Algorithmes fondamentaux

Opération *Scan*

Définition d'une opération *scan*

- un tableau en entrée
- un opérateur binaire et associatif
- un élément neutre [`I op a = a`]

Scan inclusif ou exclusif

```
int acc = élément_neutre;

for (i=0 ; i<n; i++) {
        acc = acc op in[i];
        out[i] = acc;
}
```

scan **inclusif**

```
int acc = élément_neutre;

for (i=0 ; i<n; i++) {
        out[i] = acc;
        acc = acc op in[i];
}
```

scan **exclusif**

Opération *Scan*

Exemple

- entrée : [1 2 3 4 5 6 7 8]
- opérateur : +
- élément neutre : 0

- sortie du *scan* **inclusif** :

  [1 3 6 10 15 21 28 36]

- sortie du *scan* **exclusif** :

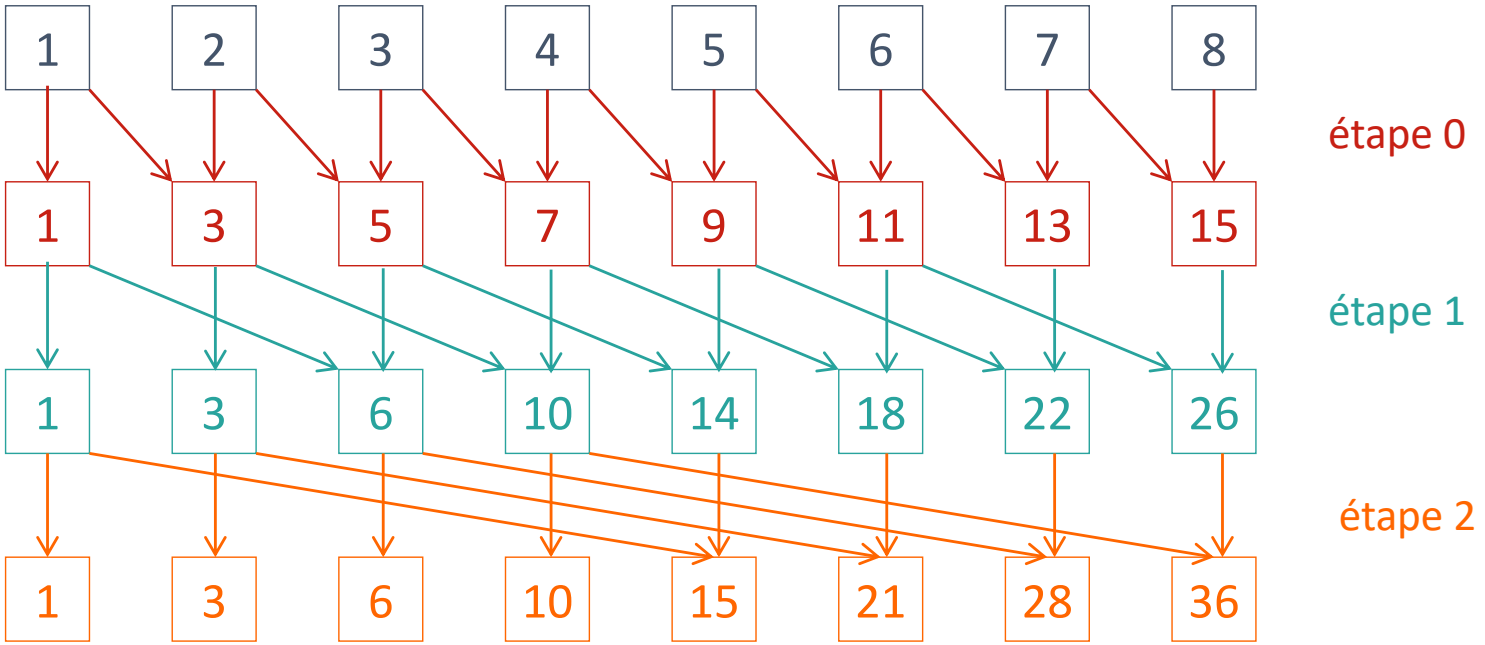
  [0 1 3 6 10 15 21 28]

```
int acc = élément_neutre;

for (i=0 ; i<n; i++) {
        acc = acc op in[i];
        out[i] = acc;
}
```

```
int acc = élément_neutre;

for (i=0 ; i<n; i++) {
        out[i] = acc;
        acc = acc op in[i];
}
```

Opération *Scan inclusif*

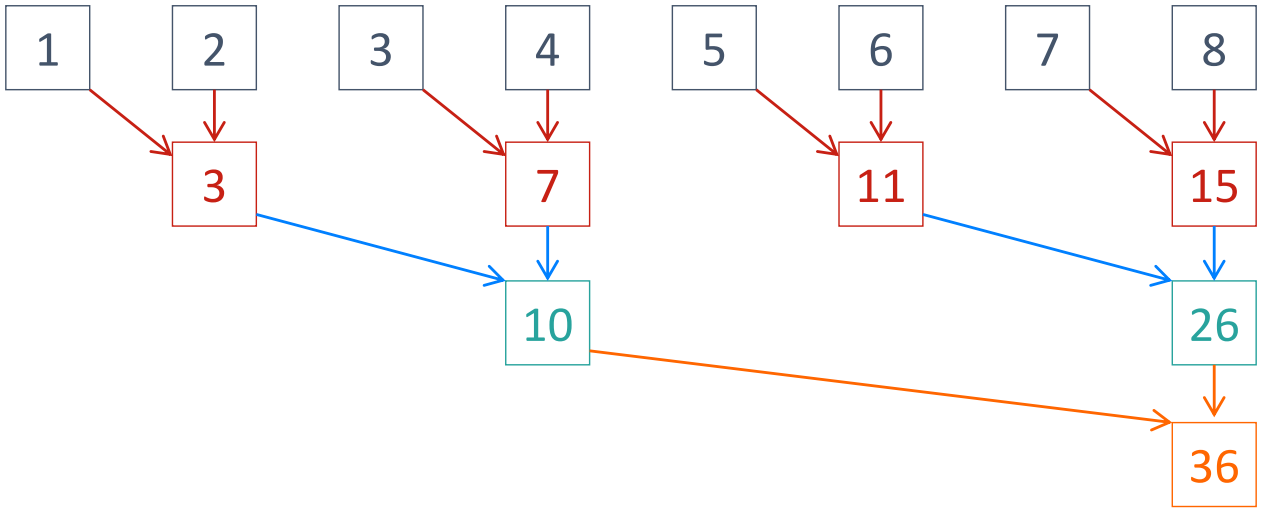
Algorithme de Hillis/Steele

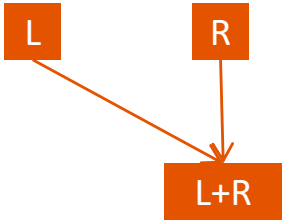


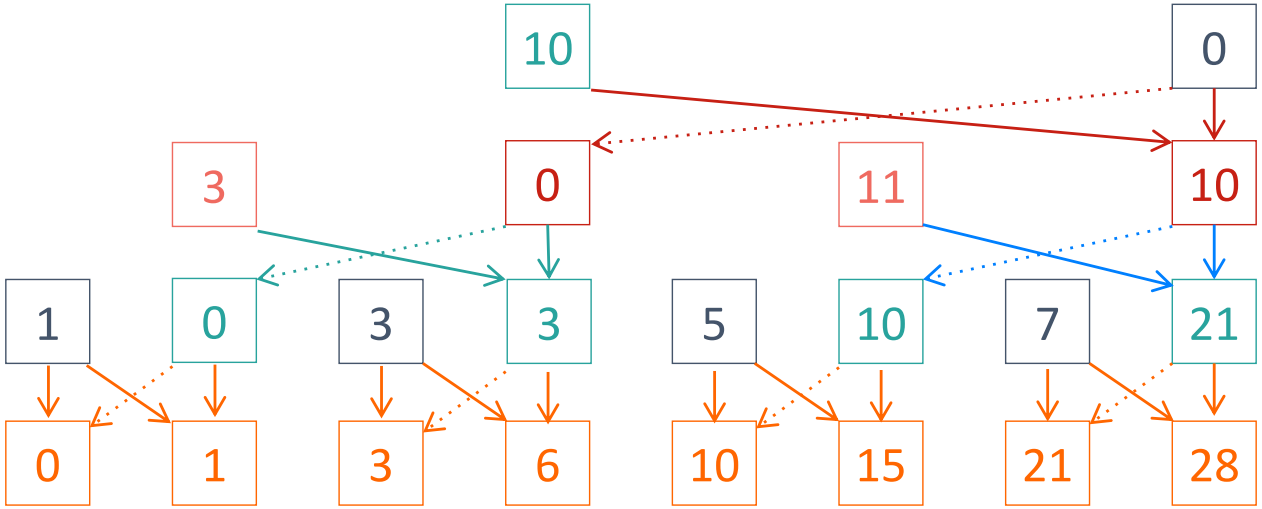
Etape i : chaque thread de numéro > 2^i ajoute à sa somme locale celle de son voisin qui se trouve 2^i positions à gauche

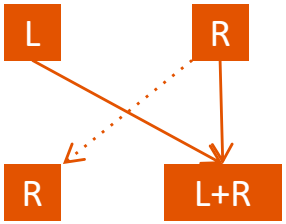
Opération *Scan exclusif*

Algorithme de Blelloch



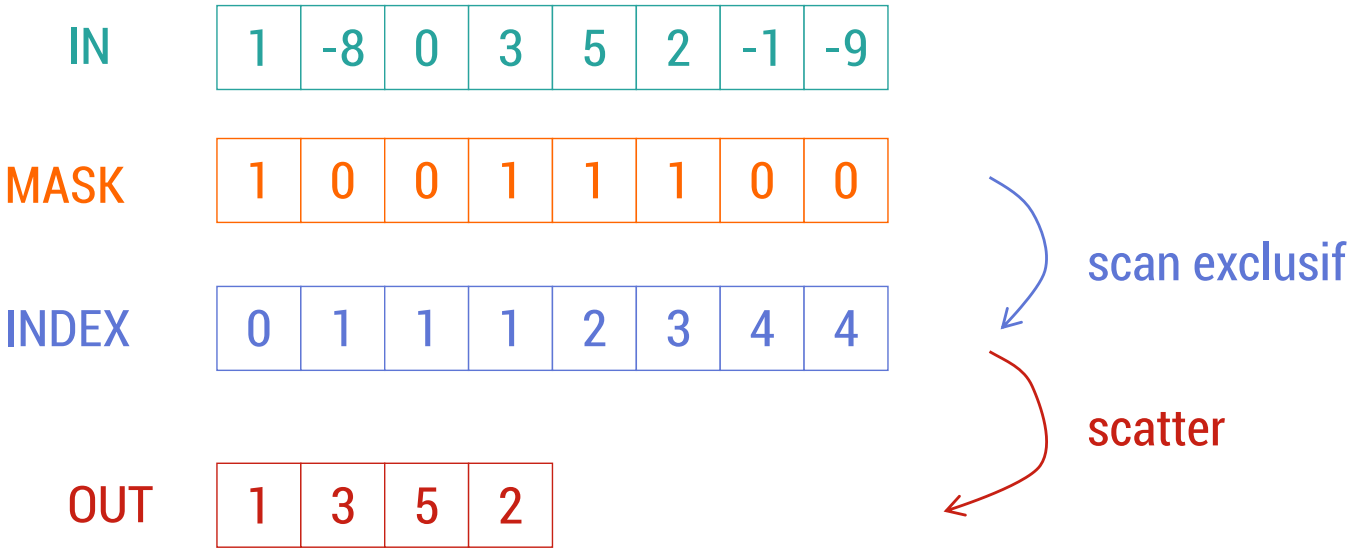






Exercice : réduction de flot

A partir d'un tableau *IN* de *n* éléments, générer un tableau *OUT* qui contient tous les éléments de *IN* strictement supérieurs à 0, dans le même ordre.

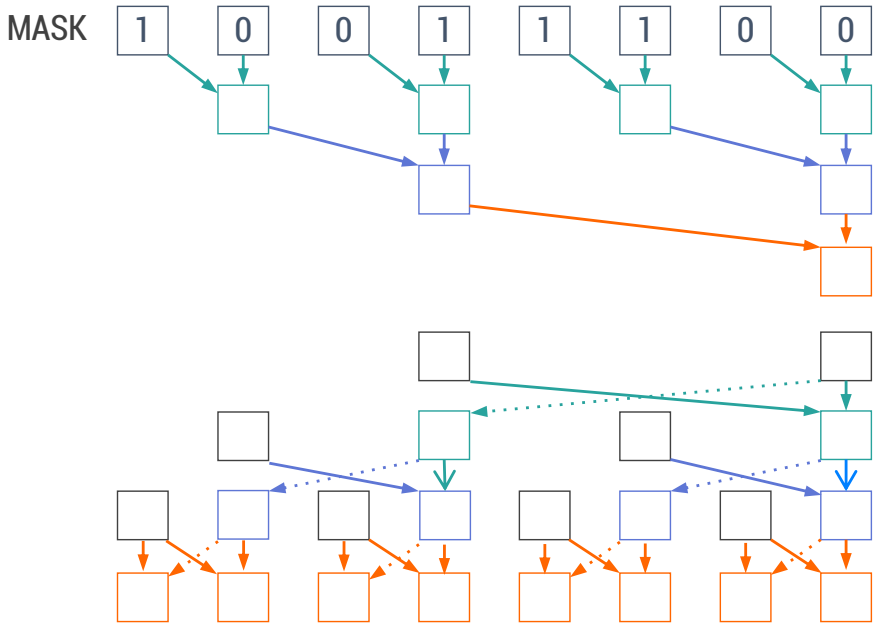


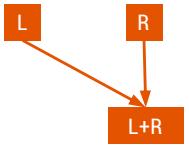
```
__global__ void stream_compaction(int *d_out, int *d_in, int n){
    extern __shared__ int mask[] ;
    int tid = threadIdx.x ;
    int offset ;
    int my_mask;
    int a, b, tmp, my_data;
    my_data = d_in[tid];
    // création du masque
    if (tid < n){
        if (my_data> 0)
           my_mask = 1 ;
        else
           my_mask = 0 ;
        mask[tid] = my_mask ;
    }
    // scan du masque selon l'algorithme de Blelloch
```

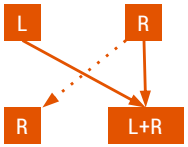
Scan du masque

IN

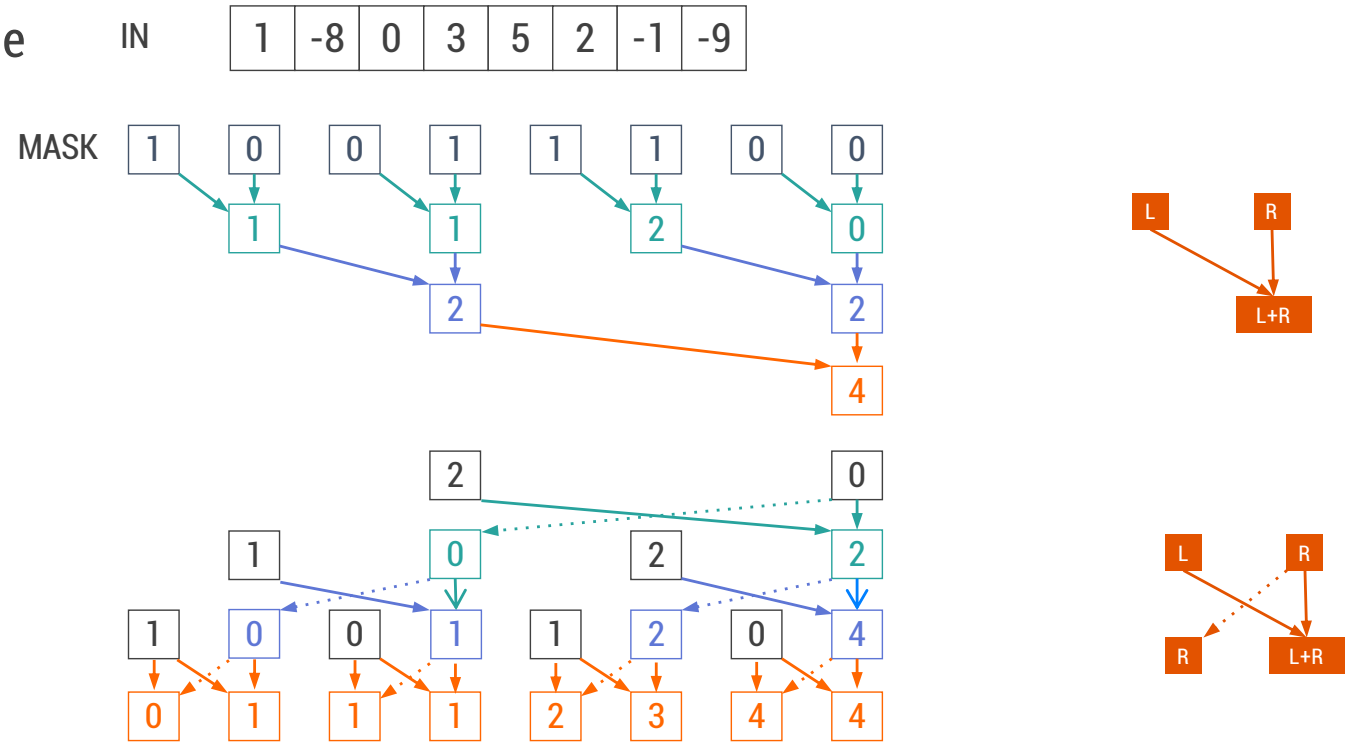
1	-8	0	3	5	2	-1	-9







Scan du masque

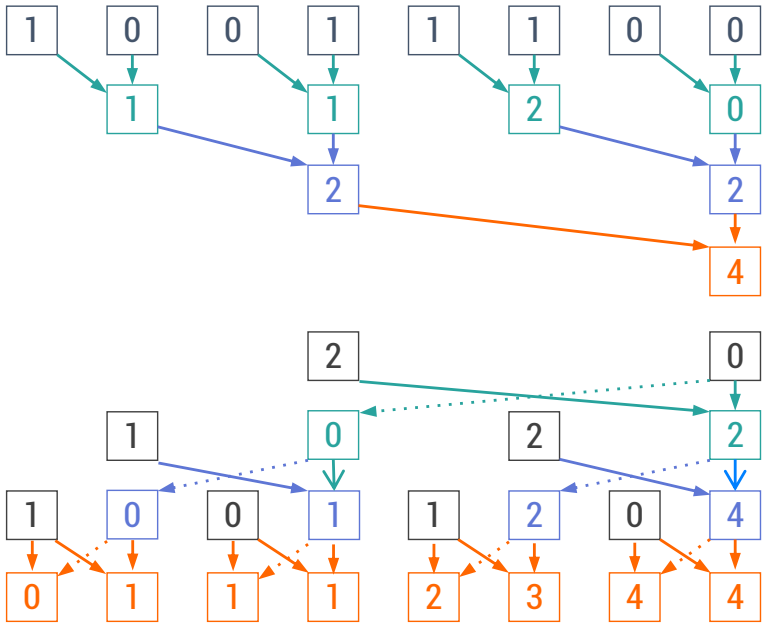


Scan du masque

	offset = 1	offset = 2	offset = 4
th0	mask[1] += mask[0]	mask[3] += mask[1]	mask[7] += mask[3]
th1	mask[3] += mask[2]	mask[7] += mask[5]	
th2	mask[5] += mask[4]		
th3	mask[7] += mask[6]		

mask[f1(offset, tid)] += mask[f2(offset, tid)]

Scan du masque



	offset = 1	offset = 2	offset = 4
th0	mask[1] += mask[0]	mask[3] += mask[1]	mask[7] += mask[3]
th1	mask[3] += mask[2]	mask[7] += mask[5]	
th2	mask[5] += mask[4]		
th3	mask[7] += mask[6]		

mask[offset * (2*tid+2) -1] += mask[offset * (2*tid+1) – 1]

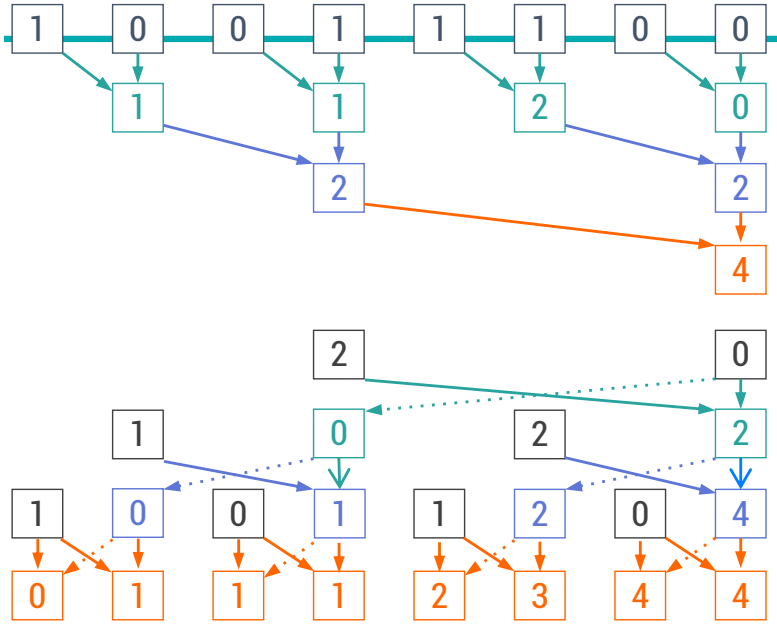
```
__global__ void stream_compaction(int *d_out, int *d_in, int n){
    extern __shared__ int mask[] ;
    int tid = threadIdx.x ;
    int offset ;
    int my_mask;
    int a, b, tmp, my_data;
    my_data = d_in[tid];
    // création du masque
    …
    // scan du masque selon l'algorithme de Blelloch – 1ère partie
    offset = 1 ;
    for (int d=n>>1 ; d>0 ; d>>=1) {
        __syncthreads() ;
        if (tid < d) {
            a = offset * (2*tid+1) – 1 ;
            b = offset * (2*tid+2) – 1 ;
            if (b<n) mask[b] += mask[a] ;
        }
        offset *= 2 ;
    }
    __syncthreads() ;
    // scan du masque selon l'algorithme de Blelloch – 2ème partie
    …
```

Scan du masque

	offset = 4	offset = 2	offset = 4
th0	tmp = mask[7] mask[7] += mask[3] mask[3] = tmp	tmp = mask[7] mask[7] += mask[5] mask[5] = tmp	tmp = mask[7] mask[7] += mask[6] mask[6] = tmp
th1		tmp = mask[3] mask[3] += mask[1] mask[1] = tmp	tmp = mask[5] mask[5] += mask[4] mask[4] = tmp
th2			tmp = mask[3] mask[3] += mask[2] mask[2] = tmp
th3			tmp = mask[1] mask[1] += mask[0] mask[0] = tmp

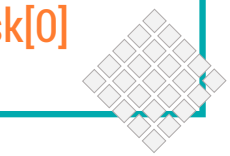
```
tmp = mask[ g1(offset, tid) ]
mask[ g1(offset, tid) ] += mask[ g2(offset, tid) ]
mask[ g2(offset, tid) ] = tmp
```

Scan du masque



	offset = 4	offset = 2	offset = 4
th0	tmp = mask[7] mask[7] += mask[3] mask[3] = tmp	tmp = mask[7] mask[7] += mask[5] mask[5] = tmp	tmp = mask[7] mask[7] += mask[6] mask[6] = tmp
th1		tmp = mask[3] mask[3] += mask[1] mask[1] = tmp	tmp = mask[5] mask[5] += mask[4] mask[4] = tmp
th2			tmp = mask[3] mask[3] += mask[2] mask[2] = tmp
th3			tmp = mask[1] mask[1] += mask[0] mask[0] = tmp

```
tmp = mask[ offset * (2*tid +2) – 1 ]
mask[ offset * (2*tid +2) – 1] += mask[ offset * (2*tid +1) – 1 ]
mask[ offset * (2*tid +1) – 1 ] = tmp
```

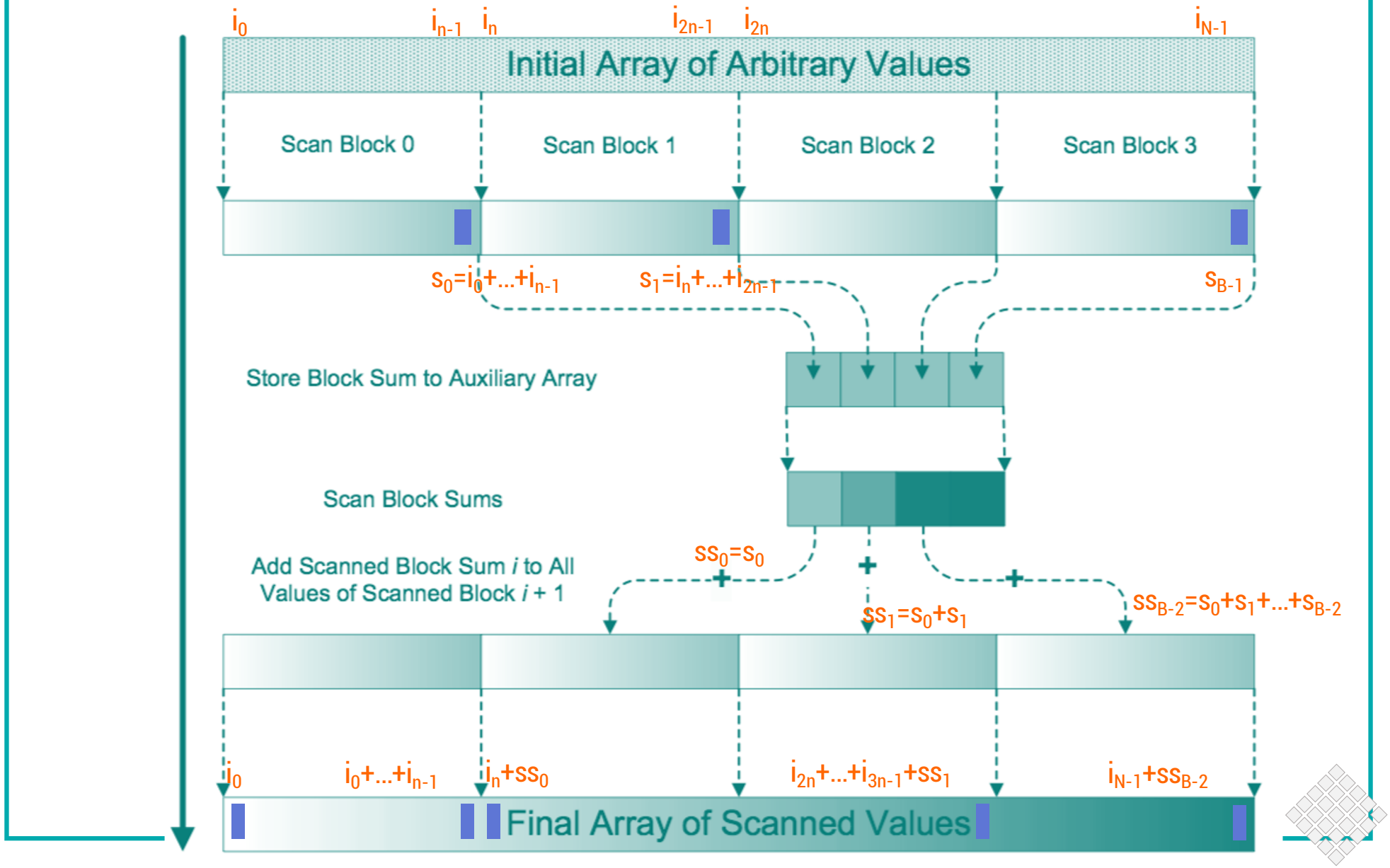


```
// scan du masque selon l'algorithme de Blelloch – 2ème partie
if (tid == 0)
     mask[n-1] = 0 ;
for (int d=1 ; d<n ; d *= 2){
     offset >>=1 ;
     _syncthreads() ;
     if (tid < d) {
          a = offset * (2*tid+1) – 1 ;
          b = offset * (2*tid+2) – 1 ;
          tmp = mask[b] ;
          mask[b] += mask[a] ;
          mask[a] = tmp ;
     }
}
__syncthreads() ;

// scatter
…
```

```
// scan du masque selon l'algorithme de Blelloch – 2ème partie
if (tid == 0)
     mask[n-1] = 0 ;
for (int d=1 ; d<n ; d *= 2){
     offset >>=1 ;
     _syncthreads() ;
     if (tid < d) {
          a = offset * (2*tid+1) – 1 ;
          b = offset * (2*tid+2) – 1 ;
          tmp = mask[b] ;
          mask[b] += mask[a] ;
          mask[a] = tmp ;
     }
}
__syncthreads() ;

// scatter
if (my_mask == 1){
     d_out[mask[tid]] = my_data ;
}
```

EXERCICE

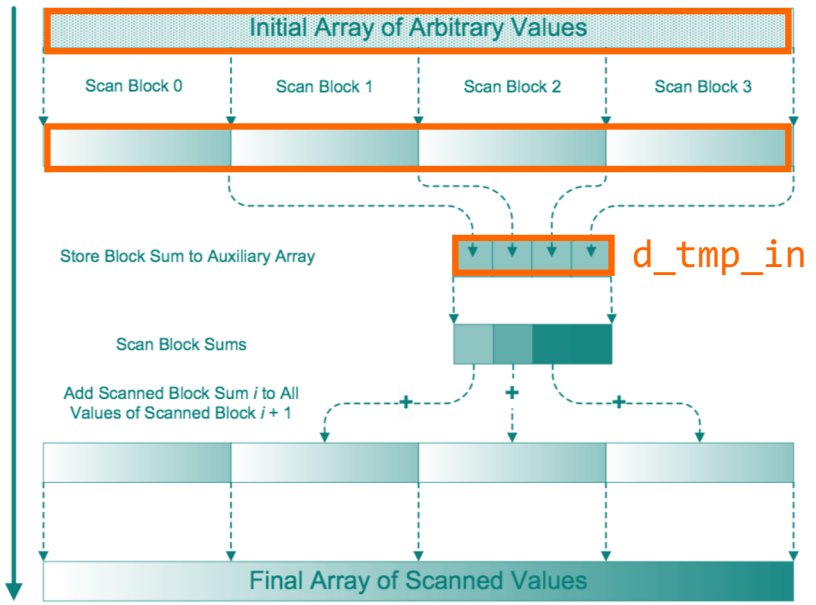
Scan pour des entrées de longueur quelconque



```
__device__ void scanBlock(int *d_out, int *d_in, int n){
      // scan sur un bloc (n <= BLOCK_SIZE) [algo de Blelloch => scan exclusif]
      int tid = threadIdx.x ;
      int offset, a, b, tmp;
      offset = 1 ;
      d_out[tid] = d_in[tid];
      __syncthreads();
      for (int d=n>>1 ; d>0 ; d>>=1) {
           __syncthreads() ;
           if (tid < d) {
               a = offset * (2*tid+1) - 1 ;
               b = offset * (2*tid+2) - 1 ;
               if (b<n) d_out[b] += d_out[a] ;
           }
           offset *= 2 ;
      }
      __syncthreads() ;
      if (tid == 0)
           d_out[n-1] = 0 ;
      for (int d=1 ; d<n ; d *= 2){
           offset >>=1 ;
           __syncthreads() ;
           if (tid < d) {
               a = offset * (2*tid+1) - 1 ;
               b = offset * (2*tid+2) - 1 ;
               tmp = d_out[b] ;
               d_out[b] += d_out[a] ;
               d_out[a] = tmp ;
           }
      }
      __syncthreads();
      d_out[tid] += d_in[tid];  // pour que le scan soit inclusif
}
```

Cette **fonction**, appelable depuis un kernel, calcule la contribution d'**un** thread au calcul du ***scan inclusif*** (+) sur un tableau.

```
__device__ void scanBlock(int *d_out, int *d_in, int n){
        … // scan sur un bloc (n <= BLOCK_SIZE)
}

__global__ void scanKernel_1(int *d_out, int *d_in, int *d_tmp, int size){
}
```



d_in

d_out

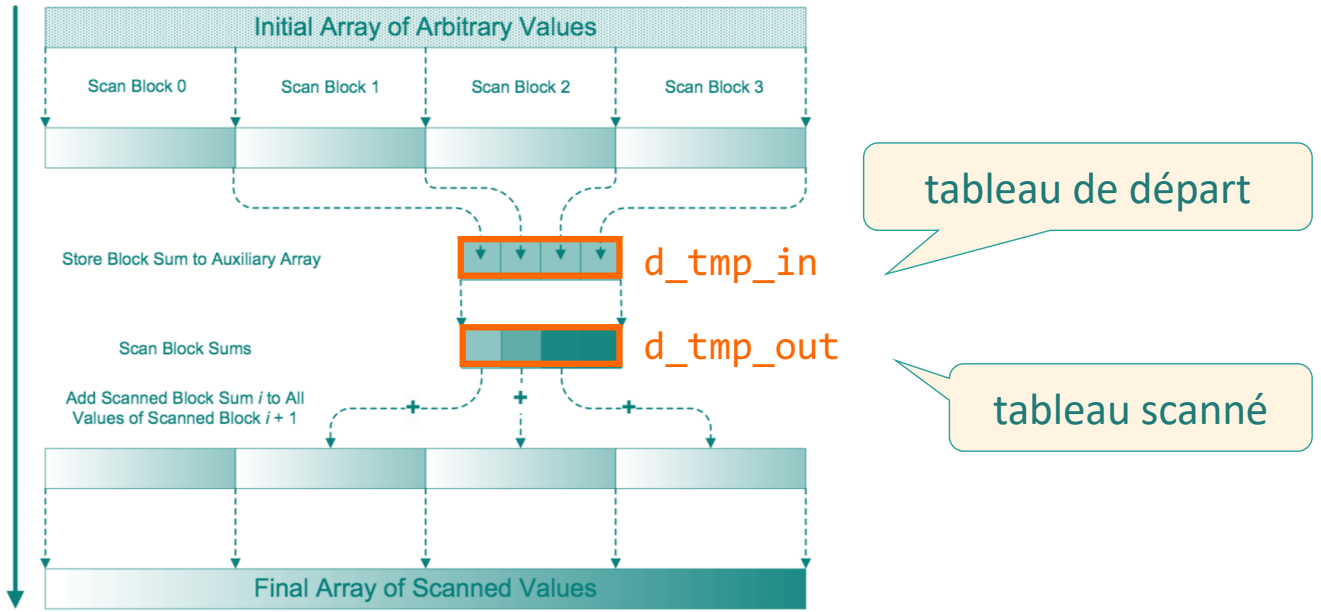
d_tmp_in

tableau de départ

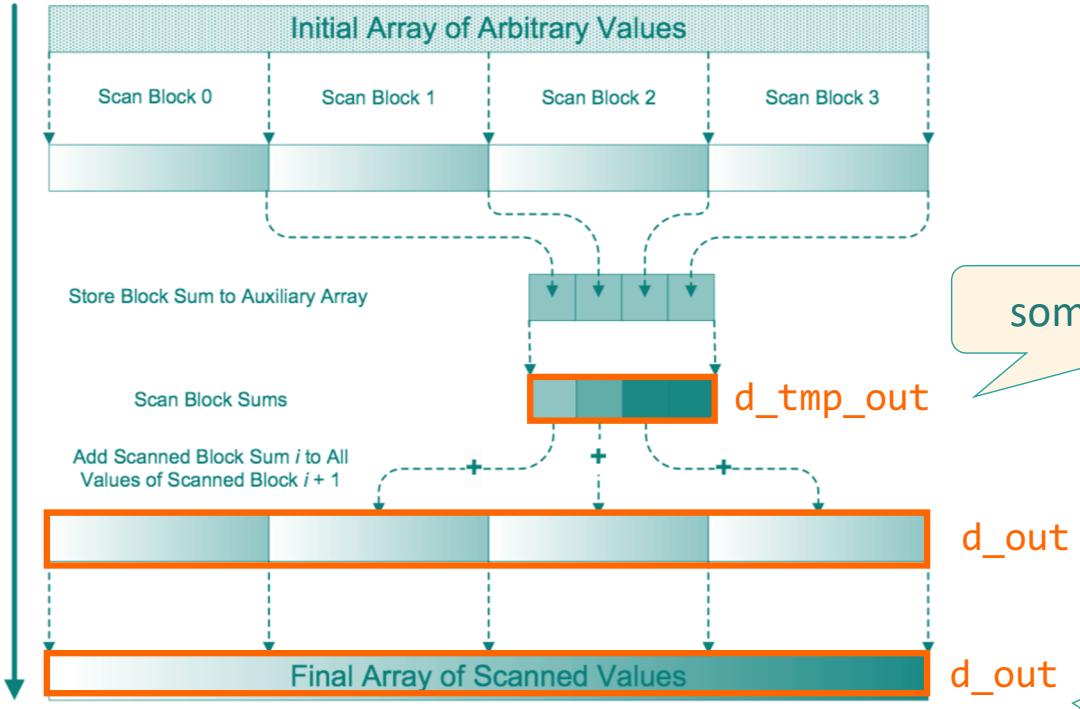
tableau scanné par blocs

sommes des blocs

```
__device__ void scanBlock(int *d_out, int *d_in, int n){
        … // scan sur un bloc (n <= BLOCK_SIZE)
}

__global__ void scanKernel_1(int *d_out, int *d_in, int *d_tmp, int size){
}

__global__ void scanKernel_2(int *d_out, int *d_in, int size){
}
```



```
__device__ void scanBlock(int *d_out, int *d_in, int n){
        … // scan sur un bloc (n <= BLOCK_SIZE)
}

__global__ void scanKernel_1(int *d_out, int *d_in, int *d_tmp, int size){
}

__global__ void scanKernel_2(int *d_out, int *d_in, int size){
}

__global__ void scanKernel_3(int *d_out, int *d_tmp, int size){
}
```

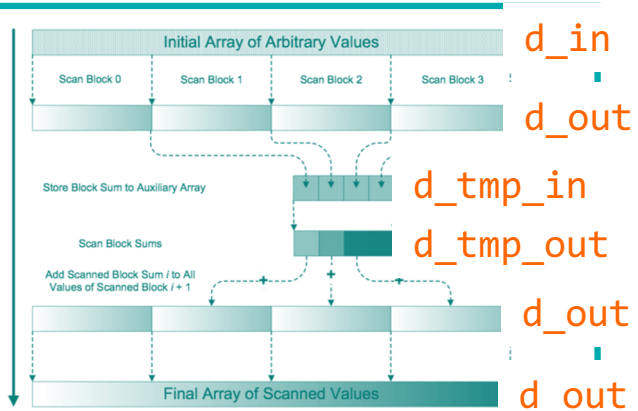


sommes des blocs

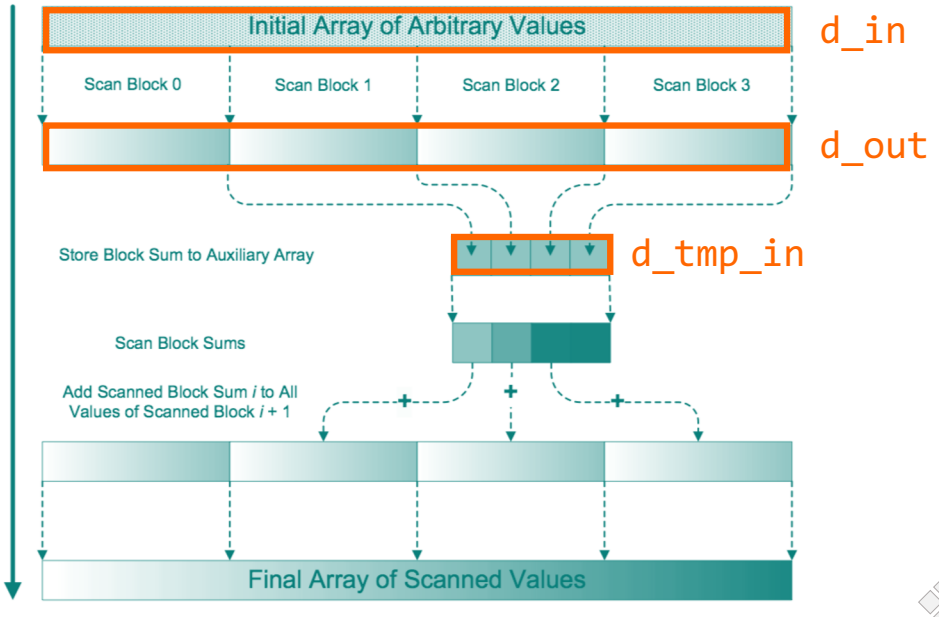
tableau scanné par blocs

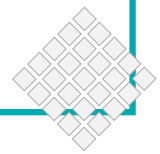
scan complet du tableau initial

```
#define BLOCK_SIZE 1024

void scan(int *out, int *in, int size){
    int *d_in, *d_out, *d_tmp_in, *d_tmp_out;
    int *tmp;
    int num_blocks = size / BLOCK_SIZE;
    if (size % BLOCK_SIZE)
        num_blocks++;

    cudaMalloc((void **)&d_in, size*sizeof(int));
    cudaMalloc((void **)&d_out, size*sizeof(int));
    cudaMalloc((void **)&d_tmp_in, num_blocks*sizeof(int));
    cudaMalloc((void **)&d_tmp_out, num_blocks*sizeof(int));

    cudaMemcpy(d_in, in, size*sizeof(int), cudaMemcpyHostToDevice);

    scanKernel_1<<<num_blocks, BLOCK_SIZE>>>(d_out, d_in, d_tmp_in, size);

    scanKernel_2<<<1, num_blocks>>>(d_tmp_out, d_tmp_in, num_blocks);

    scanKernel_3<<<num_blocks-1, BLOCK_SIZE>>>(d_out, d_tmp_out, size);

    cudaMemcpy(out, d_out, size*sizeof(int), cudaMemcpyDeviceToHost);

    cudaFree(d_in); cudaFree(d_out); cudaFree(d_tmp_in); cudaFree(d_tmp_out);
}
```

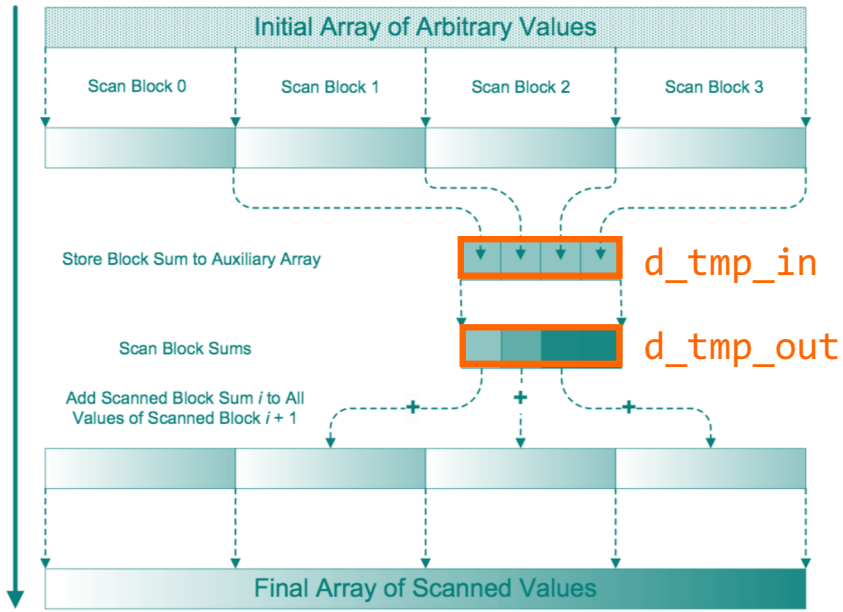


```
__device__ void scanBlock(int *d_out, int *d_in, int n){
        … // scan sur un bloc (n <= BLOCK_SIZE)
}
__global__ void scanKernel_1(int *d_out, int *d_in, int *d_tmp, int size){




}
```





```
__device__ void scanBlock(int *d_out, int *d_in, int n){
        … // scan sur un bloc (n <= BLOCK_SIZE)
}
__global__ void scanKernel_2(int *d_out, int *d_in, int size){



}
```



```
__device__ void scanBlock(int *d_out, int *d_in, int n){
        … // scan sur un bloc (n <= BLOCK_SIZE)
}
__global__ void scanKernel_3(int *d_out, int *d_tmp, int size){




}
```