

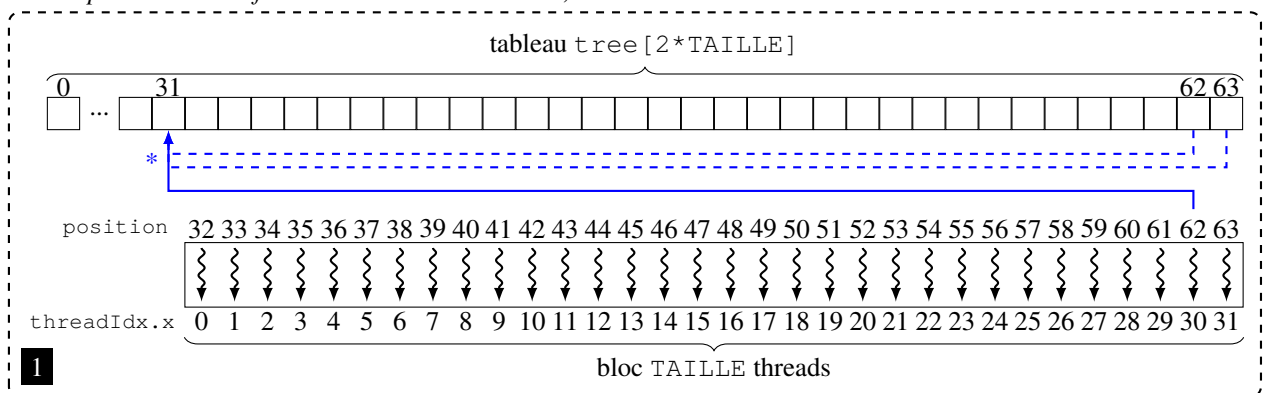
Programmation GPGPU

■ ■ ■ Structures de données et GPGPU

1 – 1. Analysez et décrivez son fonctionnement.

En analysant le programme, on note que :

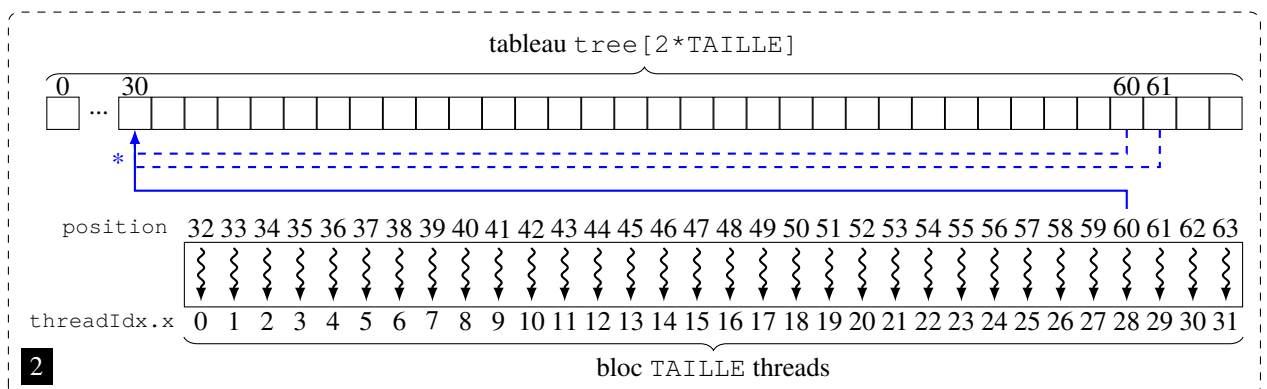
- ▷ en ligne 23 : on réserve de l'espace pour un tableau, `tree`, de taille $2 * TAILLE$;
- ▷ des lignes 33 à 37 :
 - * on lance plusieurs fois un «kernel» sur une grille d'un bloc de `TAILLE` «threads» ;
 - * à chaque lancement, on passe le paramètre `t` qui varie de `t=TAILLE`, à `t=TAILLE/2`, puis `t=TAILLE/4` jusqu'à `t=1`.
- ▷ pour décrire le fonctionnement du «kernel», on va commenter des schémas :



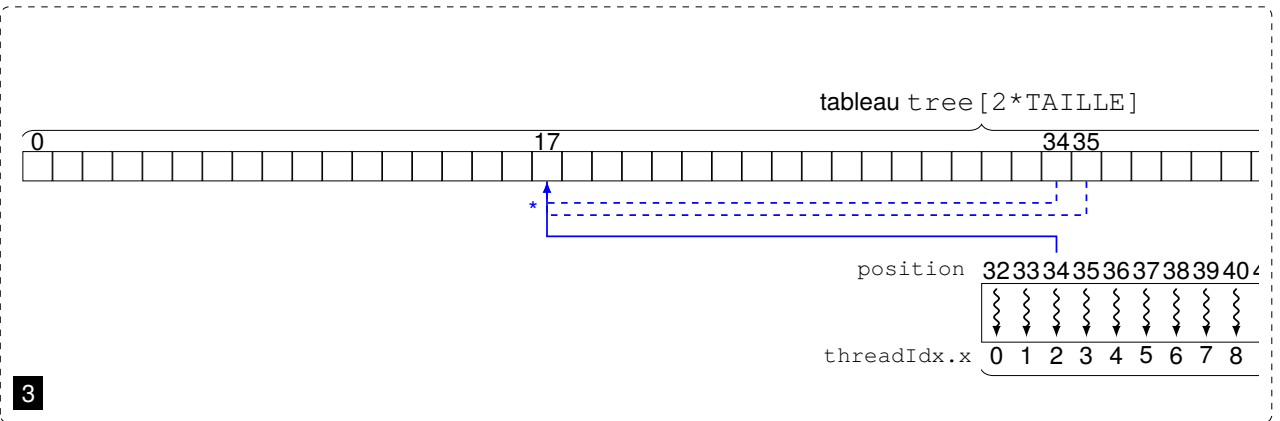
Dans le «kernel» :

- le bloc de «threads» est décalé par la variable `r` ;
- au début, `r=32` :
 - ▷ en ligne 10 : la thread 0 a pour position 32, la thread 1 a pour position 33, jusqu'à la thread 31 qui a pour position 63 ;
 - ▷ en ligne 12 : seules les threads de position paire travaillent ;
 - ▷ en ligne 14 : la thread utilise sa position pour faire le produit de la case du tableau qui lui correspond `t[position]` et de la case `t[position+1]` et range le résultat dans la case `t[position/2]`

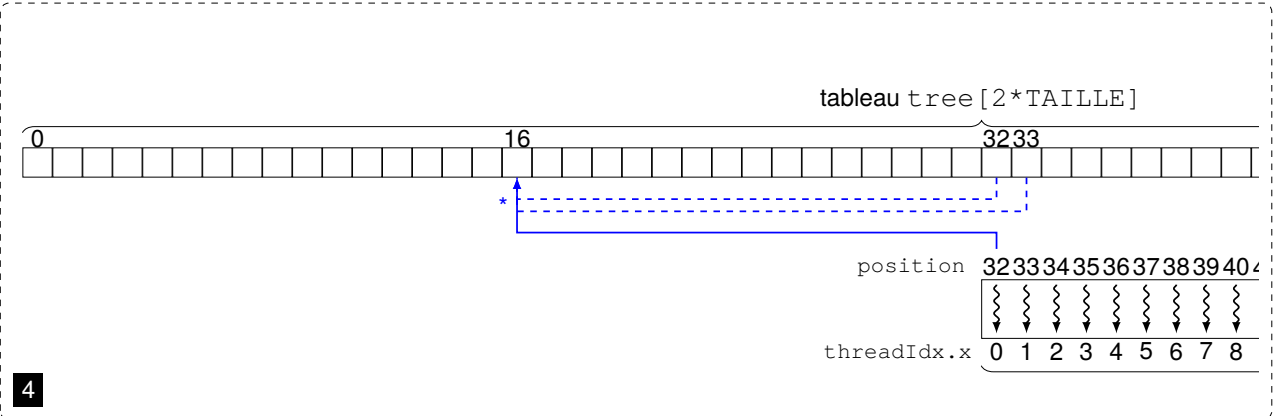
Ici, sur le schéma 1 la thread 62 range le produit des cases 62 et 63 dans la case 31.



Ici, sur le schéma 2 la thread 60 range le produit des cases 60 et 61 dans la case 30.



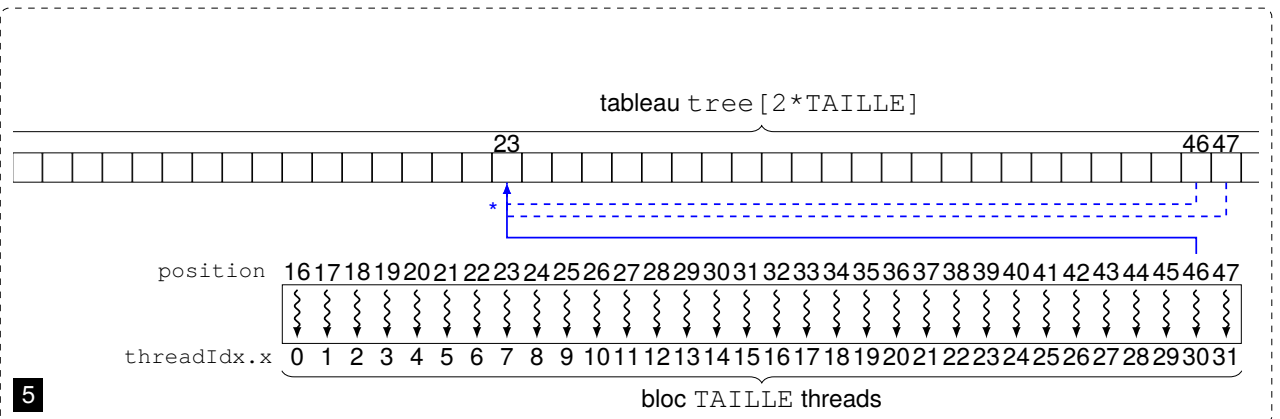
Ici, sur le schéma 3 la thread 34 range le produit des cases 34 et 35 dans la case 17.



Ici, sur le schéma 4 la thread 32 range le produit des cases 32 et 33 dans la case 17.

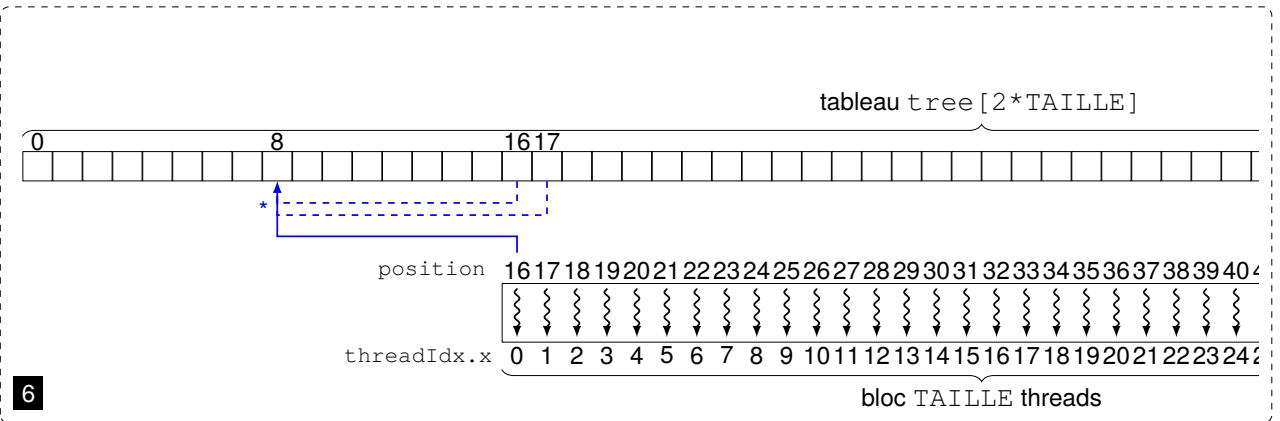
Le bloc ne contient que 32 threads, soit un warp : le travail est en SIMD, toutes les threads demandent d'abord l'accès à $t[position]$ puis $t[position+1]$.

Les accès sont contigus, mais on fait des demandes à des cases déjà obtenues par d'autres threads.



Sur le schéma 5, on passe à un nouvel appel de kernel :

- ▷ les données du tableau t restent en place dans la mémoire de la carte GPU ;
- ▷ $r = 16$, l'association des threads est décalée : la thread 0 a pour position 16, la thread 1 a pour position 17, jusqu'à la thread 31 qui a pour position 47 ;
- ▷ la thread 46 fait le travail sur la case 23, ce travail est **inutile** car il a déjà été fait dans l'étape précédente : il ne change pas la valeur de la case 23 ;



Ici, sur le schéma **6** la thread 16 range le produit des cases 16 et 17 dans la case 8.
On va continuer ainsi jusqu'à $r=2$, où le seul travail utile sera quand la thread 2 va ranger le produit des cases 2 et 3 dans la case 1 (toutes les autres threads font un travail inutile, car déjà dans les étapes précédentes).

2. Soit la trace d'exécution : Vérifiez qu'elle est **correcte**.

D'après l'analyse réalisée dans la question précédente, on a un produit de niveaux en niveaux dans un arbre binaire avec un nombre d'étape égal au $\log_2(TAILLE)$ et une valeur finale égale à 2^{TAILLE} .

Ici, $TAILLE=32$ et le résultat est $2^{32} = 4294967296$.

3. Donnez une **version améliorée**.

Voici une première version où :

- ◊ la valeur de r est mise à jour directement dans le kernel, au lieu de relancer plusieurs fois le kernel depuis la fonction `main` ;
- ◊ on utilise un tableau dans la mémoire partagée pour stocker les valeurs du tableau `t` ;

```

1 #include <stdio.h>
2 #include <cuda.h>
3
4 #define TAILLE 64
5
6 float tree[2*TAILLE];
7
8 __global__ void mon_noyau(float *t, int r)
9 {
10     __shared__ float cache[TAILLE*2];
11     cache[threadIdx.x]=t[threadIdx.x];
12     cache[threadIdx.x+TAILLE]=t[threadIdx.x+TAILLE];
13
14     int position;
15     while(r>1)
16     {
17         position = threadIdx.x + r;
18         if ((position%2) == 0)
19         {
20             cache[position/2] = cache[position] * cache[position+1];
21         }
22         r = r/2;
23     }
24     if (threadIdx.x == 0)
25         t[1] = cache[1];
26 }
27
28 int main(void)
29 {
30     float *gpu_tree;
31
32     cudaMalloc((void **)&gpu_tree, TAILLE*2*sizeof(float));
33     for(int i=TAILLE; i<TAILLE*2; i++)
34         tree[i] = float(2);
35     cudaMemcpy(gpu_tree, tree, 2*TAILLE*sizeof(float), cudaMemcpyHostToDevice);
36     mon_noyau<<<1, TAILLE>>>(gpu_tree, TAILLE);
37     cudaMemcpy(tree, gpu_tree, 2*TAILLE*sizeof(float), cudaMemcpyDeviceToHost);
38     printf("[%4.0f]", tree[1]);
39     printf("\n");
40 }

```

Le résultat du calcul est satisfaisant si on passe $TAILLE$ à 64 avec un résultat égal à $2^{64} = 18446744073709551616$.

Par contre, si l'on passe à TAILLE à 128 :

```
xterm
[ inf]
```

On a un **dépassement de capacité** du à la taille limitée des valeurs flottantes.

On change alors l'opération réalisée par le kernel de la multiplication à l'addition :

```
15 while(r>1)
16 {
17     position = threadIdx.x + r;
18     if ((position%2) == 0)
19     {
20         cache[position/2] = cache[position] + cache[position+1];
21     }
22     r = r/2;
```

On obtient le résultat :

```
xterm
[ 256]
```

Ce qui correspond à $256 = 2 * TAILLE = 2 * 128$

On notera que l'on utilise 4 warps pour l'exécution du kernel.

Si on passe à :

- ▷ TAILLE = 256, soient 8 warps, le résultat est conforme avec la valeur résultat 512 ;
- ▷ TAILLE = 512, soient 16 warps, le résultat est conforme avec la valeur résultat 1024 ;
- ▷ TAILLE = 1024, soient 32 warps, le résultat est **instable** avec la valeur résultat oscillant entre 1984, 1920, 1728, 1856, 2048, 1888...

(on peut utiliser la commande `watch -n 1 ./tree_ameliore` pour relancer le programme toutes les secondes).

Le programme précédent est donc...**FAUX** !

Il y a de **nombreux warps** qu'il faut **synchroniser**, d'où la version correcte :

```
1 #include <stdio.h>
2 #include <cuda.h>
3
4 #define TAILLE 1024
5 float tree[2*TAILLE];
6
7 __global__ void mon_noiay(float *t, int r)
8 {
9     __shared__ float cache[TAILLE*2];
10    cache[threadIdx.x]=t[threadIdx.x];
11    cache[threadIdx.x+TAILLE]=t[threadIdx.x+TAILLE];
12    __syncthreads();
13
14    int position;
15    while(r>1)
16    {
17        position = threadIdx.x + r;
18        if ((position%2) == 0)
19        {
20            cache[position/2] = cache[position] + cache[position+1];
21        }
22        r = r/2;
23        __syncthreads();
24    }
25    __syncthreads();
26    if (threadIdx.x == 0)
27        t[1] = cache[1];
28 }
29
30 int main(void)
31 {
32     float *gpu_tree;
33
34     cudaMalloc((void **) &gpu_tree, TAILLE*2*sizeof(float));
35     for(int i=TAILLE; i<TAILLE*2; i++)
36         tree[i] = float(2);
37     cudaMemcpy(gpu_tree, tree, 2*TAILLE*sizeof(float), cudaMemcpyHostToDevice);
38     mon_noiay<<<1, TAILLE>>>(gpu_tree, TAILLE);
39     cudaMemcpy(tree, gpu_tree, 2*TAILLE*sizeof(float), cudaMemcpyDeviceToHost);
40     printf("%4.0f", tree[1]);
41     printf("\n");
42 }
```

① synchronisation du remplissage du tableau partagé

② synchronisation pour le passage du niveau au niveau suivant de l'arbre

③ synchronisation inutile : redondante avec celle de fin du while

On synchronise le travail des threads pour garantir que :

- ♦ lorsque l'on commence le travail, les données nécessaires soient bien présentes dans le tableau ❶ ;
- ♦ lors du passage d'un niveau de l'arbre au niveau suivant pour disposer de toutes les valeurs calculées ❷ ;
- ♦ enfin, la synchronisation ❸ est redondante dans la mesure où le `while` termine sur une synchronisation.

Si on essaye de passer `TAILLE` à la valeur 2048, on a un nouveau problème, le résultat est nul :

```
xterm
[ 0]
```

Si on regarde les caractéristiques de la carte GPGPU :

```
xterm
$ devicequery
CUDA Device Query...
There are 1 CUDA devices.

CUDA Device #0
Major revision number:      6
Minor revision number:      1
Name:                       GeForce GTX 1060 6GB
Total global memory:        2076508160
Total shared memory per block: 49152
Total registers per block:   65536
Warp size:                   32
Maximum memory pitch:       2147483647
Maximum threads per block:   1024
Maximum dimension 0 of block: 1024
Maximum dimension 1 of block: 1024
Maximum dimension 2 of block: 64
Maximum dimension 0 of grid: 2147483647
Maximum dimension 1 of grid: 65535
Maximum dimension 2 of grid: 65535
Clock rate:                  1835000
Total constant memory:       65536
Texture alignment:           512
Concurrent copy and execution: Yes
Number of multiprocessors:    10
Kernel execution timeout:     Yes
```

Le nombre limite de threads est dépassé avec 2048 car il est limité à 1024.

Le kernel n'a pas été lancé, et on a juste la valeur 0 d'initialisation.

Si on veut aller plus loin, il nous faut utiliser **plusieurs blocs**.

D'où, une nouvelle version :

```
#include <stdio.h>
#include <cuda.h>
#define TAILLE 16384

float tree[2*TAILLE];

__global__ void mon_noyau(float *t, int r)
{
    int position;
    while(r>1)
    {
        position = threadIdx.x + blockIdx.x*1024 + r;
        if ((position%2) == 0)
        {
            t[position/2] = t[position] + t[position+1];
        }
        r = r/2;
        __syncthreads();
    }
}

int main(void)
{
    float *gpu_tree;
    int nb_blocs = TAILLE/1024 + 1;

    cudaMalloc((void **)&gpu_tree, TAILLE*2*sizeof(float));
    for(int i=TAILLE; i<TAILLE*2; i++)
        tree[i] = float(2);
    cudaMemcpy(gpu_tree, tree, 2*TAILLE*sizeof(float), cudaMemcpyHostToDevice);
    if (TAILLE>1024)
        mon_noyau<<nb_blocs, 1024>>>(gpu_tree, TAILLE);
    else
        mon_noyau<<1, TAILLE>>>(gpu_tree, TAILLE);
}
```

```

    cudaMemcpy(tree, gpu_tree, 2*TAILLE*sizeof(float), cudaMemcpyDeviceToHost);
    printf("[%4.0f]", tree[1]);
    printf("\n");
}

```

On abandonne la **mémoire partagée** car elle n'est **partageable** qu'au sein d'un **même bloc**.

Si on augmente la valeur de `TAILLE`, on a de nouveau une **instabilité** du résultat pour `TAILLE = 16384`, où le résultat oscille entre les valeurs 27328, 23888, 24704, 26048, 25088, 25536 ...

⇒ Il nous faut **synchroniser le travail des différents blocs** !

D'où une dernière version :

```

1 #include <stdio.h>
2 #include <cuda.h>
3
4 #define TAILLE 32768
5 float tree[2*TAILLE];
6
7 __global__ void mon_noyau(float *t, int r, int num_bloc)
8 {
9     int position;
10    while(r>1)
11    {
12        position = threadIdx.x + num_bloc*1024 + r;
13        if ((position%2) == 0)
14        {
15            t[position/2] = t[position] + t[position+1];
16        }
17        r = r/2;
18        __syncthreads();
19    }
20 }
21
22 int main(void)
23 {
24     float *gpu_tree;
25     int t = TAILLE;
26     int nb_blocs = (TAILLE+1023)/1024;
27
28     printf("nb blocs %d\n", nb_blocs);
29     cudaMalloc((void **)&gpu_tree, TAILLE*2*sizeof(float));
30     for(int i=TAILLE; i<TAILLE*2; i++)
31         tree[i] = float(2);
32     cudaMemcpy(gpu_tree, tree, 2*TAILLE*sizeof(float), cudaMemcpyHostToDevice);
33     if (TAILLE>1024)
34     {
35         while(t>1)
36         {
37             for(int i = 0; i<nb_blocs; i++)
38                 mon_noyau<<<1, 1024>>>(gpu_tree, TAILLE, i);
39             t=t/2;
40         }
41     }
42     else
43         mon_noyau<<<1, TAILLE>>>(gpu_tree, TAILLE, 0);
44     cudaMemcpy(tree, gpu_tree, 2*TAILLE*sizeof(float), cudaMemcpyDeviceToHost);
45     printf("[%4.0f]", tree[1]);
46     printf("\n");
47 }

```

▷ en ligne 33, on contrôle la valeur de `TAILLE` :

- ◇ si elle est inférieure à 1024, on appelle le kernel avec un seul bloc ;
- ◇ si elle est supérieure à 1024 :
 - ★ on réalise la synchronisation avec une boucle `for` extérieure s'occupant de faire travailler un seul « bloc » sur les différentes données ;
 - ★ la synchronisation se fait par lancements successifs de kernels ;
 - ★ une instruction `cudaDeviceSynchronize()` peut être nécessaire après chaque lancement de kernel, mais elle n'est pas nécessaire en général : la fin d'exécution normale du kernel entraîne le retour vers le programme hôte, mais cela peut servir pour détecter des erreurs si cette synchronisation échoue (le kernel s'arrête à cause d'un `timeout` par exemple).

Remarque

CUDA v9 introduit la notion de « *cooperative group* » de threads que l'on peut synchroniser. Le groupe peut contenir des toutes les threads de la grille $\Rightarrow$ on peut **synchroniser** les **threads de tous les blocs**.

```
#include <cooperative_groups.h>

grid_group g = this_grid();
g.sync();
```