

GP-GPU

CUDA : More optimizations

St phane Vialle

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

CUDA : optimized programming

1 – Réduction optimisée

- **Optimisation du schéma de réduction**
- Implantation coalescente et peu divergente
- Implantation en *shared memory*

2 – Kernels auto-adaptatifs

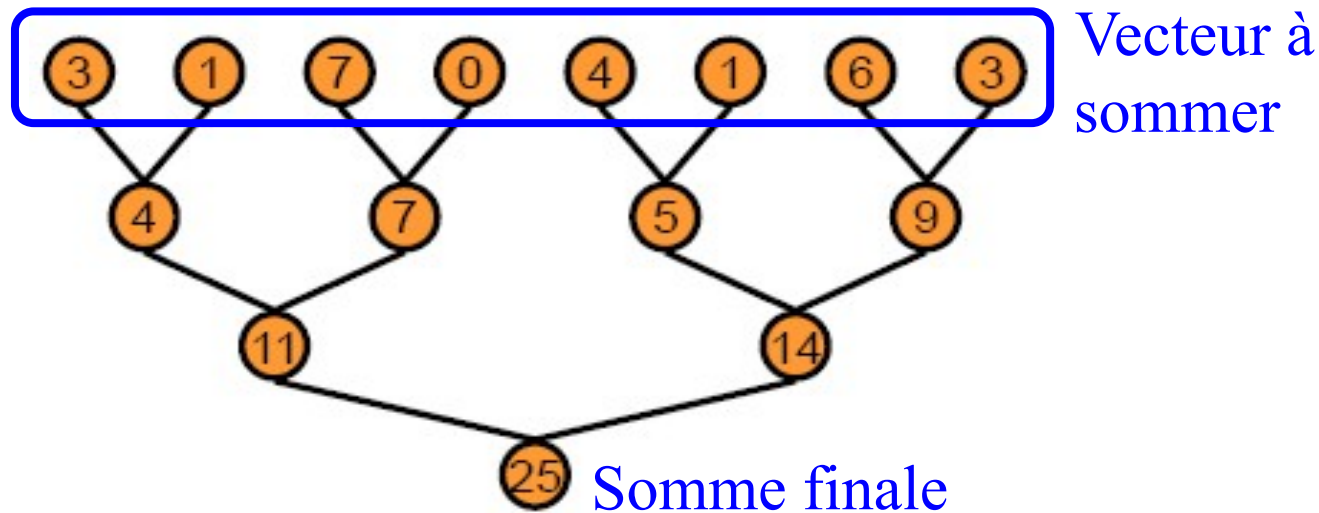
3 – Parallélisme dynamique sur GPU

4 – Opérations atomiques

5 – Bilan de la programmation CUDA

Optimisation du schéma de réduction

Schéma de base :



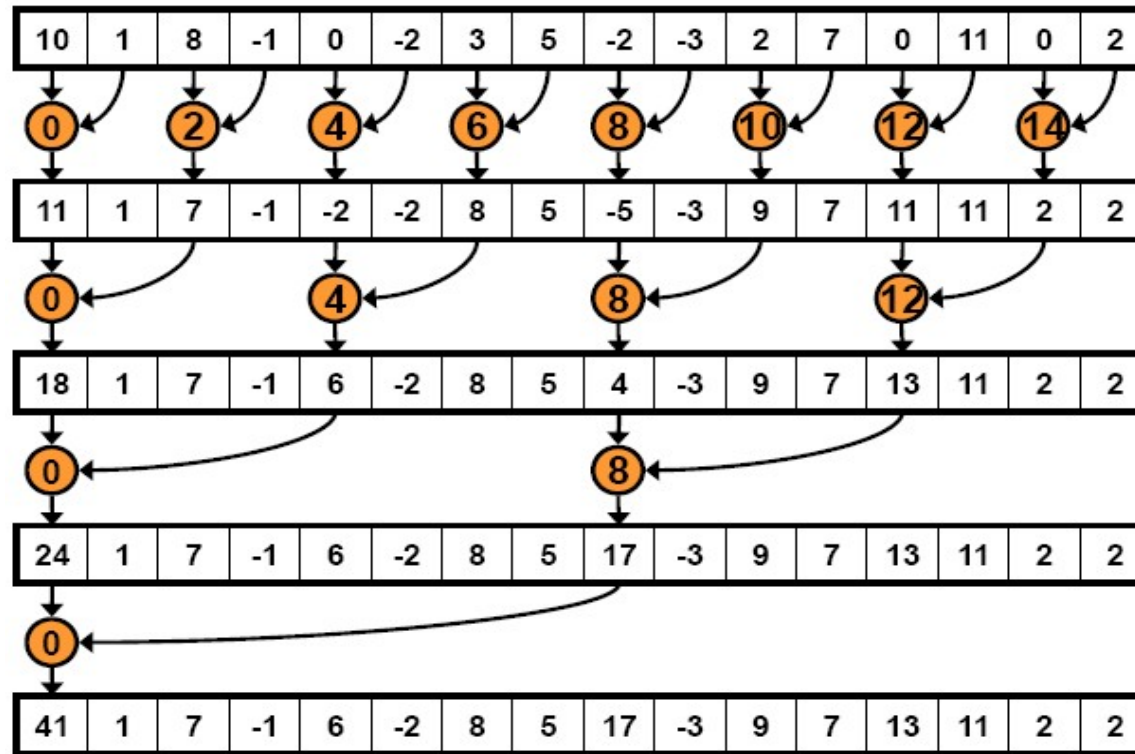
Une « réduction » contient du parallélisme difficile à exploiter :

- 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

Thread Id



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

Données à réduire de + en + dispersées

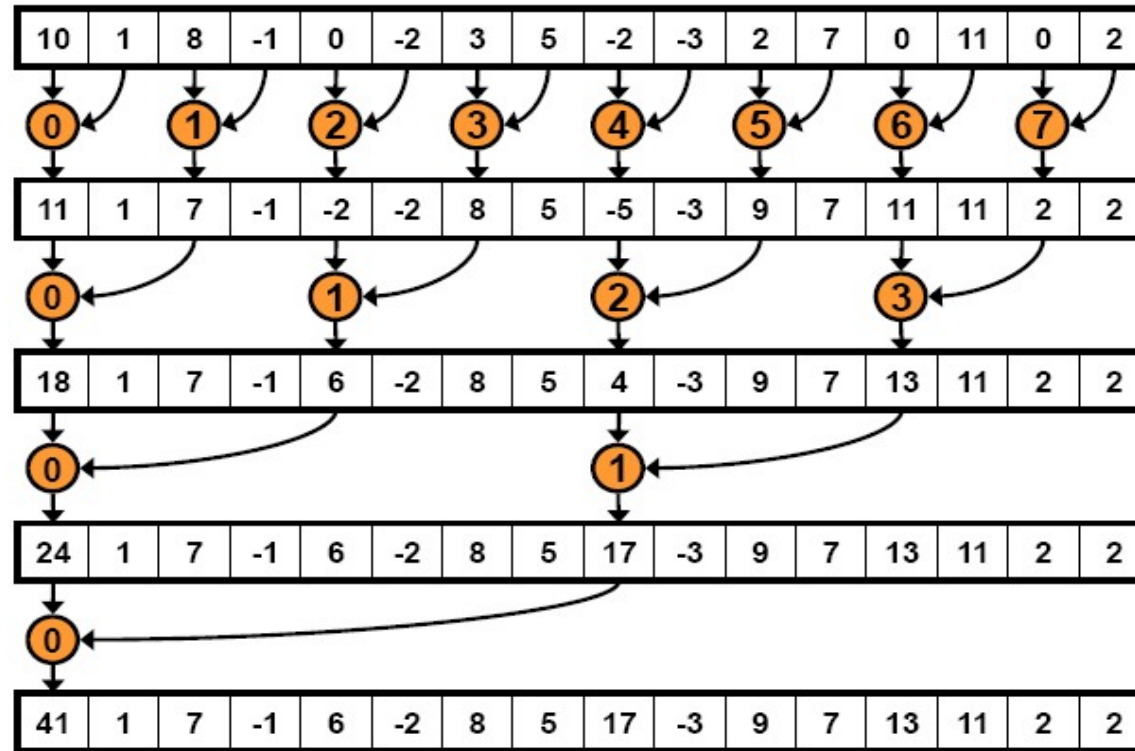
→ Accès mémoire de moins en moins « coalescents » !

Thread actifs de + en + dispersés

→ Activations de « warps » très pauvres en threads actifs

Optimisation du schéma de réduction

Thread Id



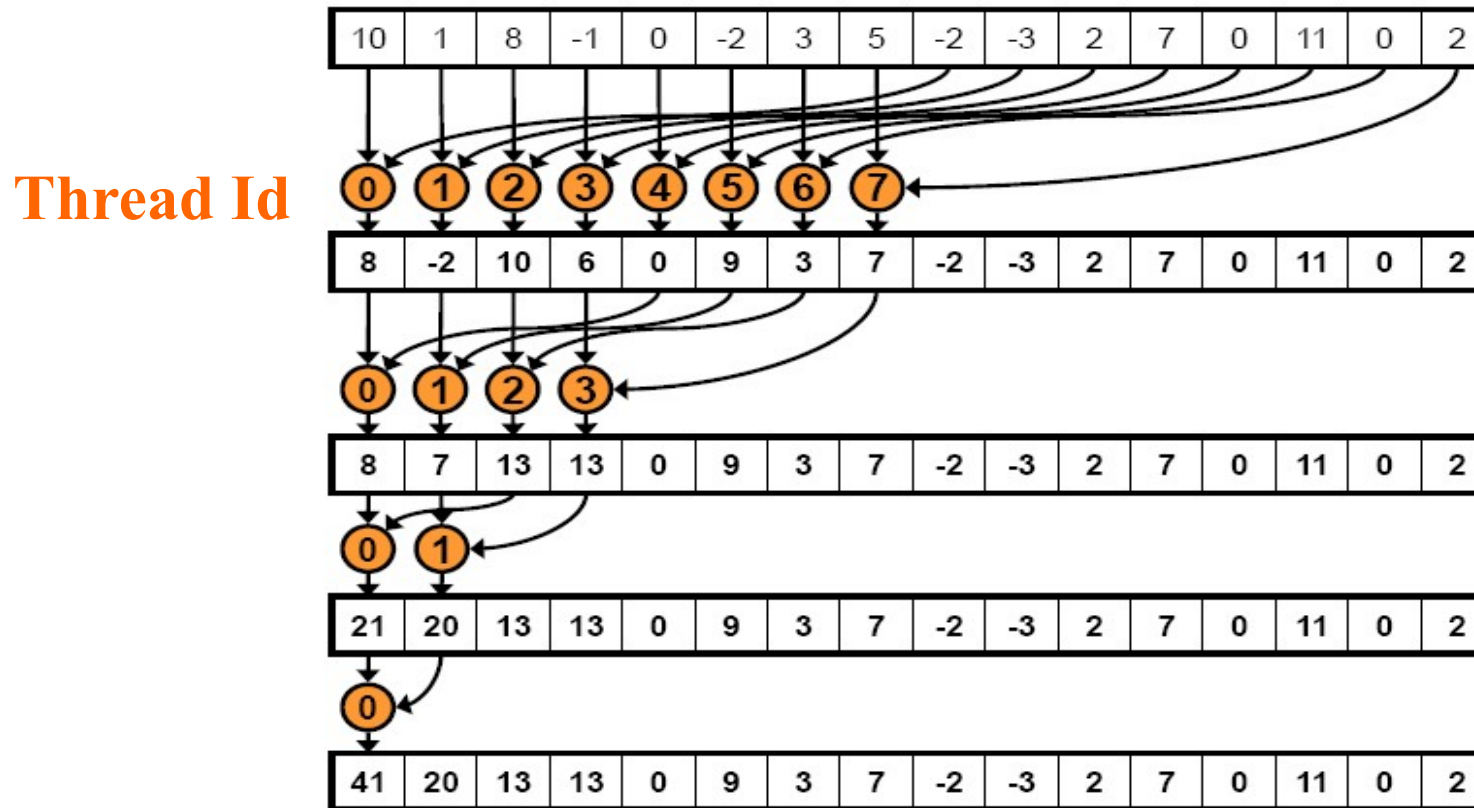
Divergence
maîtrisée,
mais **pas de
coalescence**.
Mauvaise
stratégie sur
GPU !

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 » !

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 ?**

CUDA : optimized programming

1 – Réduction optimisée

- Optimisation du schéma de réduction
- **Implantation coalescente et peu divergente**
- Implantation en *shared memory*

2 – Kernels auto-adaptatifs

3 – Parallélisme dynamique sur GPU

4 – Opérations atomiques

5 – Bilan de la programmation CUDA

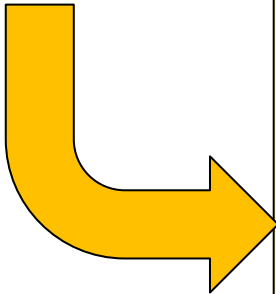
Implantation coalescente et peu divergente

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


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)

CUDA : optimized programming

1 – Réduction optimisée

- Optimisation du schéma de réduction
- Implantation coalescente et peu divergente
- **Implantation en *shared memory***

2 – Kernels auto-adaptatifs

3 – Parallélisme dynamique sur GPU

4 – Opérations atomiques

5 – Bilan de la programmation CUDA

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
}
```

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 😊

CUDA : optimized programming

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

Auto-adaptation à la compilation

Principe :

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

Solution :

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

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]);
}
```

CUDA : optimized programming

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

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

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

    .....
}
```

Optimisation SIMD

```

#if BLOCK_SIZE > 16
__syncthreads(); // Barrière de synchro NEC
if (threadIdx.x < 16) // Useful threads reduce
    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
    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
    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 !

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

CUDA : optimized programming

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

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
    }

    .....
}
```

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

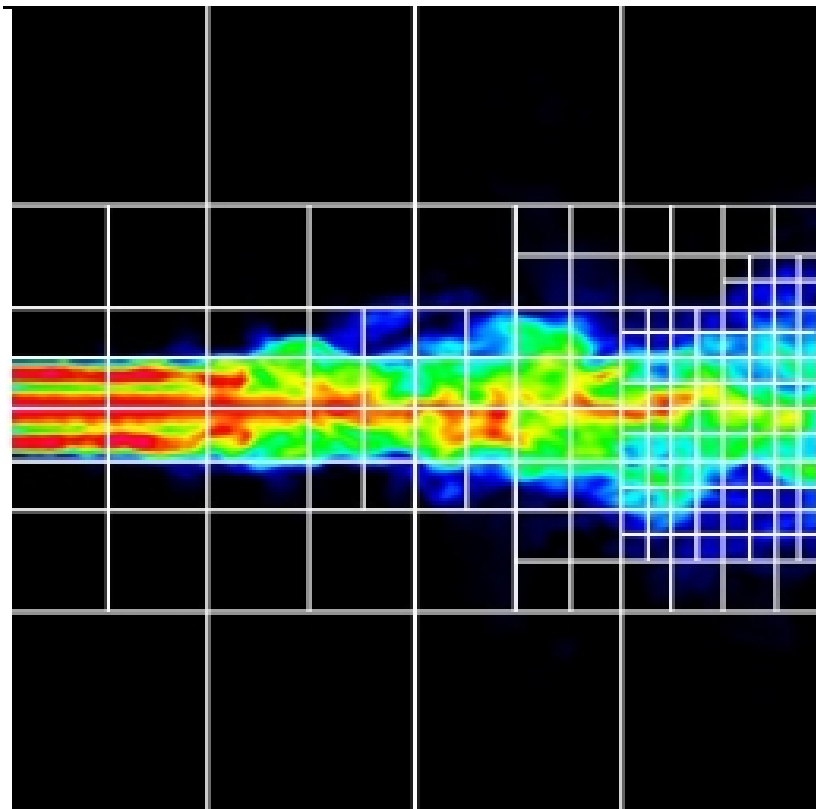
CUDA : optimized programming

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

Parallélisme dynamique

Un thread GPU peut lancer d'autres threads GPU

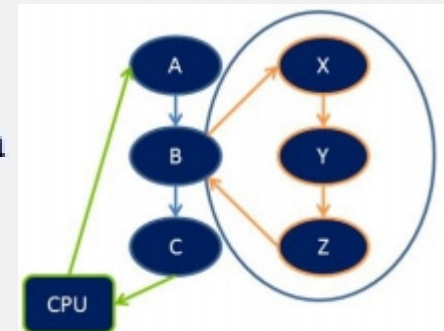
- Un thread définit et lance lui-même une grille de blocs de threads
- Très utile pour des maillages adaptatifs



```
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);
```



CUDA : optimized programming

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

CUDA : optimized programming

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

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.

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.

CUDA : More optimizations

End