Les différentes mémoires du GPU

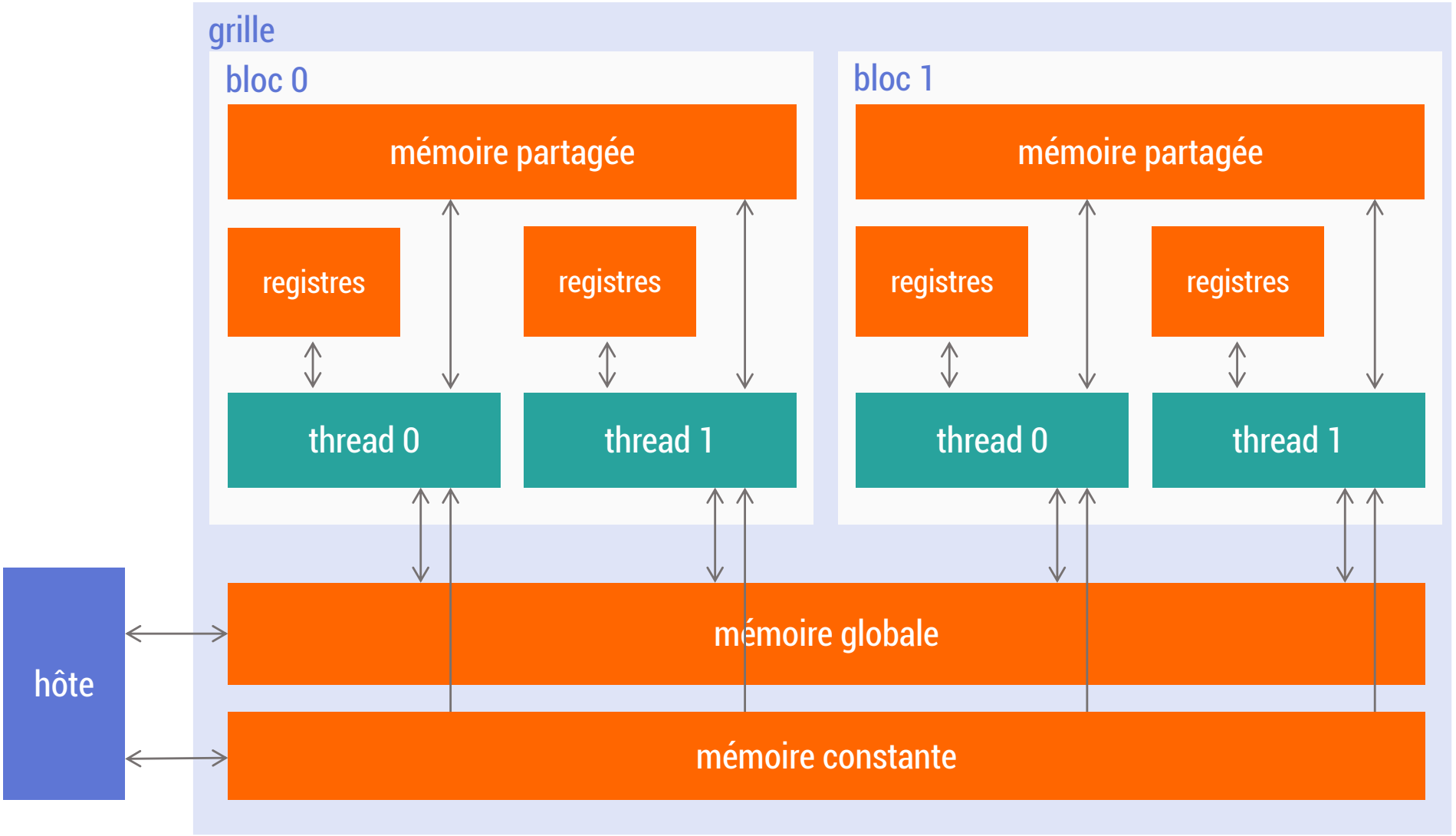
Les accès mémoire limitent les performances

Exemple : kernel qui calcule le produit de matrices

```
for (k=0 ; k<N; k++)
        res += d_A[i*N + k] * d_B[k*N + j];
```

- CGMA = Compute to Global Memory Access ratio
  - dans cet exemple, = 1:1
- impact sur les performances
  - bande passante mémoire = 200 GO/s → 50 G(float)/s
    → 50 GFLOPS (à comparer aux 1500 GFLOPS crête)
  - conclusion : il faut augmenter le ratio CGMA

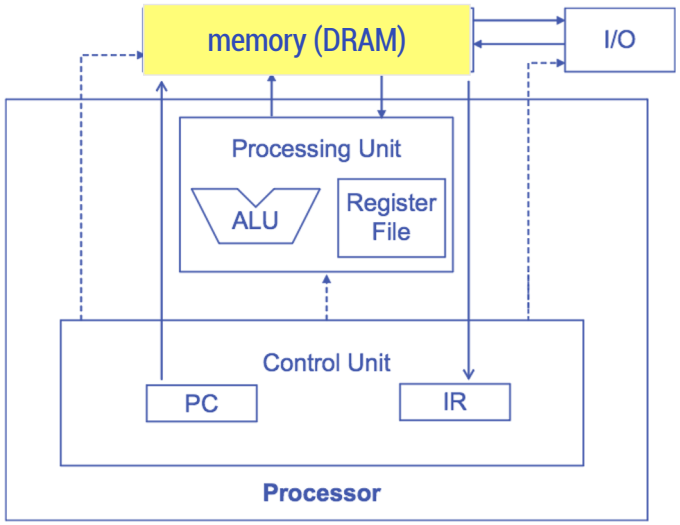
Mémoires du GPU

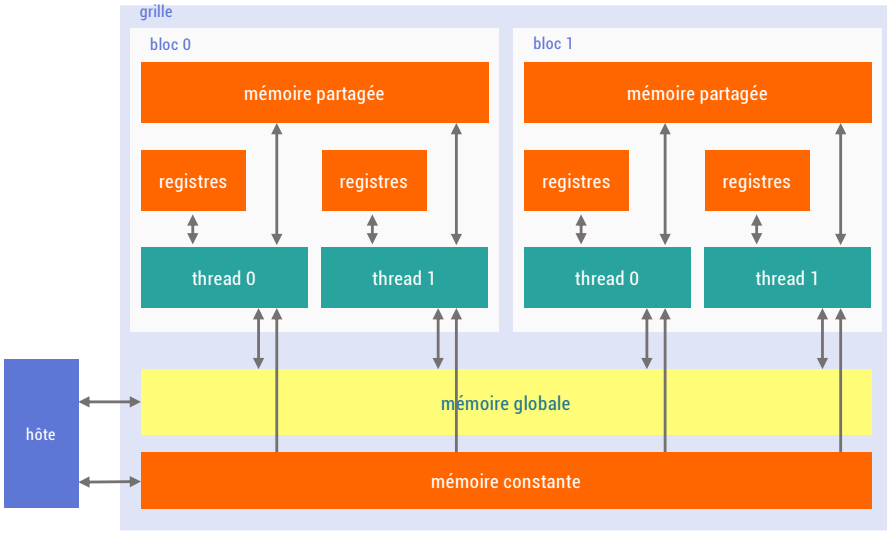


Mémoires du GPU

Mémoire globale

- accessible en lecture/écriture par le CPU (hôte)
  - via des fonctions de l'API
- analogie architecture de Von Neumann : DRAM externe
  - latence longue, bande passante limitée

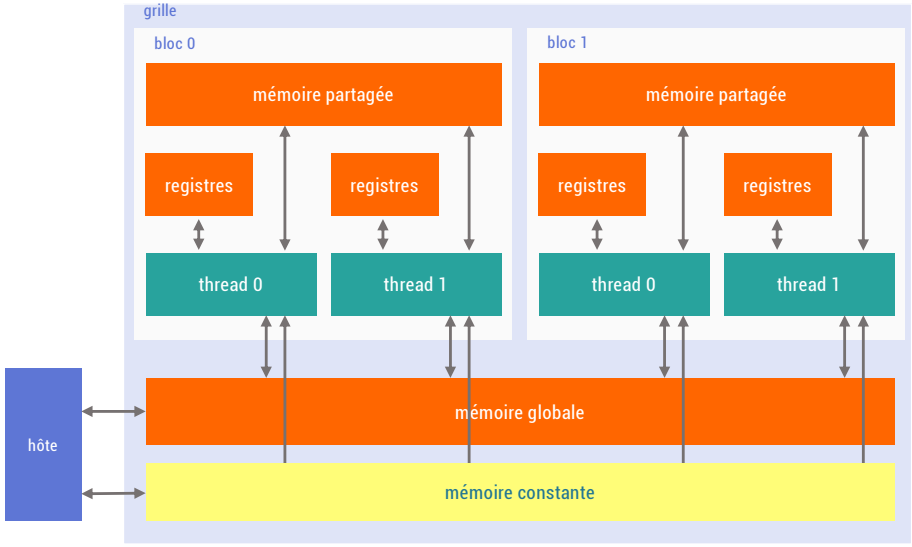




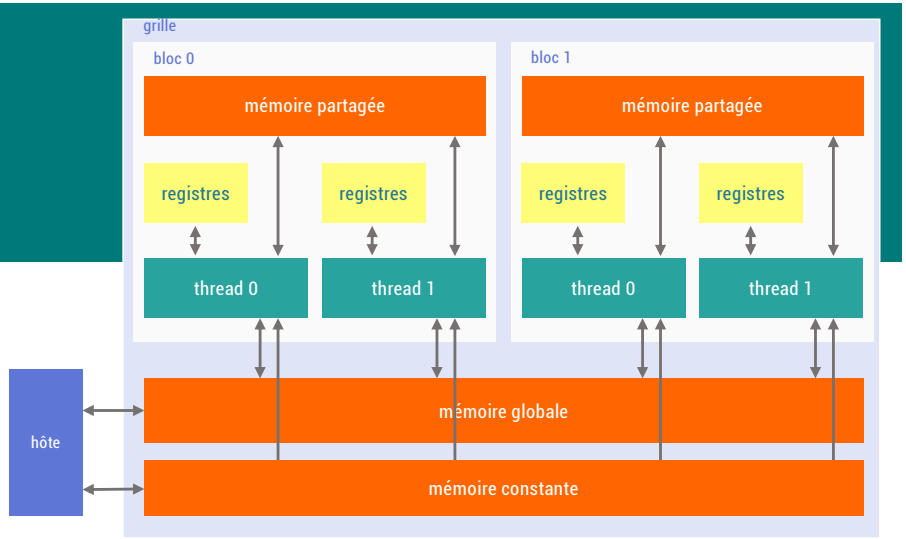
Mémoires du GPU

Mémoire constante

- accessible en lecture/écriture par le CPU (hôte)
  - via des fonctions de l'API
- accessible en lecture seulement par le GPU
  - latence courte et grande bande passante
    quand tous les threads accèdent simultanément à la même adresse

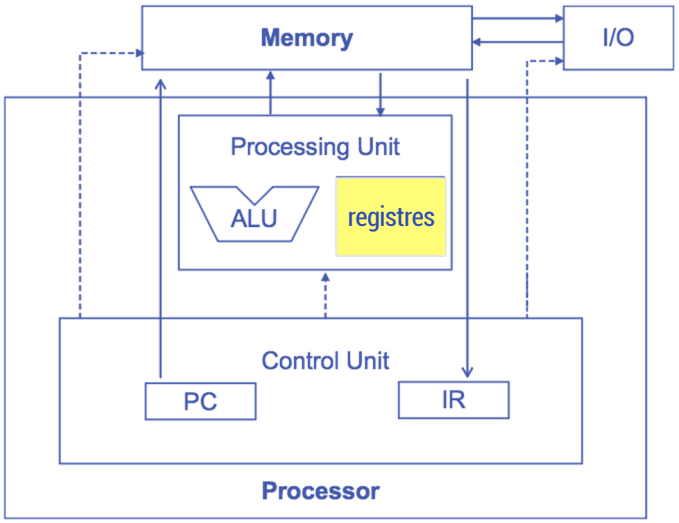


Mémoires du GPU



Registres

- accès parallèles très rapides
- privés pour chaque thread
  - variables privées utilisées fréquemment

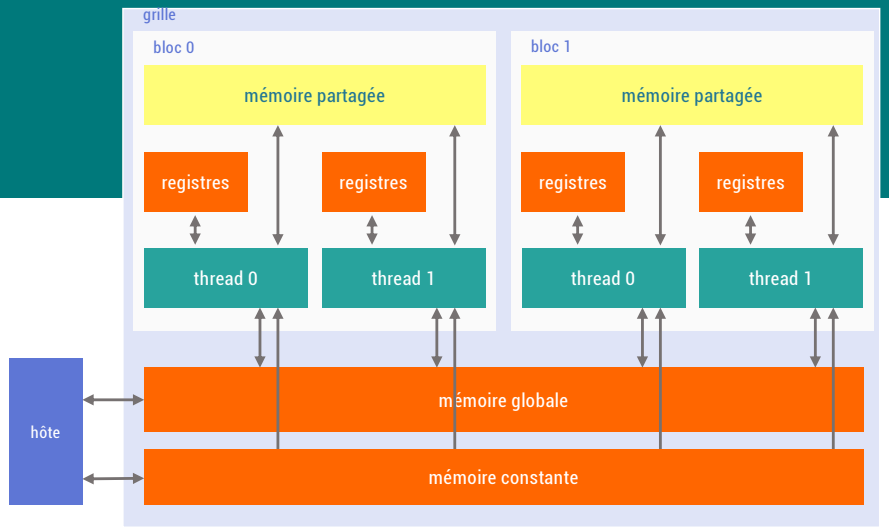


- grande bande passante
- augmente le ratio CGMA
- moins d'instructions
- consommation d'énergie moindre

mais ...

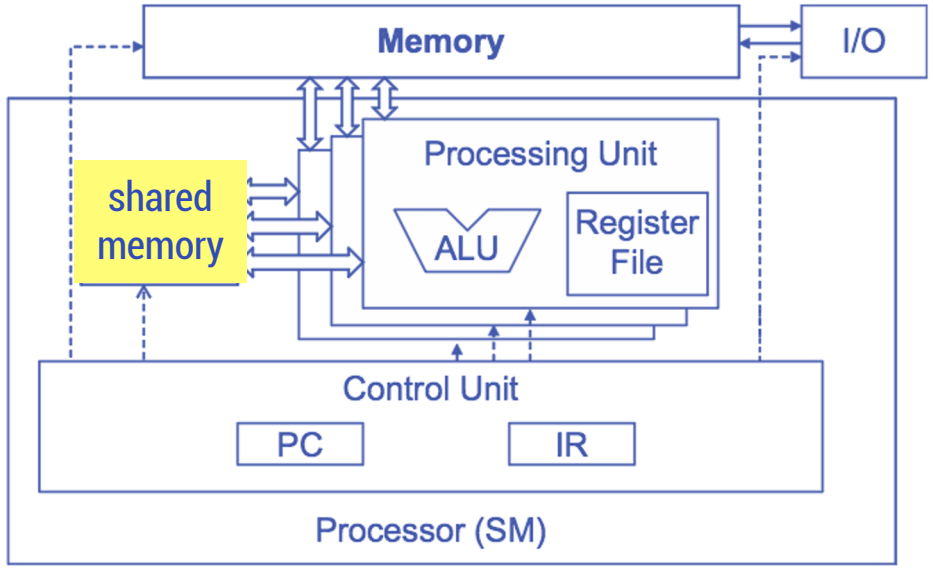
- nombre de registres limité

Mémoires du GPU



Mémoire partagée

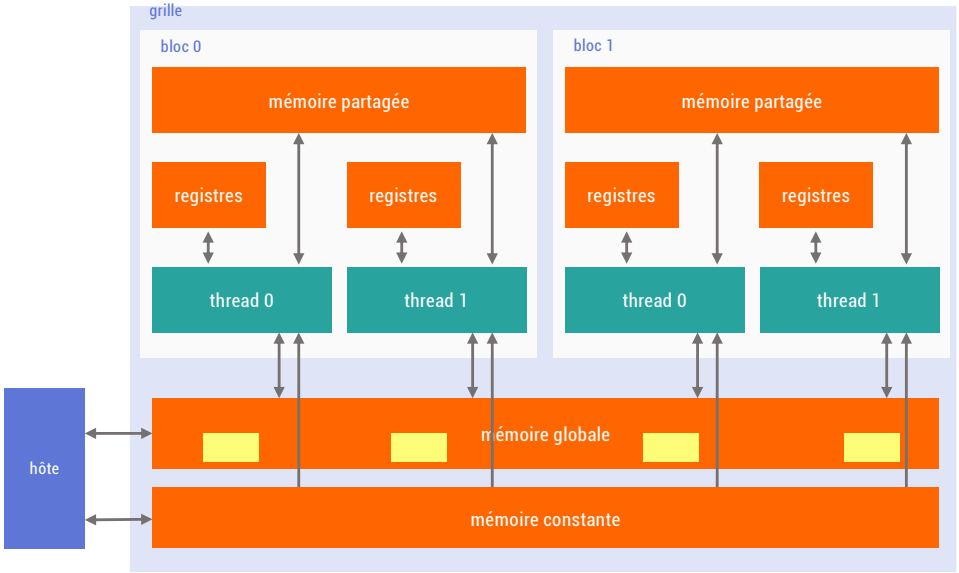
- mémoire scratchpad on-chip
  - accès mémoire (load/store)
  - latence un peu plus longue que pour un registre
  - bande passante un peu plus faible
- partagée par tous les threads du même bloc
  - ils peuvent partager leurs entrées, leurs résultats intermédiaires



Mémoires du GPU

Mémoire locale

- variables privées de type tableau
  - ne peuvent pas être rangées dans des registres
- zone allouée au thread dans la mémoire globale
  - accès lent et conflits fréquents
  - copie privée de la variable pour chaque thread
  - rarement utilisée



Qualifieurs de variables

déclaration	mémoire	portée	durée de vie
`int var;`	registre	thread	kernel
`int var[100];`	locale	thread	kernel
`__device__ __shared__ int var;`	partagée	bloc	kernel
`__device__ int var;`	globale	grille	application
`__device__ __constant__ int var;`	constante	grille	application

Les pointeurs ne peuvent pointer que sur des zones allouées ou déclarées dans la mémoire globale

- zone allouée par l'hôte, pointeur passé au kernel en paramètre

```
__global__ void kernel(float *ptr){}
```

- pointeur **calculé** comme l'adresse d'une variable globale

```
float *ptr = &globalVar;
```

Allocation de mémoire partagée

Statique

```
#define BLOCK_SIZE_X 16
#define BLOCK_SIZE_Y 16

__global__ void kernel(…) {
        …
        __shared__ float shared_data[BLOCK_SIZE_X*BLOCK_SIZE_Y];
        …
}
```

Allocation de mémoire partagée

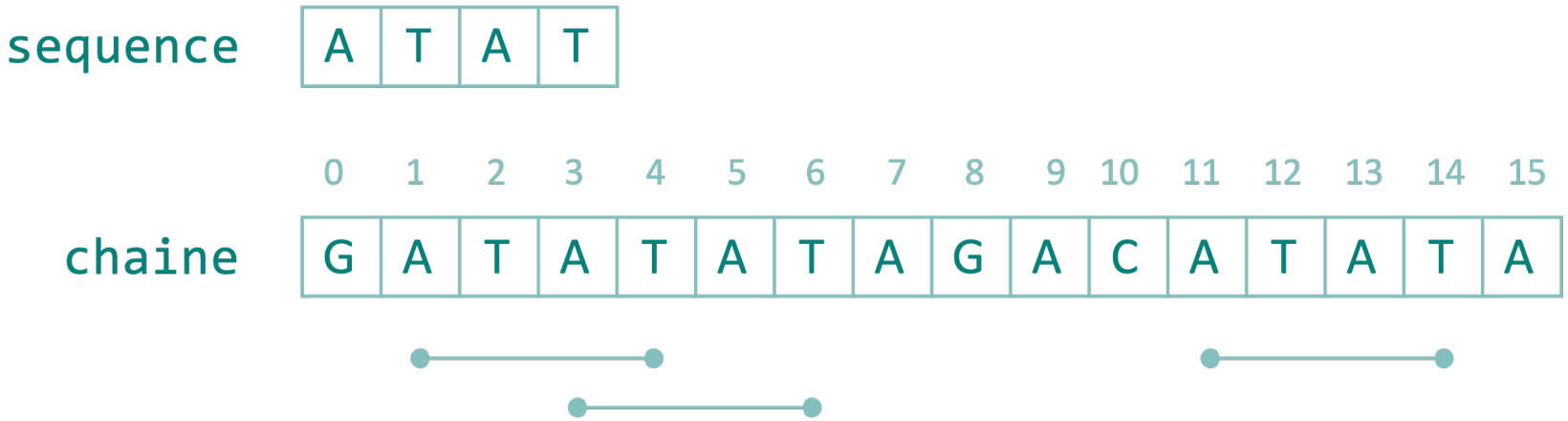
Dynamique

```
__global__ void kernel(…) {
        …
        extern __shared__ float *shared_data;
        …
}

int main(){
        int block_size_x = …;
        int block_size_y = …;
        int bytes = block_size_x * block_size_y * sizeof(float);
        // allocation dynamique de la mémoire partagée
        // lors de l'appel du kernel
        kernel<<<dimGrid, dimBlock, bytes>>>(…);
```

EXERCICE

Recherche d'une séquence dans une chaîne de ADN




```
bool Contient(unsigned char *sequence,
              unsigned char *chaine)
```

Cherche si une séquence donnée de 8 caractères est présente dans une chaîne de 16384 caractères
NB : seules les positions à partir desquelles on peut trouver la séquence complète sont considérées (il y en a 16377).

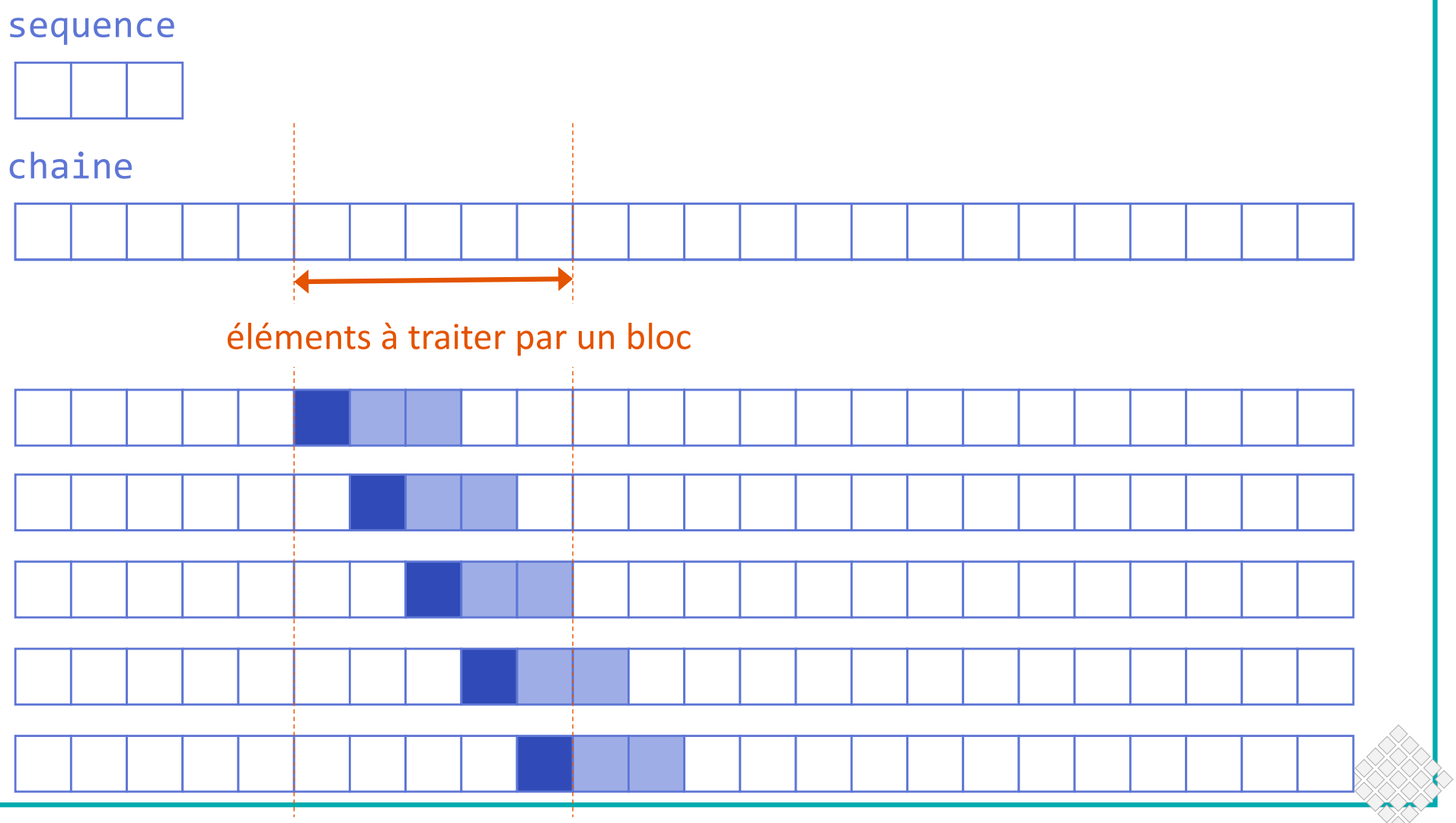
```
#define LONG_CHAINE 16384
#define LONG_SEQ 8
#define NUM_THDS 512

bool Contient(unsigned char *sequence, unsigned char *chaine){
    unsigned int num_blocks =
       (LONG_CHAINE-LONG_SEQUENCE+1 + NUM_THDS-1)/NUM_THDS) ;
    unsigned int ch_size = LONG_CHAINE*sizeof(char);
    unsigned int seq_size = LONG_SEQ*sizeof(char);
    bool presente = FALSE;
    bool *pres;
    unsigned char *ch, *seq;
    cudaMalloc((void **)&ch, ch_size);
    cudaMemcpy(ch, chaine, ch_size, CudaMemcpyHostToDevice);
    cudaMalloc((void **)&seq, seq_size);
    cudaMemcpy(seq, sequence, seq_size, CudaMemcpyHostToDevice);
    cudaMalloc((void **)&pres, sizeof(bool));
    cudaMemcpy(pres, &presente, sizeof(bool), CudaMemcpyHostToDevice);
    cherche<<<num_blocks, NUM_THDS>>>(ch, seq, pres);
    cudaMemcpy(presente, pres, sizeof(bool), CudaMemcpyDeviceToHost);
    cudaFree(ch); cudaFree(seq); cudaFree(pres);
    return(presente);
}
```

Version 1 du kernel : pas de mémoire partagée

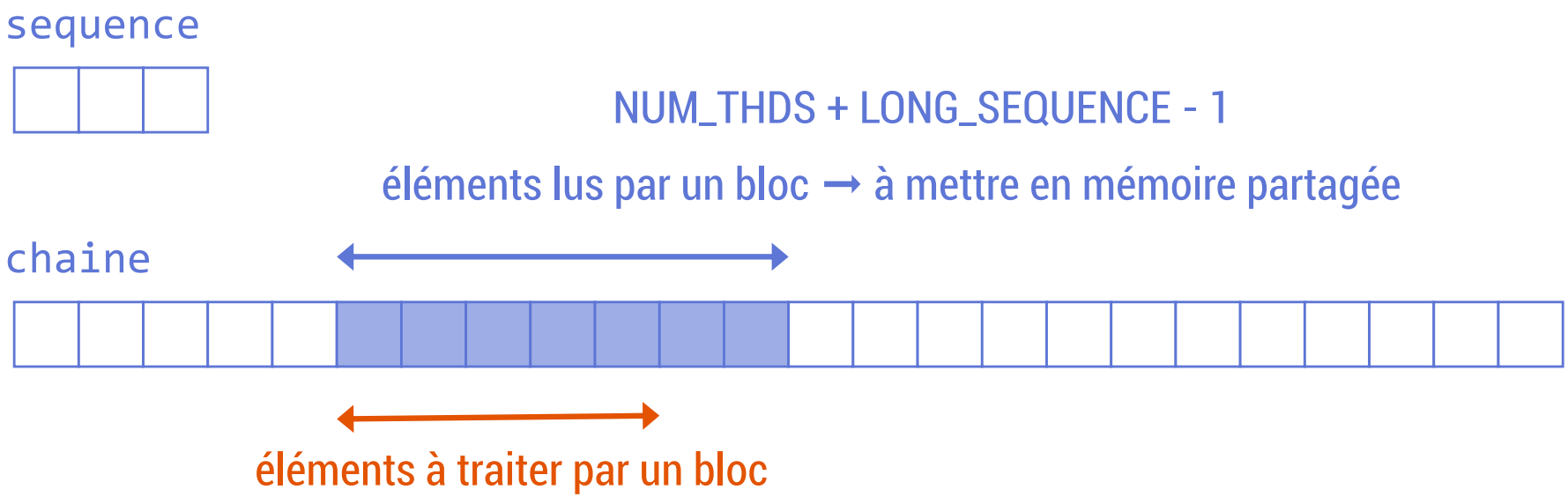
EXERCICE

Version 2 du kernel : en utilisant la mémoire partagée pour ranger la chaine



EXERCICE

Version 2 du kernel : en utilisant la mémoire partagée pour ranger la chaine



sh_chaine

Modifications du kernel :

- ajouter le code qui copie une partie de chaine dans sh_chaine
- modifier le calcul pour utiliser sh_chaine plutôt que chaine

Version 2 du kernel : en utilisant la mémoire partagée pour ranger la chaine

Utilisation de la mémoire constante

Capacité de mémoire constante ?

```
dev_prop.totalConstMem
```

Utilisation de la mémoire constante

- constante déclarée par le programme hôte

```
#define MAX_MASK_WIDTH 10
__constant__ float M[MAX_MASK_WIDTH];
```

- initialisation de constantes

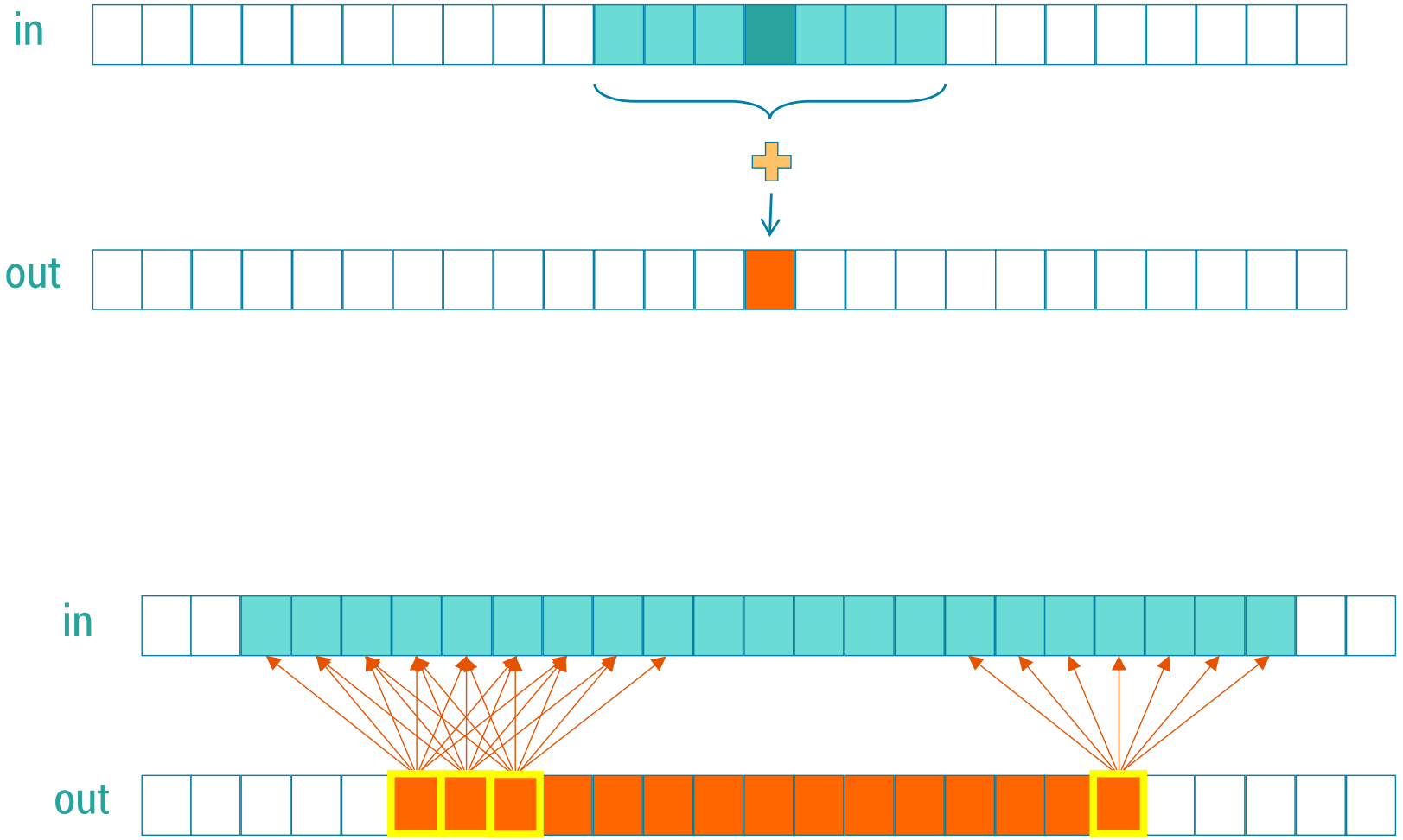
```
cudaMemcpyToSymbol(M,h_M,mask_width*sizeof(f
loat));
```

Version 3 du kernel : utilisation de la mémoire constante pour la séquence

```
bool Contient(unsigned char *sequence,
              unsigned char *chaine){
    unsigned int num_blocks =
       ceil((LONG_CHAINE-LONG_SEQUENCE+1)/NUM_THDS) ;
    bool presente = FALSE;
    bool *pres;
    __constant__ unsigned char seq[LONG_SEQ];
    unsigned char *ch;
    cudaMalloc((void **)&ch, LONG_CHAINE*sizeof(char));
    cudaMemcpy(ch, chaine, LONG_CHAINE, CudaMemcpyHostToDevice);
    cudaMemcpyToSymbol(seq, sequence, LONG_SEQUENCE);
    cudaMalloc((void **)&pres, sizeof(bool));
    cudaMemcpy(pres, &presente, sizeof(bool), CudaMemcpyHostToDevice);
    cherche<<<num_blocks, NUM_THDS>>>(ch, pres);
    cudaMemcpy(presente, pres, sizeof(bool), CudaMemcpyDeviceToHost);
    cudaFree(ch); cudaFree(pres);
    return(presente);
}
```

Version 3 du kernel : utilisation de la mémoire constante pour la séquence

Opération de type *stencil* 1D



Opération de type *stencil* 1D

pour simplifier, on ne s'occupe pas des bordures

```
#define BLOCK_SIZE 512
#define RADIUS 3

__global__ void stencil_1d(int *in, int *out) {
        __shared__ int tmp[BLOCK_SIZE + 2 * RADIUS];
        unsigned int gidx = blockIdx.x * blockDim.x + threadIdx.x;
        unsigned int lidx = threadIdx.x + RADIUS;

        // copier les données dans la mémoire partagée




        ...
```

Opération de type *stencil* 1D

```
__global__ void stencil_1d(int *in, int *out) {
        __shared__ int tmp[BLOCK_SIZE + 2 * RADIUS];
        int gidx = blockIdx.x * blockDim.x + threadIdx.x;
        int lidx = threadIdx.x + RADIUS;

        // copier les données dans la mémoire partagée
        …


        // calculer
        int res = 0;
        for (                                           )
                res += tmp[                ];

        // enregistrer le résultat
        out[gidx] = res / (2*RADIUS+1);
}
```

EXERCICE

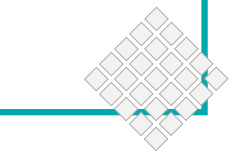
On considère un kernel qui commence comme suit :

```
#define SIZE 256
__global__ compute(char *out, char *in1, char *in2){
            extern __shared__ char *in;
            int tid= blockIdx.x * blockDim.x + threadIdx.x;
            …
```

Ce kernel doit faire un calcul qui nécessite des accès fréquents à in1 et in2 (vecteurs de longueur size) et on souhaite les copier en mémoire partagée. Cette dernière est allouée dynamiquement par le programme principal pour une capacité de 2*SIZE :

```
compute<<<gridSize, blockSize, 2*SIZE>>>(in1,in2);
```

Ecrire les instructions qui permettent le chargement des vecteurs in1 et in2 l'un à la suite de l'autre dans la mémoire partagée (vecteur in). Chaque thread copiera un élément de chacun des deux vecteurs.

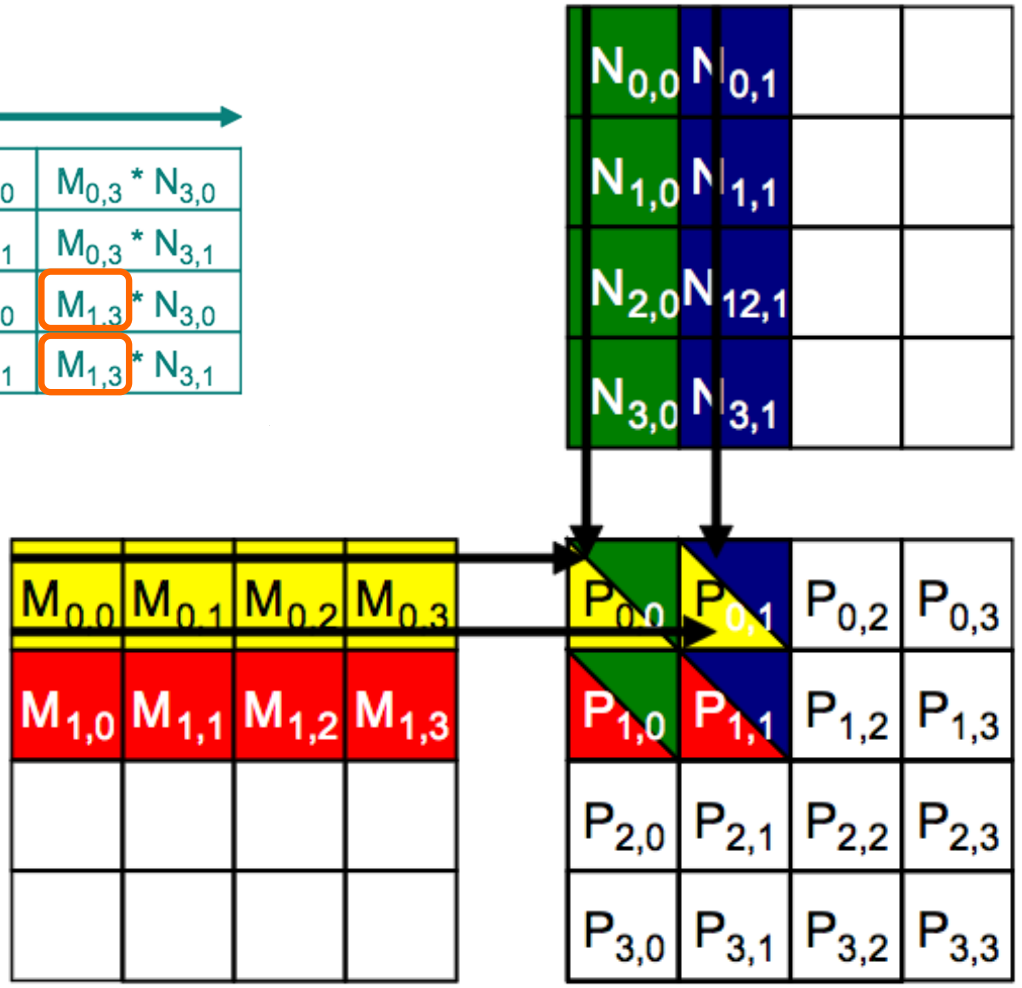


Une stratégie pour réduire le trafic vers la mémoire globale : le pavage

Exemple : produit de matrices

Access order

$thread_{0,0}$	$M_{0,0} * N_{0,0}$	$M_{0,1} * N_{1,0}$	$M_{0,2} * N_{2,0}$	$M_{0,3} * N_{3,0}$
$thread_{0,1}$	$M_{0,0} * N_{0,1}$	$M_{0,1} * N_{1,1}$	$M_{0,2} * N_{2,1}$	$M_{0,3} * N_{3,1}$
$thread_{1,0}$	$M_{1,0} * N_{0,0}$	$M_{1,1} * N_{1,0}$	$M_{1,2} * N_{2,0}$	$M_{1,3} * N_{3,0}$
$thread_{1,1}$	$M_{1,0} * N_{0,1}$	$M_{1,1} * N_{1,1}$	$M_{1,2} * N_{2,1}$	$M_{1,3} * N_{3,1}$

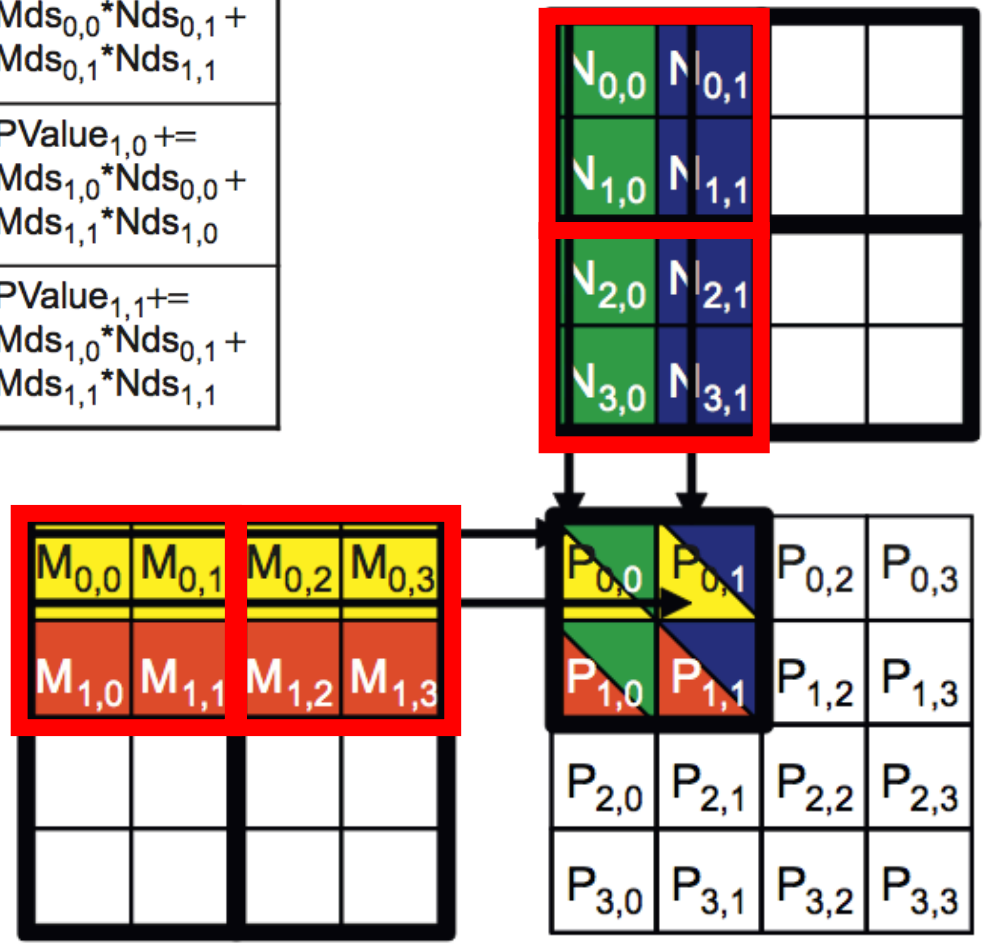


hypothèse : les deux matrices ne contiennent pas dans la mémoire partagée

Produit de matrices pavé (*tiled*)

	Phase 1			Phase 2		
$thread_{0,0}$	**$M_{0,0}$** ↓ $Mds_{0,0}$	**$N_{0,0}$** ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$ + $Mds_{0,1}$*$Nds_{1,0}$	**$M_{0,2}$** ↓ $Mds_{0,0}$	**$N_{2,0}$** ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$ + $Mds_{0,1}$*$Nds_{1,0}$
$thread_{0,1}$	**$M_{0,1}$** ↓ $Mds_{0,1}$	**$N_{0,1}$** ↓ $Nds_{1,0}$	$PValue_{0,1}$ += $Mds_{0,0}$*$Nds_{0,1}$ + $Mds_{0,1}$*$Nds_{1,1}$	**$M_{0,3}$** ↓ $Mds_{0,1}$	**$N_{2,1}$** ↓ $Nds_{0,1}$	$PValue_{0,1}$ += $Mds_{0,0}$*$Nds_{0,1}$ + $Mds_{0,1}$*$Nds_{1,1}$
$thread_{1,0}$	**$M_{1,0}$** ↓ $Mds_{1,0}$	**$N_{1,0}$** ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{1,0}$*$Nds_{0,0}$ + $Mds_{1,1}$*$Nds_{1,0}$	**$M_{1,2}$** ↓ $Mds_{1,0}$	**$N_{3,0}$** ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{1,0}$*$Nds_{0,0}$ + $Mds_{1,1}$*$Nds_{1,0}$
$thread_{1,1}$	**$M_{1,1}$** ↓ $Mds_{1,1}$	**$N_{1,1}$** ↓ $Nds_{1,1}$	$PValue_{1,1}$ += $Mds_{1,0}$*$Nds_{0,1}$ + $Mds_{1,1}$*$Nds_{1,1}$	**$M_{1,3}$** ↓ $Mds_{1,1}$	**$N_{3,1}$** ↓ $Nds_{1,1}$	$PValue_{1,1}$+= $Mds_{1,0}$*$Nds_{0,1}$ + $Mds_{1,1}$*$Nds_{1,1}$

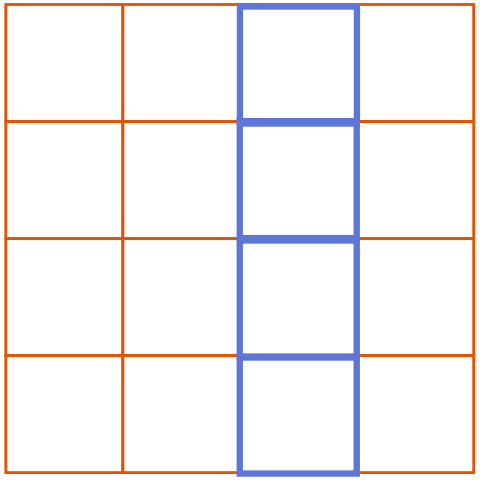
time →



Produit de matrices pavé (*tiled*)

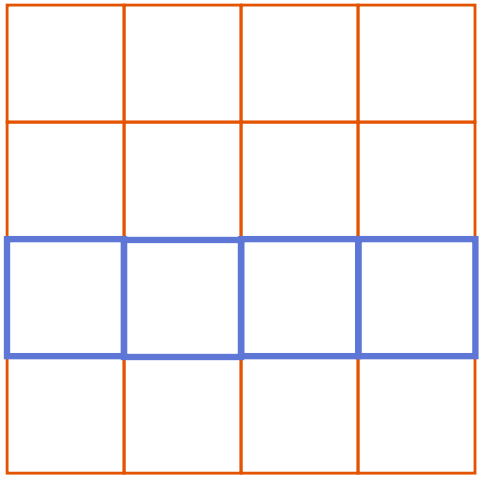
hypothèse :
taille d'un bloc = taille d'une tuile

B

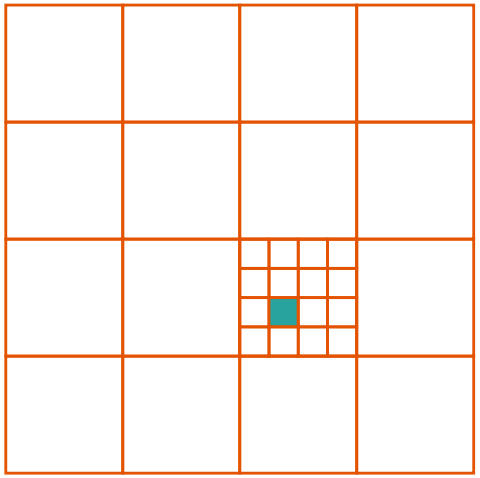


```
t = 0
t = 1
t = 2
t = 3
```

A



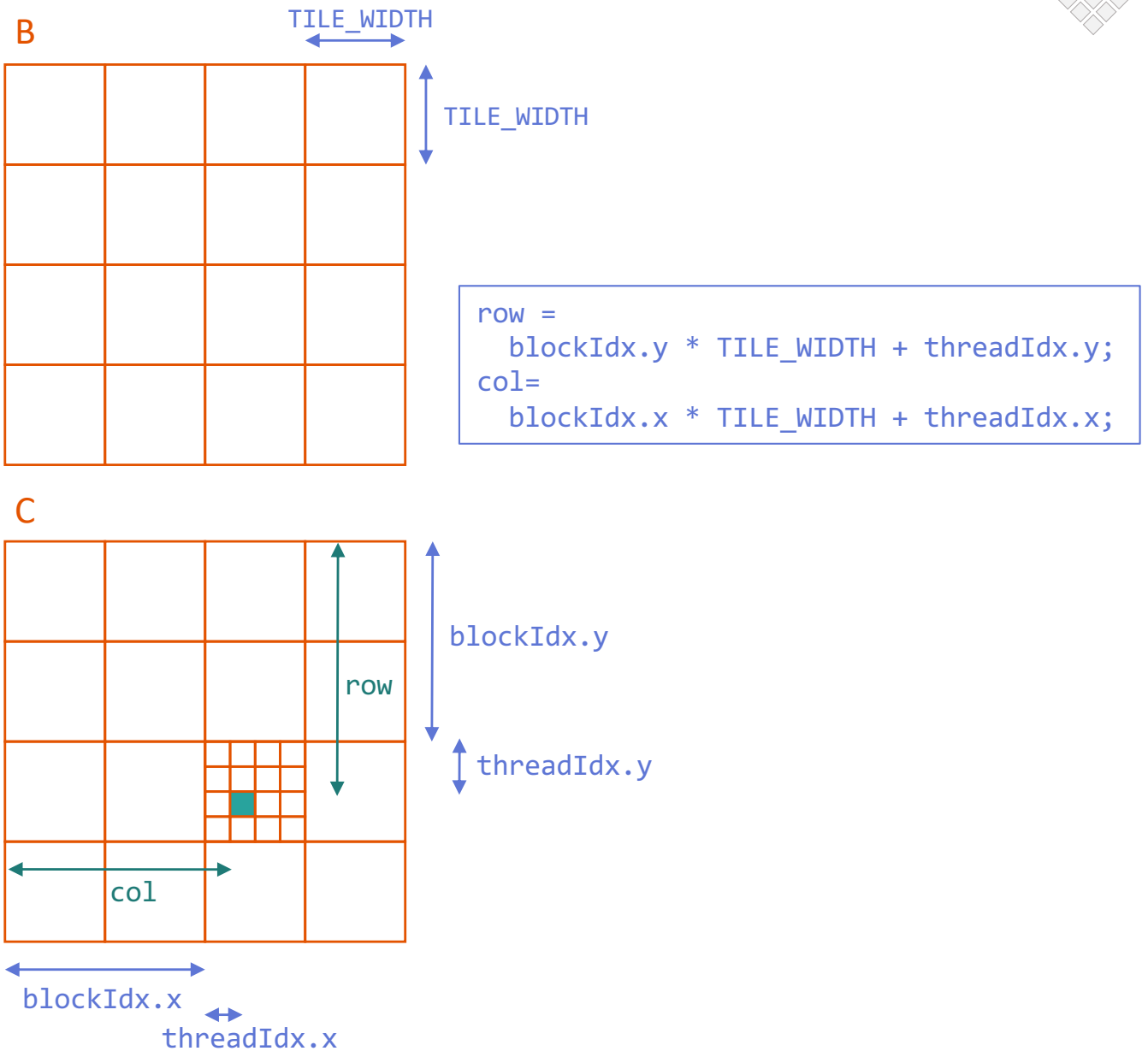
C



Produit de matrices pavé (*tiled*)

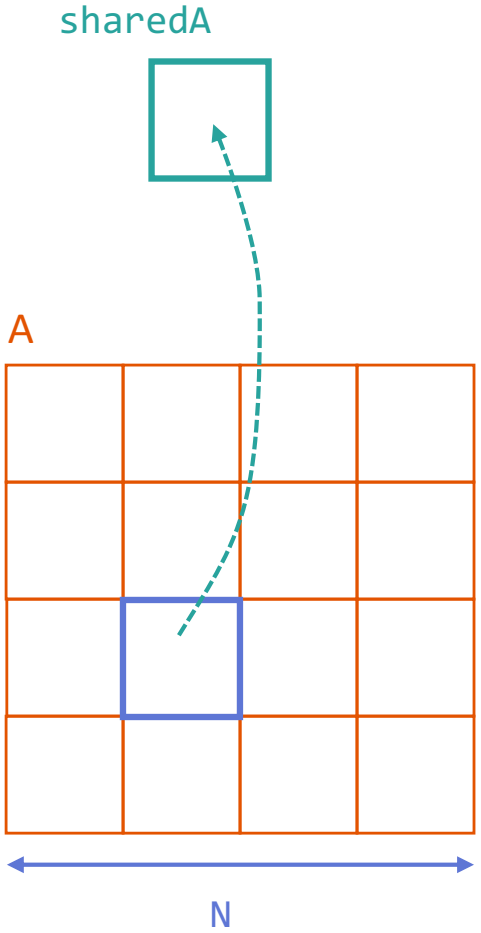
hypothèse :
taille d'un bloc = taille d'une tuile

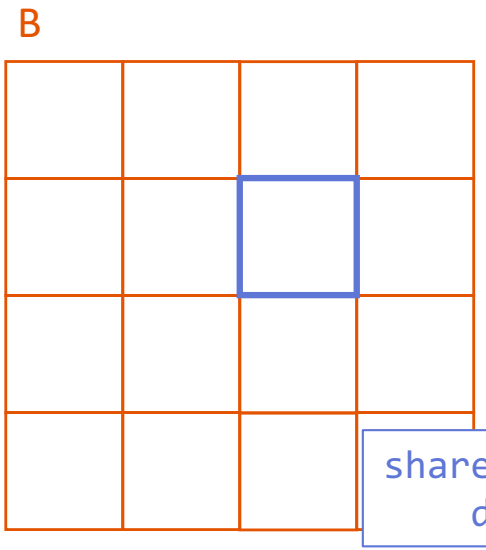
```
blockDim.x == TILE_WIDTH
blockDim.y == TILE_WIDTH
```



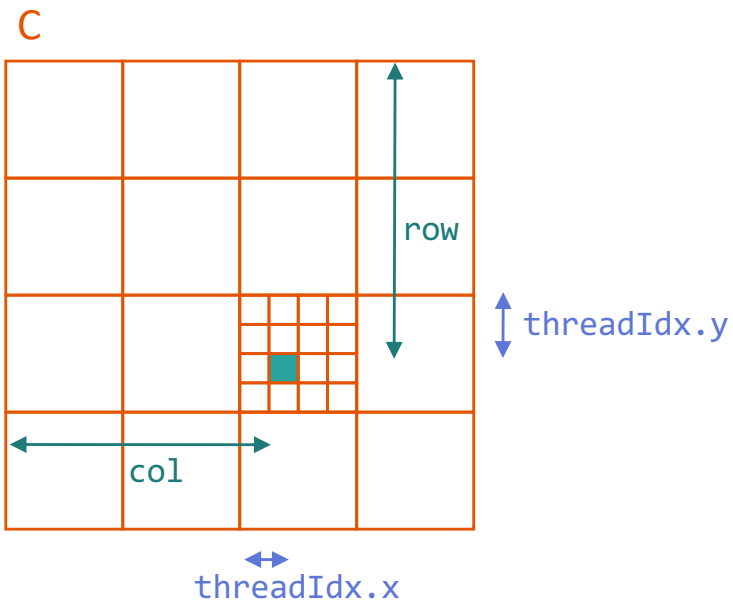
Produit de matrices pavé (*tiled*)

hypothèse :
taille d'un bloc = taille d'une tuile






```
sharedA[threadIdx.y][threadIdx.x] =
    d_A[                                ];
```



Produit de matrices pavé (*tiled*)

```
__global__ void MatMulKernel(float *d_A, float *d_B, float *d_C, int N){

        __shared__ float sharedA[TILE_WIDTH][TILE_WIDTH];
        __shared__ float sharedB[TILE_WIDTH][TILE_WIDTH];
        int row = blockIdx.y * TILE_WIDTH + threadIdx.y;
        int col= blockIdx.x * TILE_WIDTH + threadIdx.x;
        int t,k;
        float res = 0;


        for (t=0 ; t<N/TILE_WIDTH ; t++){
                sharedA[threadIdx.y][threadIdx.x] =
                        d_A[                                    ];
                sharedB[threadIdx.y][threadIdx.x] =
                        d_B[                                    ];
                _syncthreads();
                for (k=0; k<2; k++){
                        res += sharedA[threadIdx.y][k] * sharedB[k][threadIdx.x];
                }
                __syncthreads();
        }
        d_C[                ] = res;
}
```