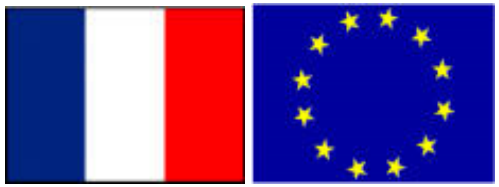


GP-GPU

Advanced CUDA programming Part 2

Stéphane Vialle

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

POLYTECH
PARIS-SACLAY

Advanced CUDA programming
Part 2

1 – Réduction optimisée
- **Optimisation du schéma de réduction**
- Stratégies d'implantation complète
- Implantation détaillée en *shared memory*

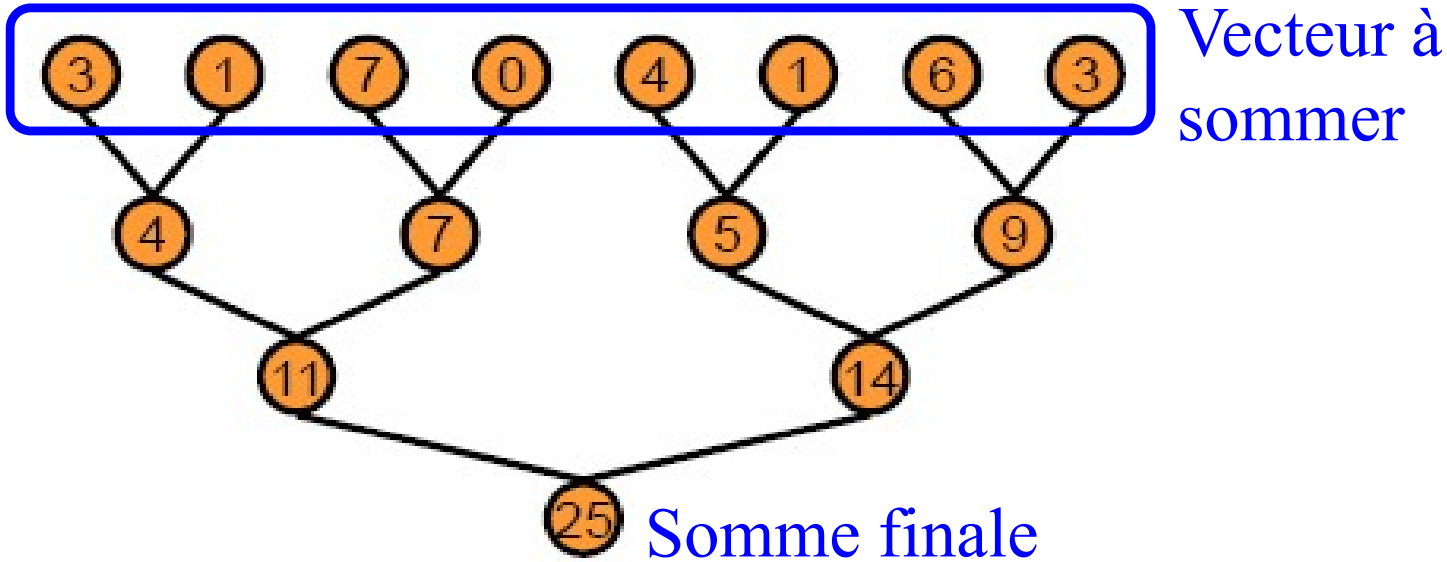
2 – Déroulement de boucle optimisé

3 – Occupation optimisée de registres

4 – Bilan de la programmation CUDA

Optimisation du schéma de réduction

Schéma de base :

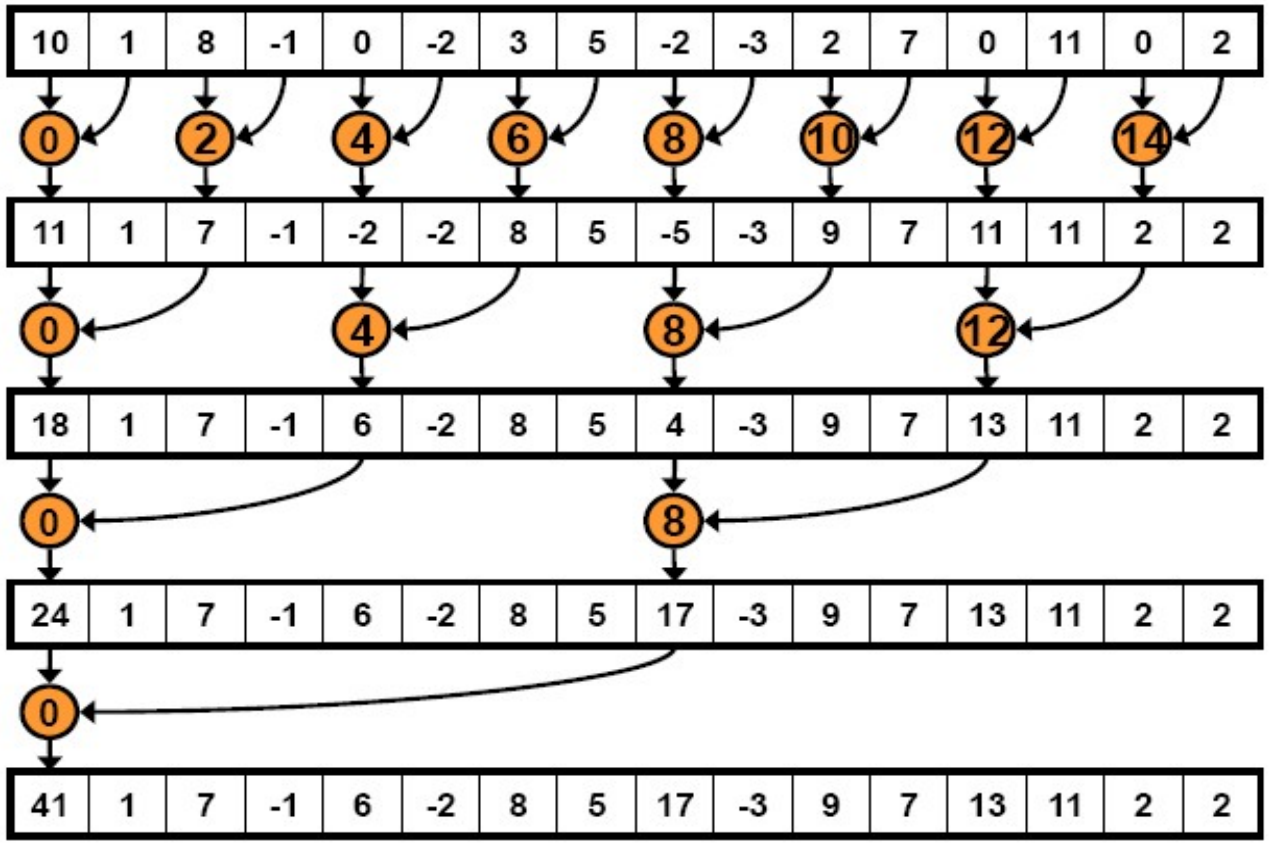


Une « réduction » contient du parallélisme difficile à exploiter :

- 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



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

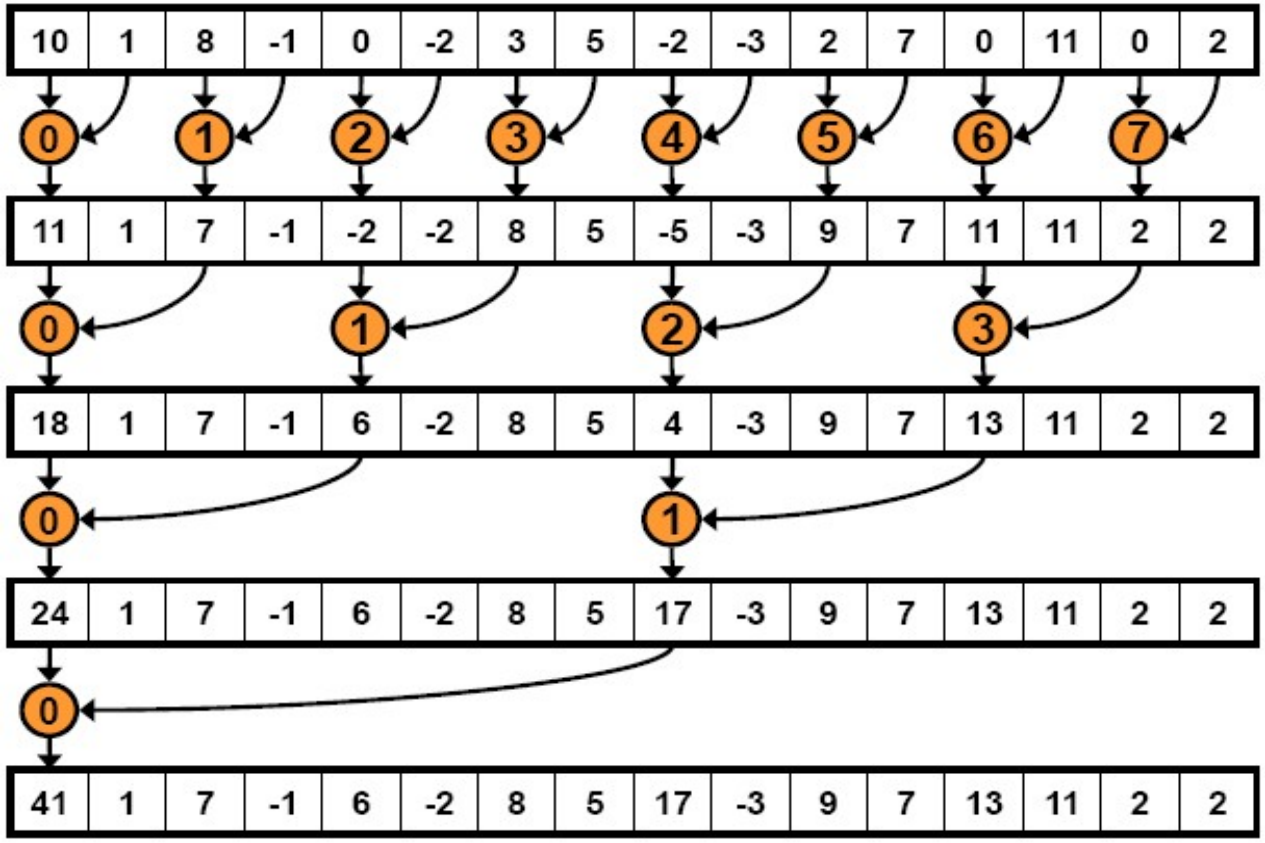
Données à réduire de + en + dispersées

→ Accès mémoire de moins en moins « coalescents » !

Thread actifs de + en + dispersées

→ Activations de « warps » très pauvres en threads actifs

Optimisation du schéma de réduction



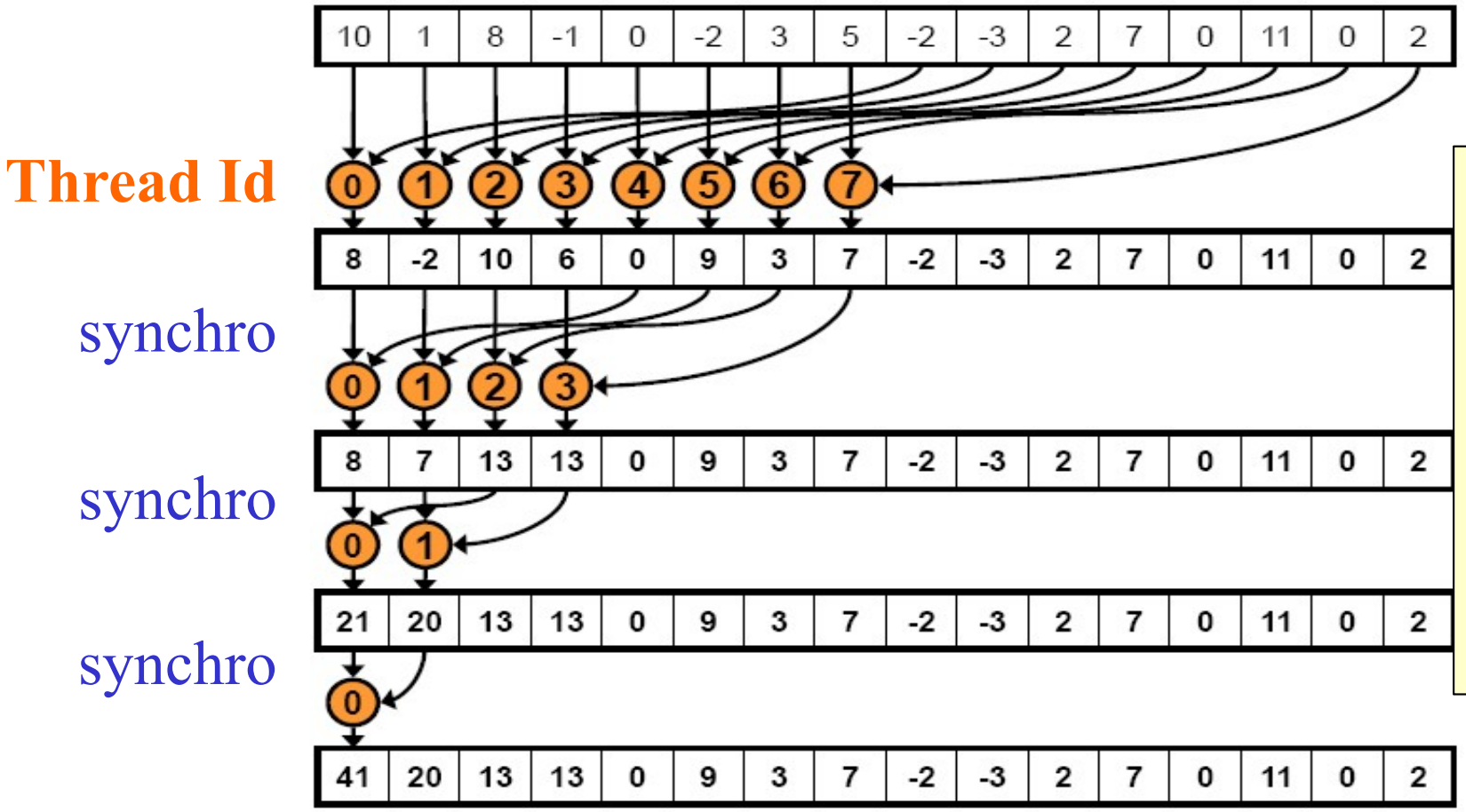
Divergence maîtrisée, mais **pas de coalescence**. Mauvaise stratégie sur GPU !

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 » !

Optimisation du schéma de réduction

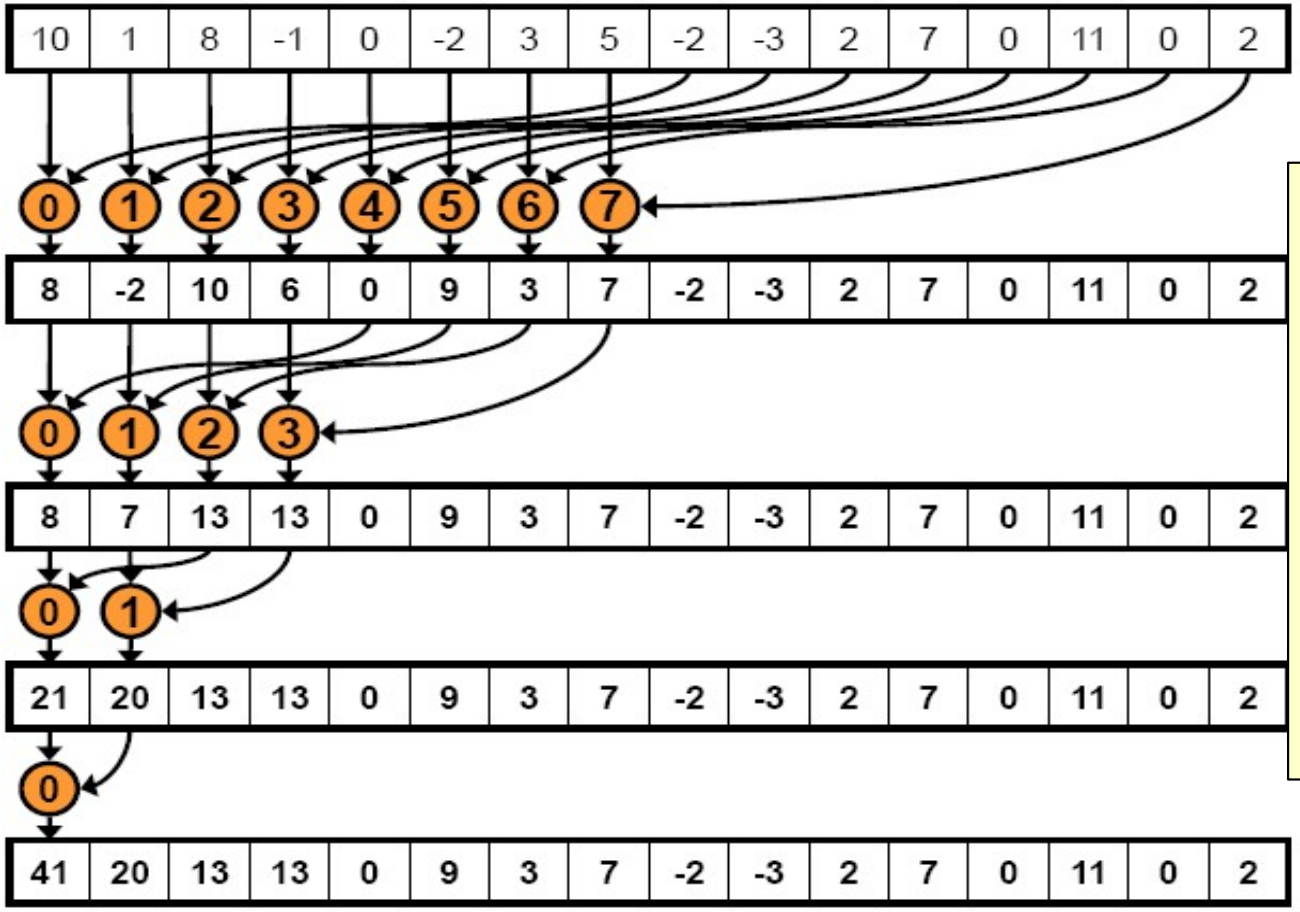


Pas de divergence et accès coalescents. **Bonne stratégie sur GPU !**

- Sous-ensembles de threads actifs « contiguës depuis le thread 0 »
- Accès mémoires qui restent coalescents

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

Optimisation du schéma de réduction



Pas de divergence et accès coalescents. **Bonne stratégie sur GPU** !

La synchronisation est simple et rapide entre les threads d'UN bloc

→ On va devoir faire des réductions locales dans chaque bloc

→ Puis combiner ces réductions locales…(voir plus loin)

Implantation coalescente et peu divergente

Réduction au sein d'un bloc

```
int idx = blockIdx.x*BLOCKSIZE + threadIdx.x;

if (idx%2 == 0) { //1st step: half of the threads work
  A[idx] = A[idx] + A[idx+1];
}
__syncthreads();

if (idx%4 == 0) { //2nd step: a quarter of the threads work
   ……
}
…
```

Mauvais

```
int idx = blockIdx.x*BLOCKSIZE + threadIdx.x;

//1st step: half of the threads work
if (threadIdx.x < BLOCKSIZE_X/2) {
  A[idx] = A[idx] + A[idx + BLOCKSIZE_X/2];
}
__syncthreads();

//2nd step: a quarter of the threads work
if (threadIdx.x < BLOCKSIZE_X/4) ……
……
```

BON!

Implantation coalescente et peu divergente

On peut même terminer explicitement les threads inutiles

```
int idx = blockIdx.x*BLOCKSIZE + threadIdx.x;

//1st step: half of the threads work
if (threadIdx.x < BLOCKSIZE_X/2) {
  A[idx] = A[idx] + A[idx + BLOCKSIZE_X/2];
else return;

__syncthreads();

//2nd step: a quarter of the threads work
if (threadIdx.x < BLOCKSIZE_X/4) {
  A[idx] = A[idx] + A[idx + BLOCKSIZE_X/4];
else return;

__syncthreads();
……
```

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

POLYTECH
PARIS-SACLAY

Advanced CUDA programming
Part 2

1 – Réduction optimisée
- Optimisation du schéma de réduction
- **Stratégies d'implantation complète**
- Implantation détaillée en *shared memory*

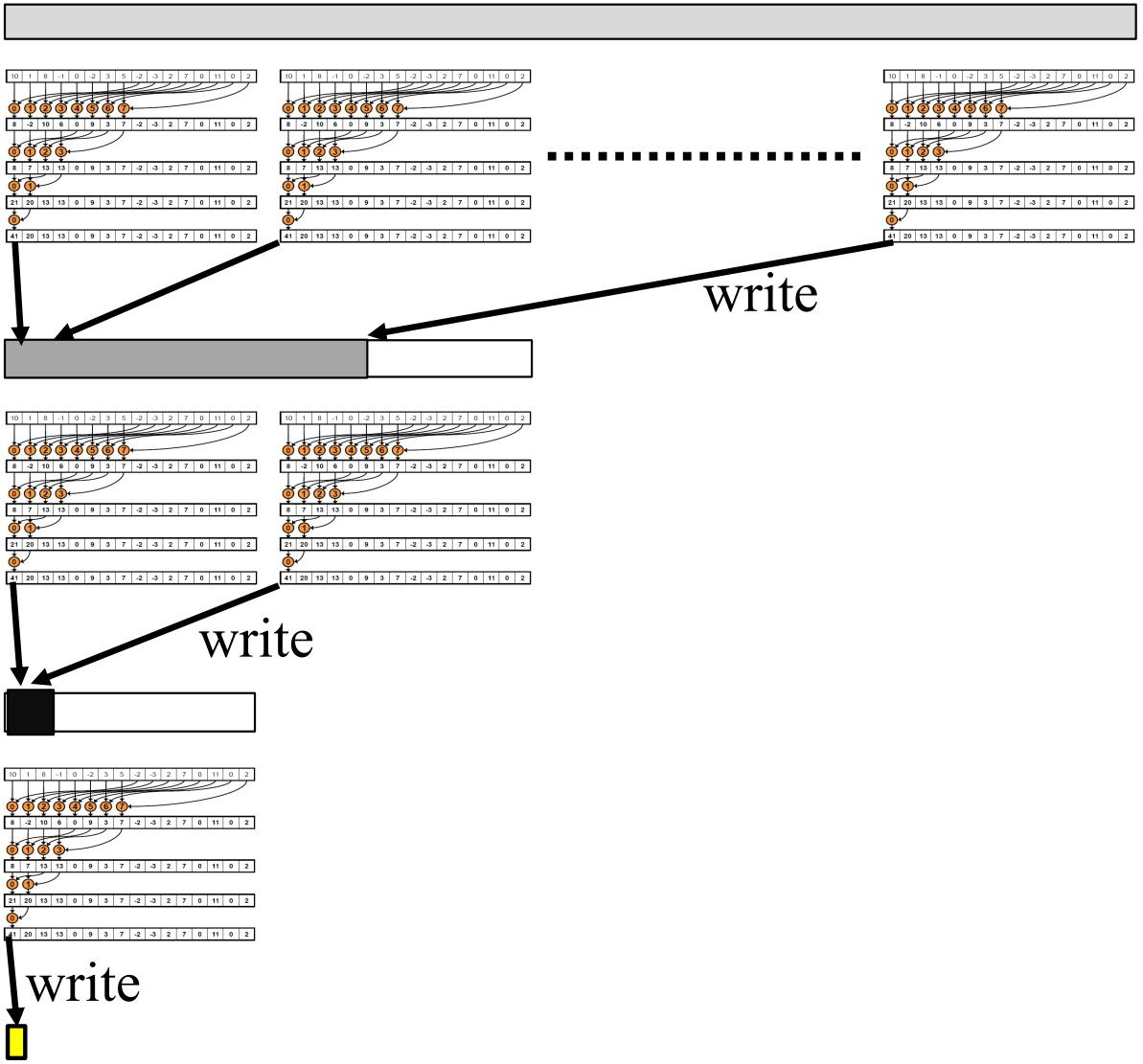
2 – Déroulement de boucle optimisé

3 – Occupation optimisée de registres

4 – Bilan de la programmation CUDA

Réduction optimisée

Approche récursive



Input data 0

Grid 0

Input data 1

Grid 1

Input data 2

Grid 2

Output data

Nombre de grilles successives fonction de la quantité de données et de la taille des blocs

Approche récursive : kernel du niveau 0

Chaque bloc travaille sur **A0** et sauve sa réduction partielle dans **A1**

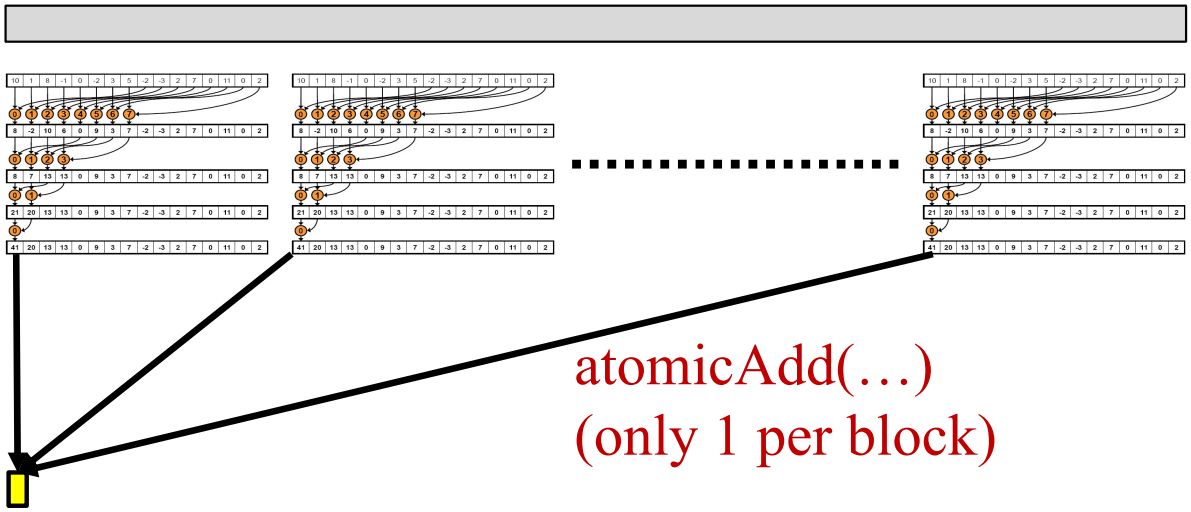
```
int idx = blockIdx.x*BSX + threadIdx.x;
……
// Reduction of the values
// - first step
if (threadIdx.x < BLOCKSIZE_X/2)
  A0[idx] = A0[idx] + A0[idx + BLOCKSIZE_X/2];
else return;

__syncthreads();

// - second step
if (threadIdx.x < BLOCKSIZE_X/4)
 A0[idx] = A0[idx] + A0[idx + BLOCKSIZE_X/4];
else return;

__syncthreads();

……
// - last step: final action by thread 0 alone
A1[blockIdx.x] = A0[idx] + A0[idx+1];
```

Approche récursive : kernel du niveau 0

Tous les threads travaillent au moins à une étape

```
int idx = blockIdx.x*(2*BSX) + threadIdx.x;
……
// Reduction of the values
// - first step
//if (threadIdx.x < BLOCKSIZE_X)
  A0[idx] = A0[idx] + A0[idx + BLOCKSIZE_X];
//else return;

__syncthreads();

// - second step
if (threadIdx.x < BLOCKSIZE_X/2)
 A0[idx] = A0[idx] + A0[idx + BLOCKSIZE_X/2];
else return;

__syncthreads();

……

// - last step: final action by thread 0 alone
A1[blockIdx.x] = A0[idx] + A0[idx+1];
```

Approche utilisant des *atomics*



Une seule grille de blocs faisant des accès coalescents
+ un atomicAdd par bloc

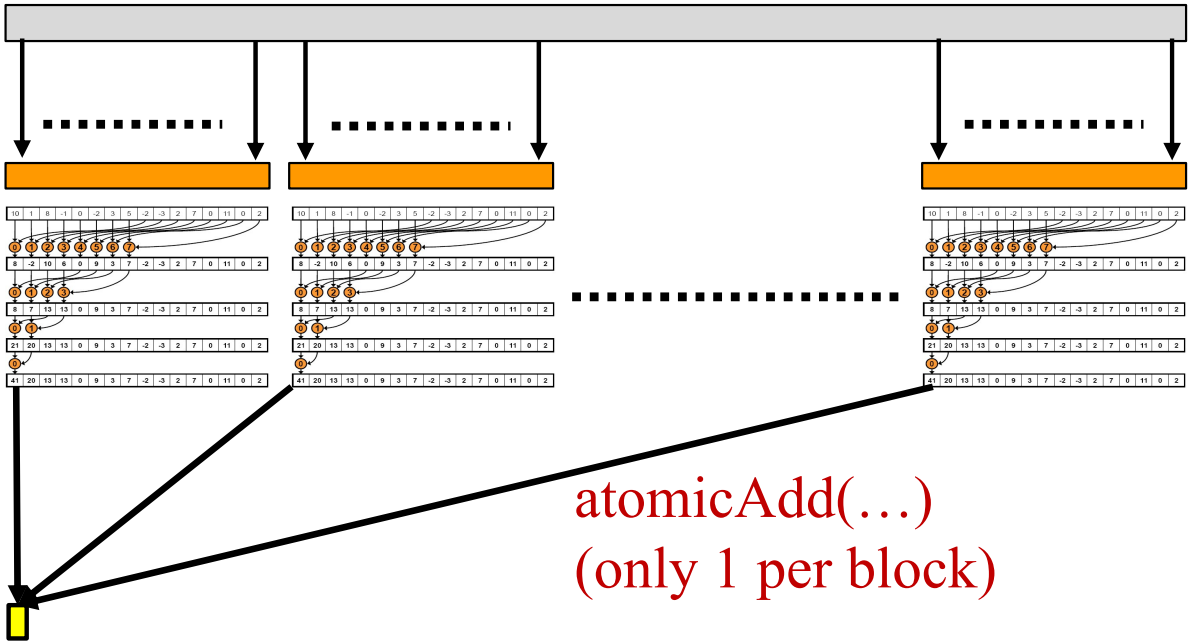
Bien limiter les atomicAdd à un par bloc
(sinon ce sera lent!)

Approche *atomics*

Chaque bloc travaille sur **A0** and sauve sa réduction partielle dans ***PtReduced**

```
int idx = blockIdx.x*(2*BSX) + threadIdx.x;
……
// Reduction of the values
// - first step
//if (threadIdx.x < BLOCKSIZE_X)
  A0[idx] = A0[idx] + A0[idx + BLOCKSIZE_X];
//else return;

__syncthreads();

// - second step
if (threadIdx.x < BLOCKSIZE_X/2)
 A0[idx] = A0[idx] + A0[idx + BLOCKSIZE_X/2];
else return;

__syncthreads();

……

// - last step: final action by thread 0 alone
atomicAdd(PtReduced, A0[idx] + A0[idx+1]);
```

Approche utilisant des *atomics* et la *shared*



Une seule grille de blocs :

- chargement coalescent des shared memories
- réduction locale dans chaque shared memory
  (encore plus efficace si on fait des accès coalescent en shm!)
- + un atomicAdd par bloc

Approche *atomics + shared memory*

```
int idx = blockIdx.x*BSX + threadIdx.x;
__shared__ Sh[BSX];

// Loading the shared memory: all threads working
Sh[threadIdx.x] = A[idx];
__syncthreads();

// Reduction of the values
// - first step
if (threadIdx.x < BLOCKSIZE_X/2)
 Sh[threadIdx.x] += Sh[threadIdx.x + BLOCKSIZE_X/2];
else return;

__syncthreads();

// - second step
if (threadIdx.x < BLOCKSIZE_X/4)
 Sh[threadIdx.x] += Sh[threadIdx.x + BLOCKSIZE_X/4]; else
return;

__syncthreads();

……

// - last step: final action by thread 0 alone
atomicAdd(PtReduced, Sh[0] + Sh[1]);
```

POLYTECH
PARIS-SACLAY

Advanced CUDA programming
Part 2

1 – Réduction optimisée
- Optimisation du schéma de réduction
- Stratégies d'implantation complète
- **Implantation détaillée en *shared memory***

2 – Déroulement de boucle optimisé

3 – Occupation optimisée de registres

4 – Bilan de la programmation CUDA

Implantation détaillée 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
}
```

Implantation détaillée 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 ☺

POLYTECH
PARIS-SACLAY

Advanced CUDA programming
Part 2

1 – Réduction optimisée
2 – Déroulement de boucle optimisé
- **Auto-adaptation à la compilation**
- Optimisation SIMD
- Implantation en template C++

3 – Occupation optimisée de registres
4 – Bilan de la programmation CUDA

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

Déroulement de boucle optimisé

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
```

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
```

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]);
}
```

POLYTECH
PARIS-SACLAY

Advanced CUDA programming
Part 2

1 – Réduction optimisée
2 – Déroulement de boucle optimisé
- Auto-adaptation à la compilation
- **Optimisation SIMD**
- Implantation en template C++

3 – Occupation optimisée de registres
4 – Bilan de la programmation CUDA

Déroulement de boucle optimisé

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

POLYTECH PARIS-SACLAY

Déroulement de boucle optimisé

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

 ..........................
```

Déroulement de boucle optimisé

Optimisation SIMD

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

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

.....

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

// Accumulation in global memory by th0 (warning 32
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**

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

Déroulement de boucle optimisé

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**

POLYTECH
PARIS-SACLAY

Advanced CUDA programming
Part 2

1 – Réduction optimisée
2 – Déroulement de boucle optimisé
- Auto-adaptation à la compilation
- Optimisation SIMD
- **Implantation en template C++**

3 – Occupation optimisée de registres
4 – Bilan de la programmation CUDA

Implantation en template C++

NVCC est un compilateur C++…

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

Déroulement de boucle optimisé

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
 }

 ..........
```

Déroulement de boucle optimisé

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**

POLYTECH
PARIS-SACLAY

Advanced CUDA programming
Part 2

1 – Réduction optimisée
2 – Déroulement de boucle optimisé
3 – Occupation optimisée de registres
4 – Bilan de la programmation CUDA

Occupation optimisée de registres

Au cours d'un développement de code CUDA de Clustering:

- Conception de plusieurs « métriques » de distance entre données
- Pour chaque métrique : calculs différents au sein d'un même kernel
- Implantation avec un `switch(metric) {case 0:… case 1:…}`

→ Plus on développe de métriques différentes, plus les performances baissent!

→ Et il faut diminuer la taille des blocs pour obtenir des performances optimales!

Occupation optimisée de registres

```
__global__ void k(int metric)
{
  int row, col;
  int nnzThread = ...;
  __shared__ int shNnzRow;

  ...
  while (col < maxCol) {
    switch (metric) {
     case 0 :
     {
      T_real diff, distSq;
      ...
     }
     break;
     case 1 :
     {
      T_real diff, distSq;
      ...
     }
     break;

     case 2 :
     {
      T_real diff, distSq;
      ...
      T_real sim;
      ...
     }
     break;
     case 3 :
     {
      T_real diff, distSq;
      ...
     }
     break;
     case 4 :
     {
      T_real diff, distSq;
      ...
     }
     break;
     case 5 :
     {
      T_real elm1, elm2,
             dot, sq1, sq2;
      ...
      T_real sim_sq;
      ...
     }
     break;
     default :
      return;
    }
    ...
  }
  ...
}
```

Occupation optimisée de registres

La déclaration de plus en plus de variables locales à chaque « case » entraine la réservation de plus en plus de registres

→ De moins en moins de blocs « résidents » dans le même StreamProcessor

→ Une moins bonne utilisation du GPU

Solution 1

Diminuer la taille des blocs (efficace)

Rechercher régulièrement la taille de bloc optimale (qui diminue quand on ajoute des métriques)

Solution 2

Ecrire un code qui ne compile que la bonne métrique!

Avec des : `#ifdef … #endif`

Avec des Template C++ : `__global__ void k<int Metric>(…)`

Occupation optimisée de registres

```
__global__ void k<int METRIC>(…)
{
  int row, col;
  int nnzThread = ...;
  __shared__ int shNnzRow;

  ...
  while (col < maxCol) {
    switch (METRIC) {
     case 0 :
     {
      T_real diff, distSq;
      ...
     }
     break;
     case 1 :
     {
      T_real diff, distSq;
      ...
     }
     break;
     case 2 :
     {
      T_real diff, distSq;
      ...
      T_real sim;
      ...
     }
     break;
     case 3 :
     {
      T_real diff, distSq;
      ...
     }
     break;
     case 4 :
     {
      T_real diff, distSq;
      ...
     }
     break;
     case 5 :
     {
      T_real elm1, elm2,
             dot, sq1, sq2;
      ...
      T_real sim_sq;
      ...
     }
     break;
     default :
      return;
    }
    ...
  }
  ...
}
```

POLYTECH
PARIS-SACLAY

Advanced CUDA programming
Part 2

1 – Réduction optimisée
2 – Déroulement de boucle optimisé
3 – Occupation optimisée de registres
4 – Bilan de la programmation CUDA

POLYTECH PARIS-SACLAY

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.

POLYTECH PARIS-SACLAY

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 synchros 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.

POLYTECH
PARIS-SACLAY

Advanced CUDA programming
Part 2

End