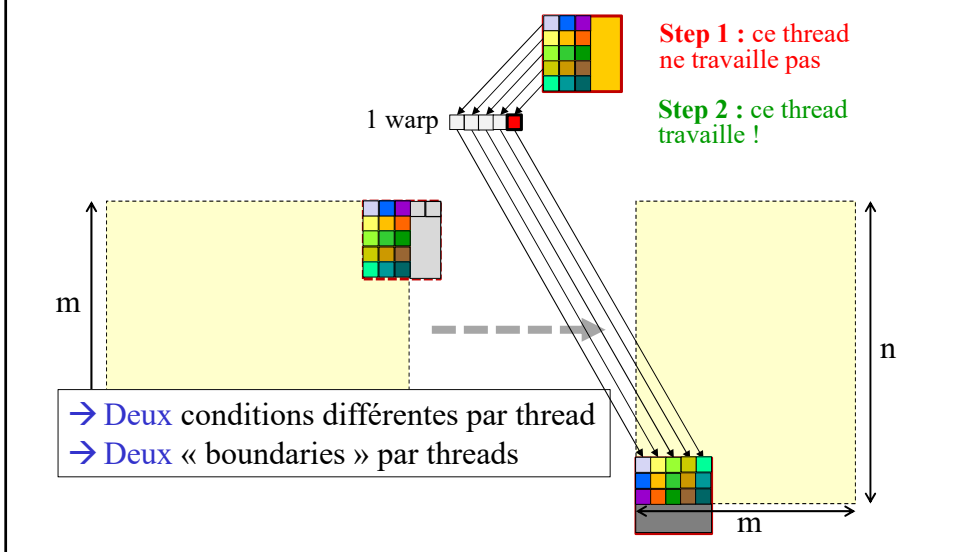
Kernel optimisé de transposition d'une matrice



29

Ex 3: Transposition de matrice

Kernel optimisé de transposition d'une matrice



30

Ex 3: Transposition de matrice

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];
}

```

31

Second generic scheme of a 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);

for (int step = 0; step < ...; step++) {
// shared memory update
...
__syncthreads();
// local computation
res += ...
__syncthreads();
}

if (...) {
Output[idxY][idxX] = res;
}
}

```

32

Second generic scheme of a ShM 2D kernel

```

__global__ void k2D(void)
{
    .....

    for (int step = 0; step < ...; step++) {
        // Loading input data into the ShM,
        if (...) {
            shdata[threadIdx.y][threadIdx.x] =
                Input[gg(idyY,step)][ff(idyX,step)];
        }
        __syncthreads(); // REQUIRED: wait for ShM has been updated
        // Computations using any data into the shared memory,
        if (...) {
            res += ... shdata[...] [...] ... ;
        }
        __syncthreads(); // REQUIRED: wait for all ShM has been read
    }

    .....
    }
    
```

33

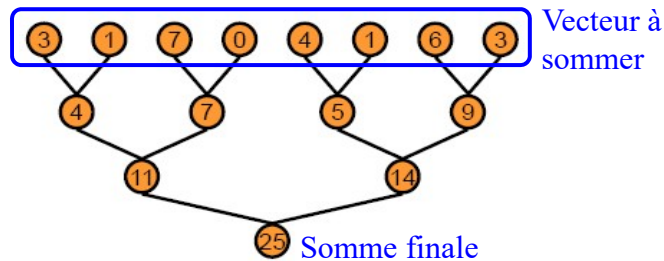
CUDA : optimized programming

- 1 – Utilisation de la *shared memory*
- 2 – **Réduction optimisée**
 - **Optimisation du schéma de réduction**
 - Implantation coalescente et peu divergente
 - Implantation en *shared memory*
- 3 – Kernels auto-adaptatifs
- 4 – Parallélisme dynamique sur GPU
- 5 – Bilan de la programmation CUDA

34

Optimisation du schéma de réduction

Schéma de base :



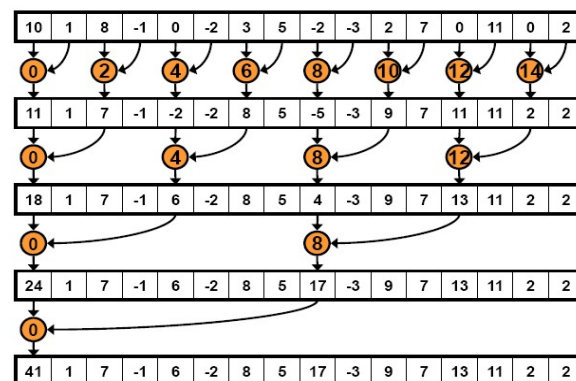
Une « réduction » contient du parallélisme difficile à exploiter :

- 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)

Optimisation du schéma de réduction

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 + dispersées

→ Activations de « warps » très pauvres en threads actifs

CentraleSupélec

Réduction optimisée

Optimisation du schéma de réduction

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 » !

37

CentraleSupélec

Réduction optimisée

Optimisation du schéma de réduction

Thread Id

Pas de divergence 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 ?**

38

CUDA : optimized programming

- 1 – Utilisation de la *shared memory*
- 2 – Réduction optimisée
 - Optimisation du schéma de réduction
 - **Implantation coalescente et peu divergente**
 - Implantation en *shared memory*
- 3 – Kernels auto-adaptatifs
- 4 – Parallélisme dynamique sur GPU
- 5 – Bilan de la programmation CUDA

39

Réduction optimisée

Implantation coalescente et peu divergente

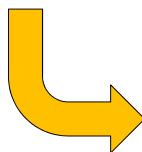
On garde actif un sous-ensemble [0;n] de threads

```
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) .....
.....

```

40

Implantation coalescente et peu divergente

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;
}
.....
```

- Moins de « warps » activés en 2nd partie de kernels
- Moins d'accès en mémoire globale (hyp : accès coalescents par warp)

41

CUDA : optimized programming

- 1 – Utilisation de la *shared memory*
- 2 – **Réduction optimisée**
 - Optimisation du schéma de réduction
 - Implantation coalescente et peu divergente
 - **Implantation en *shared memory***
- 3 – Kernels auto-adaptatifs
- 4 – Parallélisme dynamique sur GPU
- 5 – Bilan de la programmation CUDA

42

Implantation en *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; // 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
}

```

43

Implantation en *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; // 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 😊

44

CUDA : optimized programming

- 1 – Utilisation de la *shared memory*
- 2 – Réduction optimisée
- 3 – Déroulement de boucle auto-adaptatif**
 - Auto-adaptation à la compilation
 - Optimisation SIMD de la boucle déroulée
 - Implantation en template C++
- 4 – Parallélisme dynamique sur GPU
- 5 – Bilan de la programmation CUDA

45

Auto-adaptation à la compilation

Principe :

- Implanter un kernel sans limite de taille (générique) :
BLOCK_SIZE_X = 1, 2, 4, 8, ...512, 1024
- Mais ne compiler que les parties correspondant à sa taille
→ Compiler le strict minimum d'instruction à exécuter

Solution :

- Dérouler la boucle de réduction
- Éliminer à la compilation les étapes inutiles

46

Auto-adaptation à la compilation

```

__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 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 (coalescent)
    else
        buff[threadIdx.x] = 0.0; // padding when necessary

    // Reduction loop
    #if BLOCK_SIZE > 512
        __syncthreads(); // Barrière de synchro NECESSAIRE
        if (threadIdx.x < 512) // Useful threads reduce data
            buff[threadIdx.x] += buff[threadIdx.x + 512];
        else
            return; // Useless threads terminate
    #endif

    #if BLOCK_SIZE > 256
        __syncthreads(); // Barrière de synchro NECESSAIRE
        if (threadIdx.x < 256) // Useful threads reduce data
            buff[threadIdx.x] += buff[threadIdx.x + 256];
        else
            return; // Useless threads terminate
    #endif
}

```

47

Auto-adaptation à la compilation

```

#if BLOCK_SIZE > 128
    __syncthreads(); // Barrière de synchro NECESSAIRE
    if (threadIdx.x < 128) // Useful threads reduce data
        buff[threadIdx.x] += buff[threadIdx.x + 128];
    else
        return; // Useless threads terminate
#endif

#if BLOCK_SIZE > 64
    __syncthreads(); // Barrière de synchro NECESSAIRE
    if (threadIdx.x < 64) // Useful threads reduce data
        buff[threadIdx.x] += buff[threadIdx.x + 64];
    else
        return; // Useless threads terminate
#endif

#if BLOCK_SIZE > 32
    __syncthreads(); // Barrière de synchro NECESSAIRE
    if (threadIdx.x < 32) // Useful threads reduce data
        buff[threadIdx.x] += buff[threadIdx.x + 32];
    else
        return; // Useless threads terminate
#endif

```

48

Auto-adaptation à la compilation

```

#if BLOCK_SIZE > 16
    __syncthreads();           // Barrière de synchro NECESSAIRE
    if (threadIdx.x < 16)     // Useful threads reduce data
        buff[threadIdx.x] += buff[threadIdx.x + 16];
    else
        return;              // Useless threads terminate
#endif

#if BLOCK_SIZE > 8
    __syncthreads();           // Barrière de synchro NECESSAIRE
    if (threadIdx.x < 8)     // Useful threads reduce data
        buff[threadIdx.x] += buff[threadIdx.x + 8];
    else
        return;              // Useless threads terminate
#endif

.....

#if BLOCK_SIZE > 1
    __syncthreads();           // Barrière de synchro NECESSAIRE
    if (threadIdx.x < 1)     // Useful threads reduce data
        buff[threadIdx.x] += buff[threadIdx.x + 1];
    else
        return;              // Useless threads terminate
#endif

// Accumulation in global memory by th 0 (the only survivor !)
atomicAdd(AdrGRes,buff[0]);
}

```

49

CUDA : optimized programming

- 1 – Utilisation de la *shared memory*
- 2 – Réduction optimisée
- 3 – Déroutement de boucle auto-adaptatif**
 - Auto-adaptation à la compilation
 - **Optimisation SIMD de la boucle déroulée**
 - Implantation en template C++
- 4 – Parallélisme dynamique sur GPU
- 5 – Bilan de la programmation CUDA

50

Optimisation SIMD

Principe :

- Profiter des propriétés SIMD des *warps* lorsqu'il ne reste plus qu'un *warp* actif dans le bloc
- On peut alors supprimer les opérations de synchronisation entre threads (`__syncthreads()`) !

Solution :

- Simplifier le code quand le nombre de threads actifs devient inférieur à 32

51

Optimisation SIMD

```

__global__ void Reduce_kernel(float gtab[N], int l, float *AdrGRes)
{
    __shared__ float buff[BLOCK_SIZE > 64 ? BLOCK_SIZE : 64]; //power of 2
    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 (coalescent)
    else
        buff[threadIdx.x] = 0.0; // padding when necessary

    // Reduction loop
    #if BLOCK_SIZE > 512
        __syncthreads(); // Barrière de synchro NECESSAIRE
        if (threadIdx.x < 512) // Useful threads reduce data
            buff[threadIdx.x] += buff[threadIdx.x + 512];
        else
            return; // Useless threads terminate
    #endif

    #if BLOCK_SIZE > 256
        __syncthreads(); // Barrière de synchro NECESSAIRE
        if (threadIdx.x < 256) // Useful threads reduce data
            buff[threadIdx.x] += buff[threadIdx.x + 256];
        else
            return; // Useless threads terminate
    #endif
    .....
}
    
```

52

Optimisation SIMD

```

#if BLOCK_SIZE > 16
syncthreads(); // Barrière de synchro NECESSAIRE
if (threadIdx.x < 16) // Useful threads reduce data
    buff[threadIdx.x] += buff[threadIdx.x + 16];
else
return; // Useless threads terminate
#endif

#if BLOCK_SIZE > 8
syncthreads(); // Barrière de synchro NECESSAIRE
if (threadIdx.x < 8) // Useful threads reduce data
    buff[threadIdx.x] += buff[threadIdx.x + 8];
else
return; // Useless threads terminate
#endif

.....

#if BLOCK_SIZE > 1
syncthreads(); // Barrière de synchro NECESSAIRE
if (threadIdx.x < 1) // Useful threads reduce data
    buff[threadIdx.x] += buff[threadIdx.x + 1];
else
return; // Useless threads terminate
#endif

// Accumulation in global memory by th0 (warning 32 threads still alive)
if (threadIdx.x == 0) atomicAdd(AdrGRes,buff[0]);
}

```

32 th vivants seulement dans 1 seul warp
→ SIMD pur

Attention, on n'a pas tué les threads [1;31] du warp
→ Ne faire écrire que le dernier

Écriture atomique pour éviter les conflits avec les threads 0 des autres blocs !

53

Optimisation SIMD

Simple ré-écriture sans les lignes supprimées :

```

#if BLOCK_SIZE > 32
    syncthreads(); // Barrière de synchro NECESSAIRE
    if (threadIdx.x < 32) // Useful threads reduce data
        buff[threadIdx.x] += buff[threadIdx.x + 32];
    else
        return; // Useless threads terminate
#endif

#if BLOCK_SIZE > 16
    buff[threadIdx.x] += buff[threadIdx.x + 16];
#endif

#if BLOCK_SIZE > 8
    buff[threadIdx.x] += buff[threadIdx.x + 8];
#endif

.....

#if BLOCK_SIZE > 1
    buff[threadIdx.x] += buff[threadIdx.x + 1];
#endif

// Accumulation in global memory by th0 (warning 32 threads still alive)
if (threadIdx.x == 0) atomicAdd(AdrGRes,buff[0]);
}

```

Les 32 th survivants sont dans 1 seul warp
→ SIMD pur

54

CUDA : optimized programming

- 1 – Utilisation de la *shared memory*
- 2 – Réduction optimisée
- 3 – Déroulement de boucle auto-adaptatif**
 - Auto-adaptation à la compilation
 - Optimisation SIMD de la boucle déroulée
 - **Implantation en template C++**
- 4 – Parallélisme dynamique sur GPU
- 5 – Bilan de la programmation CUDA

55

Implantation en template C++

NVCC est un compilateur C++...

→ On peut se servir du mécanisme des « **templates** » pour spécialiser le code à la compilation

56

Implantation en template C++

```

template <int BLOCK_SIZE>
__global__ void Reduce_kernel(float gtab[N], int l, float *AdrGRes)
{
    __shared__ float buff[BLOCK_SIZE > 64 ? BLOCK_SIZE : 64]; // a power of 2
    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 (coalescent)
    else
        buff[threadIdx.x] = 0.0; // padding when necessary

    // Reduction loop
    if (BLOCK_SIZE > 512) {
        __syncthreads(); // Barrière de synchro NECESSAIRE
        if (idx < 512) // Useful threads reduce data
            buff[threadIdx.x] += buff[threadIdx.x + 512];
        else
            return; // Useless threads terminate
    }
    if (BLOCK_SIZE > 256) {
        __syncthreads(); // Barrière de synchro NECESSAIRE
        if (idx < 256) // Useful threads reduce data
            buff[threadIdx.x] += buff[threadIdx.x + 256];
        else
            return; // Useless threads terminate
    }
    .....
}
    
```

57

Implantation en template C++

```

if (BLOCK_SIZE > 32) {
    __syncthreads(); // Barrière de synchro NECESSAIRE
    if (idx < 32) // Useful threads reduce data
        buff[threadIdx.x] += buff[threadIdx.x + 32];
    else
        return; // Useless threads terminate
}

if (BLOCK_SIZE > 16) {
    buff[threadIdx.x] += buff[threadIdx.x + 16];
}

if (BLOCK_SIZE > 8) {
    buff[threadIdx.x] += buff[threadIdx.x + 8];
}

.....

if (BLOCK_SIZE > 1) {
    buff[threadIdx.x] += buff[threadIdx.x + 1];
}

// Accumulation in global memory by th0 (warning 32 threads still alive)
if (threadIdx.x == 0) atomicAdd(AdrGRes, buff[0]);
}
    
```

Les 32 th survivants
sont dans 1 seul warp
→ SIMD pur

58

CUDA : optimized programming

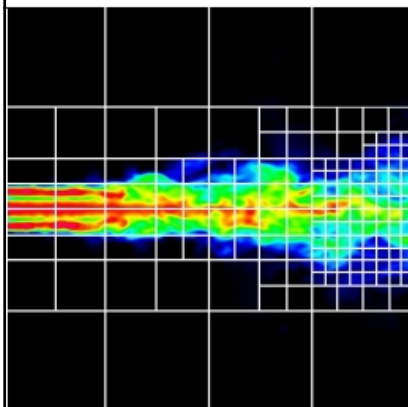
- 1 – Utilisation de la *shared memory*
- 2 – Réduction optimisée
- 3 – Déroulement de boucle auto-adaptatif
- 4 – Parallélisme dynamique sur GPU**
- 5 – Bilan de la programmation CUDA

59

Parallélisme dynamique

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

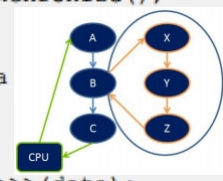


```

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);
    
```



60

CUDA : optimized programming

- 1 – Utilisation de la *shared memory*
- 2 – Réduction optimisée
- 3 – Déroulement de boucle auto-adaptatif
- 4 – Parallélisme dynamique sur GPU
- 5 – Bilan de la programmation CUDA**

61

Bilan de la programmation CUDA

Une nouvelle façon de programmer (ou que l'on redécouvre) :

- Demande une période d'apprentissage (!) debug difficile...
- Arriver à **identifier rapidement si un algorithme est adapté au GPU**
- Apprendre les optimisations principales : voir le « *CUDA C Best Practices Guide* ».

Performances :

- Annonces de gains *spectaculaires* vis-à-vis d'un coeur CPU
- **Souvent un gain de 2 à 10 seulement vis-à-vis d'un code parallèle et optimisé sur dual-CPU (serveur standard) !**
- Codes hybrides CPU+GPU efficaces mais restent plus complexes.

62

Bilan de la programmation CUDA

Les bonnes pratiques :

- Ecrire des kernels coalescents et non-divergents
- Utiliser la *shared memory* avec un « algo de cache dédié au pb »
- Terminer les threads devenues inutiles, et éliminer des *warps* entiers
- Ne pas oublier de resynchroniser les threads !
- Mais éliminer les synchronos quand il ne reste qu'un seul *warp* actif!
- Écrire des kernels génériques avec des constantes (connues à la compilation), afin que le compilateur :
 - élimine les lignes de code inutiles (par « `#define` » ou « `template functions` »)
 - spécialise le kernel pour le problème.

63

CUDA : optimized programming

End

64