Introduction à CUDA C

Amina Guermouche

CUDA C

- Basé sur le standard C
- Extension du langage pour la programmation hétérogène
- API pour gérer les GPU, la mémoire, ...
- 3 niveaux d'abstraction : groupe de threads, mémoire partagée et une barrière de synchronisation

Fonctionnement

CPU (**Host**)

```
int main(int
argc, char
**argv)
```

GPU (**Device**)


```
__device__
ma_fonction_gpu
()
```

Fonctionnement

CPU (**Host**)

GPU (**Device**)


```
int main(int
argc, char
**argv)
```

```
__global__
ma_fonction_interface
()
```

```
__device__
ma_fonction_gpu
()
```

Fonctionnement

1. Host : Copier les données d'entrée de la mémoire du CPU à la mémoire du GPU
2. Device : Charger les instructions sur le GPU
3. Device : Copier les données vers la mémoire du CPU

Interrogation du GPU

- Quel est le nombre de GPUs disponibles
- Quelle est la taille de la mémoire disponible ?
- Quelles sont les caractéristiques des GPUs ?

```
cudaDeviceProp prop;
int count;
cudaGetDeviceCount(&count);

for (int i = 0; i < count; i ++)
{
   cudaGetDeviceProperties(&prop, i);
   printf(''Taille total de la memoire globale
        %ld\n'', prop.totalGlobalMem);
}
```

- On peut même choisir le GPU qu'on veut selon des critères ! ! ! !

```
cudaChooseDevice(&dev, &prop)
```

Exercice 1

Hello, World !

```
int main (void) {

   printf (''Hello, World!\n'');
   return 0;
}
```

- Compilation avec nvcc (compilateur NVIDIA)
- nvcc ne se plaint pas s'il n'y a pas de code pour le device

Hello, World !

```
__global__ void kernel (void){}

int main (void) {
  kernel<<<1,1>>>();
   printf (''Hello, World!\n'');
   return 0;
}
```

- Compilation avec nvcc (compilateur NVIDIA)
- nvcc ne se plaint pas s'il n'y a pas de code pour le device

Hello, World !

Code du device

```
__global__ void kernel (void)
```

- `__global__` indique que :

→ Le code s'exécute sur le device

→ Le code est appelé du host

- La partie device et interface est gérée par le compilateur nvidia
- La partie host par le compilateur C
- La syntaxe est obligatoire
- `__global__` ne retourne pas de valeur, JAMAIS

ok on utilise le GPU pour appeler la fonction kernel qui ne fait rien, super ! ! ! !

Et si on faisait faire quelque chose au GPU

```
__global__ void add (int *a, int *b, int *c){
  *c = *a + *b;
}
```

- add(...) sera appelé du host
- add(...) sera exécutée sur le device
- et la mémoire ?

Et si on faisait faire quelque chose au GPU

```
__global__ void add (int *a, int *b, int *c){
  *c = *a + *b;
}
```

- add(...) sera appelé du host
- add(...) sera exécutée sur le device
- et la mémoire ?
  - a, b, et c pointent sur la mémoire du device
  - Comment allouer de la mémoire sur le GPU ?
    `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`

Et si on faisait faire quelque chose au GPU (1/2)

```
__global__ void add (int *a, int *b, int *c){
  *c = *a + *b;
}
int main ( void ){
  int a, b, c;
  int *gpu_a, *gpu_b, *gpu_c;
  int size = sizeof(int);
```

Et si on faisait faire quelque chose au GPU (1/2)

```
__global__ void add (int *a, int *b, int *c){
  *c = *a + *b;
}
int main ( void ){
  int a, b, c;
  int *gpu_a, *gpu_b, *gpu_c;
  int size = sizeof(int);
  // allocation de l'espace pour le device
  cudaMalloc( (void **)&gpu_a, size);
  cudaMalloc( (void **)&gpu_b, size);
  cudaMalloc( (void **)&gpu_c, size);
  a=2;
  b=7;
```

Et si on faisait faire quelque chose au GPU (2/2)

```
    // Copie des donnees vers le Device
    cudaMemcpy (gpu_a, &a, size,
        cudaMemcpyHostToDevice);
    cudaMemcpy (gpu_b, &b, size,
        cudaMemcpyHostToDevice);

    add <<< 1, 1 >>> (gpu_a, gpu_b, gpu_c);









    return 0
}
```

Et si on faisait faire quelque chose au GPU (2/2)

```
// Copie des donnees vers le Device
   cudaMemcpy (gpu_a, &a, size,
       cudaMemcpyHostToDevice);
   cudaMemcpy (gpu_b, &b, size,
       cudaMemcpyHostToDevice);

   add <<< 1, 1 >>> (gpu_a, gpu_b, gpu_c);

   //Copie du resultat vers Host
   cudaMemcpy (&c, gpu_c, size, cudaMemcpyDeviceToHost);









   return 0
}
```

Et si on faisait faire quelque chose au GPU (2/2)

```
// Copie des donnees vers le Device
   cudaMemcpy (gpu_a, &a, size,
       cudaMemcpyHostToDevice);
   cudaMemcpy (gpu_b, &b, size,
       cudaMemcpyHostToDevice);

   add <<< 1, 1 >>> (gpu_a, gpu_b, gpu_c);

   //Copie du resultat vers Host
   cudaMemcpy (&c, gpu_c, size, cudaMemcpyDeviceToHost);

   //Liberation de l'espace alloue
   cudaFree(gpu_a);
   cudaFree( gpu_b);
   cudaFree ( gpu_c);

   return 0
}
```

Et si on faisait faire quelque chose au GPU (2/2)

```
// Copie des donnees vers le Device
    checkCudaErrors(cudaMemcpy (gpu_a, &a, size,
        cudaMemcpyHostToDevice));
    cudaMemcpy (gpu_b, &b, size,
        cudaMemcpyHostToDevice);

    add <<< 1, 1 >>> (gpu_a, gpu_b, gpu_c);

    //Copie du resultat vers Host
    cudaMemcpy (&c, gpu_c, size, cudaMemcpyDeviceToHost);

    //Liberation de l'espace alloue
    cudaFree(gpu_a);
    cudaFree( gpu_b);
    cudaFree ( gpu_c);

    return 0
}
```

addition de 2 entiers : super utilisation du parallélisme

- Comment exécuter le code en parallèle ?

  On veut faire N fois add en parallèle

  ```
  add<<<1, 1>>> (gpu_a, gpu_b, gpu_c)
  ```

addition de 2 entiers : super utilisation du parallélisme

- Comment exécuter le code en parallèle ?

  On veut faire N fois add en parallèle

```
add<<<1, 1>>> (gpu_a, gpu_b, gpu_c)

add<<<N, 1>>> (gpu_a, gpu_b, gpu_c)
```

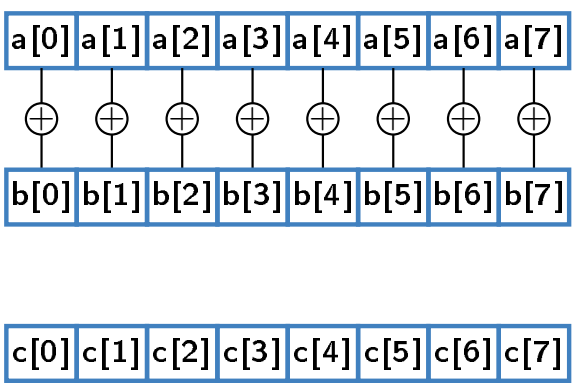
addition de 2 entiers : super utilisation du parallélisme

- Comment exécuter le code en parallèle ?

  On veut faire N fois add en parallèle

```
add<<<1, 1>>> (gpu_a, gpu_b, gpu_c)

add<<<N, 1>>> (gpu_a, gpu_b, gpu_c)
```

- Dans ce cas, autant faire un add sur un vecteur



- Comment sont exprimés les indices sur le GPU ?

Programmation parallèle en CUDA

- Chaque appel parallèle à `add(...)` est appelé **block**
- L'accès à un block donné se fait via `blockIdx.x`
- Chaque `blockIdx.x` référence un élément du tableau

```
__global__ void add (int *a, int *b, int *c){
  c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

Programmation parallèle en CUDA

- Chaque appel parallèle à `add(...)` est appelé **block**
- L'accès à un block donné se fait via `blockIdx.x`
- Chaque `blockIdx.x` référence un élément du tableau

```
__global__ void add (int *a, int *b, int *c){
  c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

$\texttt{Block}_0$

```
c[0] = a[0] + b[0]
```

$\texttt{Block}_1$

```
c[1] = a[1] + b[1]
```

$\texttt{Block}_2$

```
c[2] = a[2] + b[2]
```

$\texttt{Block}_3$

```
c[3] = a[3] + b[3]
```

Programmation parallèle en CUDA : le main

```
#define N 512 //nombre d'elements du tableau
int main (void){
  int *a, *b, *c;
  int *gpu_a, *gpu_b, *gpu_c;
  int size = N * sizeof(int);
  // allocation de l'espace pour le device}
  cudaMalloc((void **)&gpu_a, size);
  cudaMalloc((void **)&gpu_b, size);
  cudaMalloc((void **)&gpu_c, size);

  a=(int*) malloc (size);
  b=(int*) malloc (size);
  random_ints(a, N);
  random_ints(b, N);
```

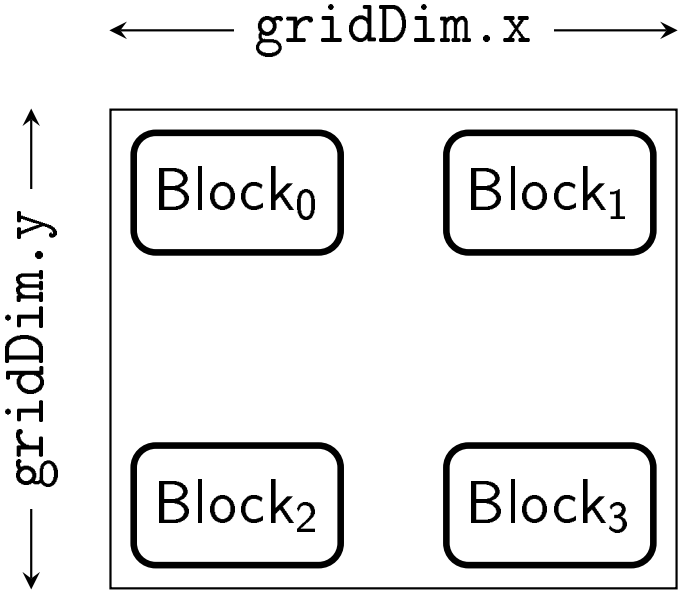
Programmation parallèle en CUDA : le main

```
// Copie des donnees vers le Device
cudaMemcpy (gpu_a,a,size,cudaMemcpyHostToDevice);
cudaMemcpy (gpu_b,b,size,cudaMemcpyHostToDevice);

add <<< N, 1 >>> (gpu_a, gpu_b, gpu_c);

//Copie du resultat
cudaMemcpy(c, gpu_c, size, cudaMemcpyDeviceToHost);

free(a); free(b); free(c);
cudaFree(gpu_a);
cudaFree(gpu_b);
cudaFree(gpu_c);
return 0;
}
```

Et si on avait un vecteur à 2 dimensions (une matrice donc) ?

- Le nombre de blocks lancés représentent une grille (*grid*)
- Le nombre de blocks par dimension est limité (`maxGridSize[3]`)
- `blockIdx.x`, `blockIdx.y`, `blockIdx.z`
- `dim3 grid(DIM, DIM)` initialise la variable `grid` de type `dim3` qui indique la dimension de la grille (2D)
- `gridDim.x`, `gridDim.y` donnent la dimension de la grille

Grille et blocks



Addition de deux matrices

```
#define N 512 //taille d'une dimension de la
    matrice

__global__ void add (int *a, int *b, int *c){
  int x = blockIdx.x;
  int y = blockIdx.y;
  int indice = x + y * gridDim.x;
  c[indice] = a[indice] + b[indice];
}

int main (void){
  int *a, *b, *c;
  int *gpu_a, *gpu_b, *gpu_c;
  int size = N * sizeof(int);
  ...
  dim3 grid(N, N);
  add <<<grid, 1 >>> (dev_a, dev_b, dev_c);
  ...
}
```

Exercice 2

Threads

- Un block peut être divisé en plusieurs threads parallèles

- CUDA définit un unique id par thread `threadIdx.x`
- On utilise `threadIdx.x` au lieu de `blockIdx.x`

```
__global__ void add (int *a, int *b, int *c){
  c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];
}
```

Programmation parallèle en CUDA : le main

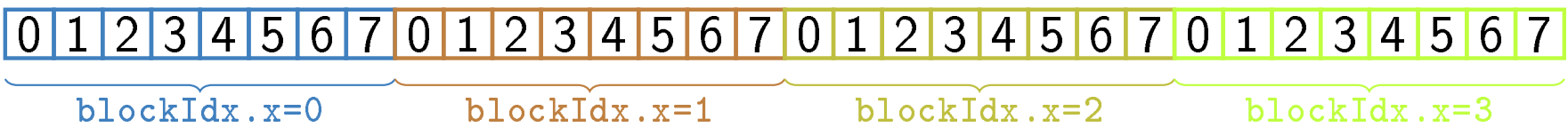
```
#define N 512 //nombre d'elements du tableau
int main (void){
  int *a, *b, *c;
  int *gpu_a, *gpu_b, *gpu_c;
  int size = N * sizeof(int);
  // allocation de l'espace pour le device}
  cudaMalloc((void **)&gpu_a, size);
  cudaMalloc((void **)&gpu_b, size);
  cudaMalloc((void **)&gpu_c, size);

  a=(int*) malloc (size);
  b=(int*) malloc (size);
  random_ints(a, N);
  random_ints(b, N);
```

Programmation parallèle en CUDA : le main

```
// Copie des donnees vers le Device
cudaMemcpy (gpu_a,a,size,cudaMemcpyHostToDevice);
cudaMemcpy (gpu_b,b,size,cudaMemcpyHostToDevice);

//Lancement de l'operation avec N threads
add <<< 1, N >>> (gpu_a, gpu_b, gpu_c);

//Copie du resultat
cudaMemcpy(c, gpu_c, size, cudaMemcpyDeviceToHost);

free(a); free(b); free(c);
cudaFree(gpu_a);
cudaFree(gpu_b);
cudaFree(gpu_c);
return 0;
}
```

Des blocks et des threads

- Les threads sont numérotés de 0 à `nb_thread` par block
- `threadIdx.x` est par block



- Si on a `M` threads/block, l'indice dans un vecteur est calculé par :
  `indice = threadIdx.x + blockIdx.x * M`
- Le nombre de threads par block est donné par la variable `blockDim.x`
  `indice = threadIdx.x + blockIdx.x * blockDim.x`

add avec blocks et threads

```
#define N (2048 * 2048) //taille du tableau
#define THREAD_PER_BLOCK 512 //nombre de threads

__global__ void add (int *a, int *b, int *c){
  int indice = threadIdx.x + blockIdx.x * blockDim.
      x;
  c[indice] = a[indice] + b[indice];
}

int main (void){
  int *a, *b, *c;
  int *gpu_a, *gpu_b, *gpu_c;
  int size = N * sizeof(int);
  // allocation de l'espace pour le device}
  cudaMalloc((void **)&gpu_a, size);
  cudaMalloc((void **)&gpu_b, size);
  cudaMalloc((void **)&gpu_c, size);
```

Programmation parallèle en CUDA : le main

```
a=(int*) malloc (size);
b=(int*) malloc (size);
random_ints(a, N);
random_ints(b, N);

// Copie des donnees vers le Device
cudaMemcpy (gpu_a,a,size,cudaMemcpyHostToDevice);
cudaMemcpy (gpu_b,b,size,cudaMemcpyHostToDevice);

//Lancement de l'operation avec THREAD_PER_BLOCK
    par block
add <<< N/THREAD_PER_BLOCK,THREAD_PER_BLOCK >>>
     (gpu_a, gpu_b, gpu_c);

//Copie du resultat
cudaMemcpy(c, gpu_c, size, cudaMemcpyDeviceToHost);
```

Programmation parallèle en CUDA : le main

```
free(a); free(b); free(c);
cudaFree(gpu_a);
cudaFree(gpu_b);
cudaFree(gpu_c);
return 0;
}
```

Exercice 3

Vecteurs de taille quelconque

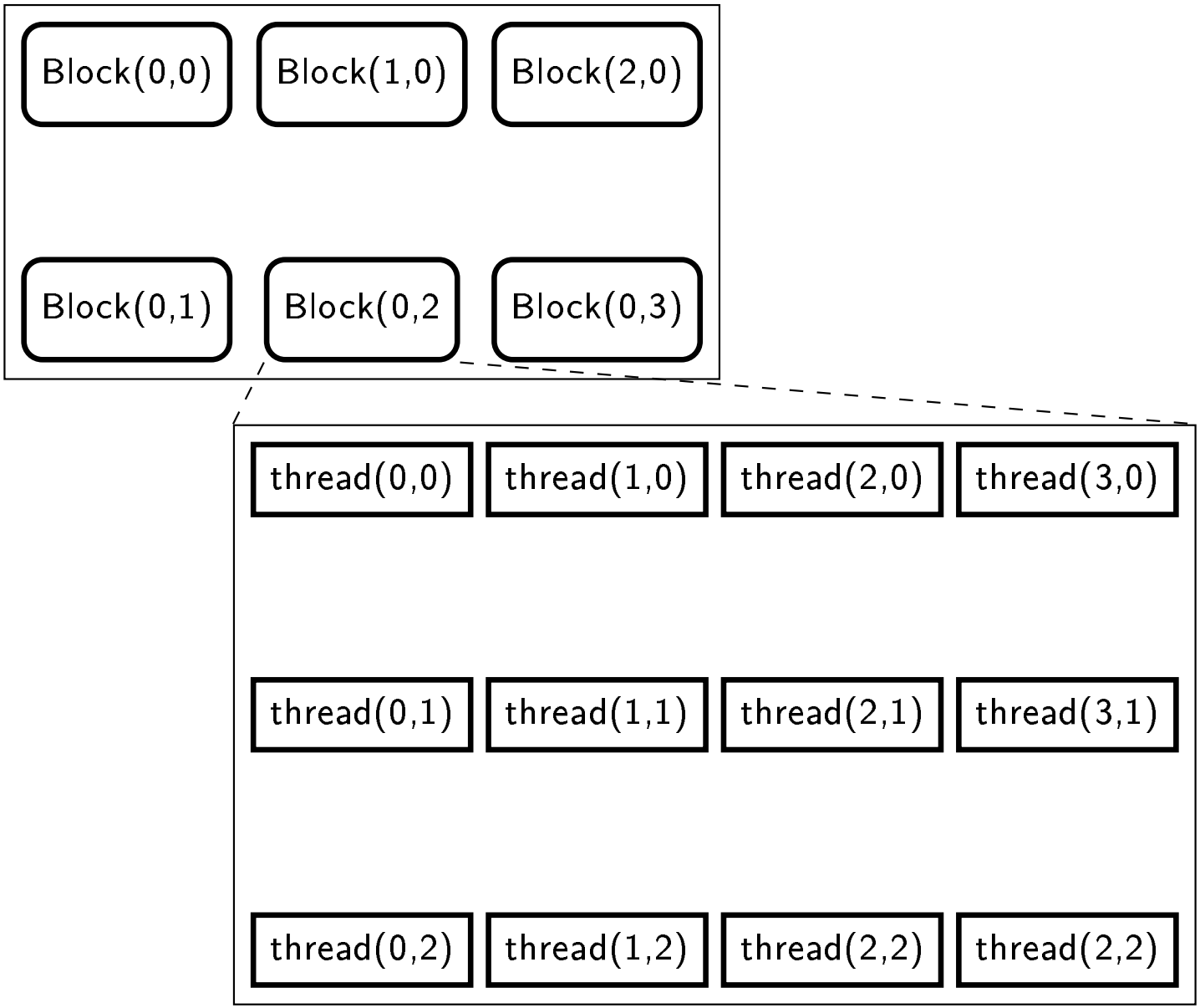
- Vecteur de taille non multiple de `blockDim.x`

```
__global__ void add (int *a, int *b, int *c,
    int n){
  int indice = threadIdx.x + blockIdx.x *
      blockDim.x;
  if (indice < n)
      c[indice] = a[indice] + b[indice];
}
```

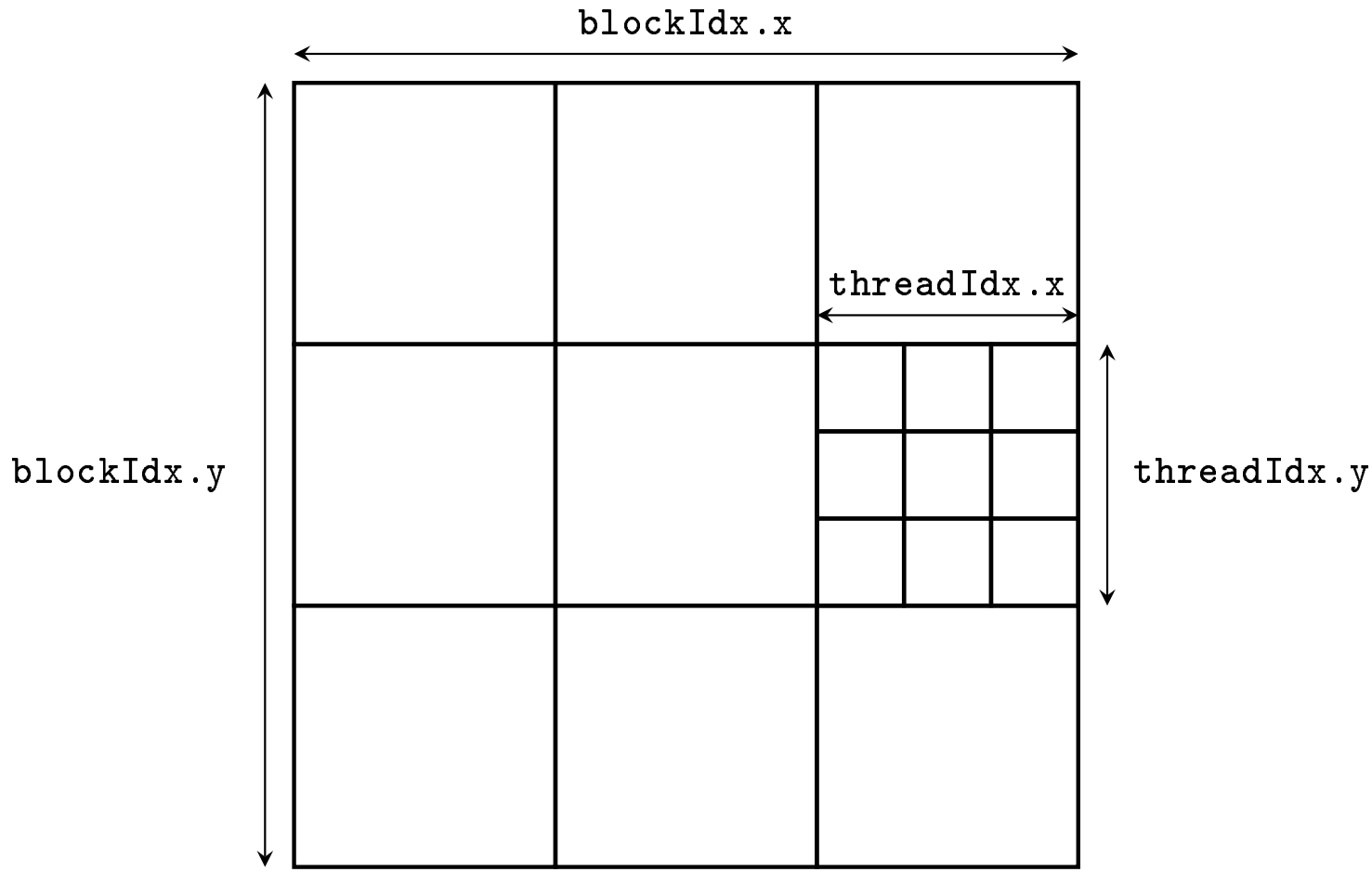
- Au niveau du main

```
add <<< (N+THREAD_PER_BLOCK-1)/
    THREAD_PER_BLOCK, THREAD_PER_BLOCK >>> (
    gpu_a, gpu_b, gpu_c, N);
```

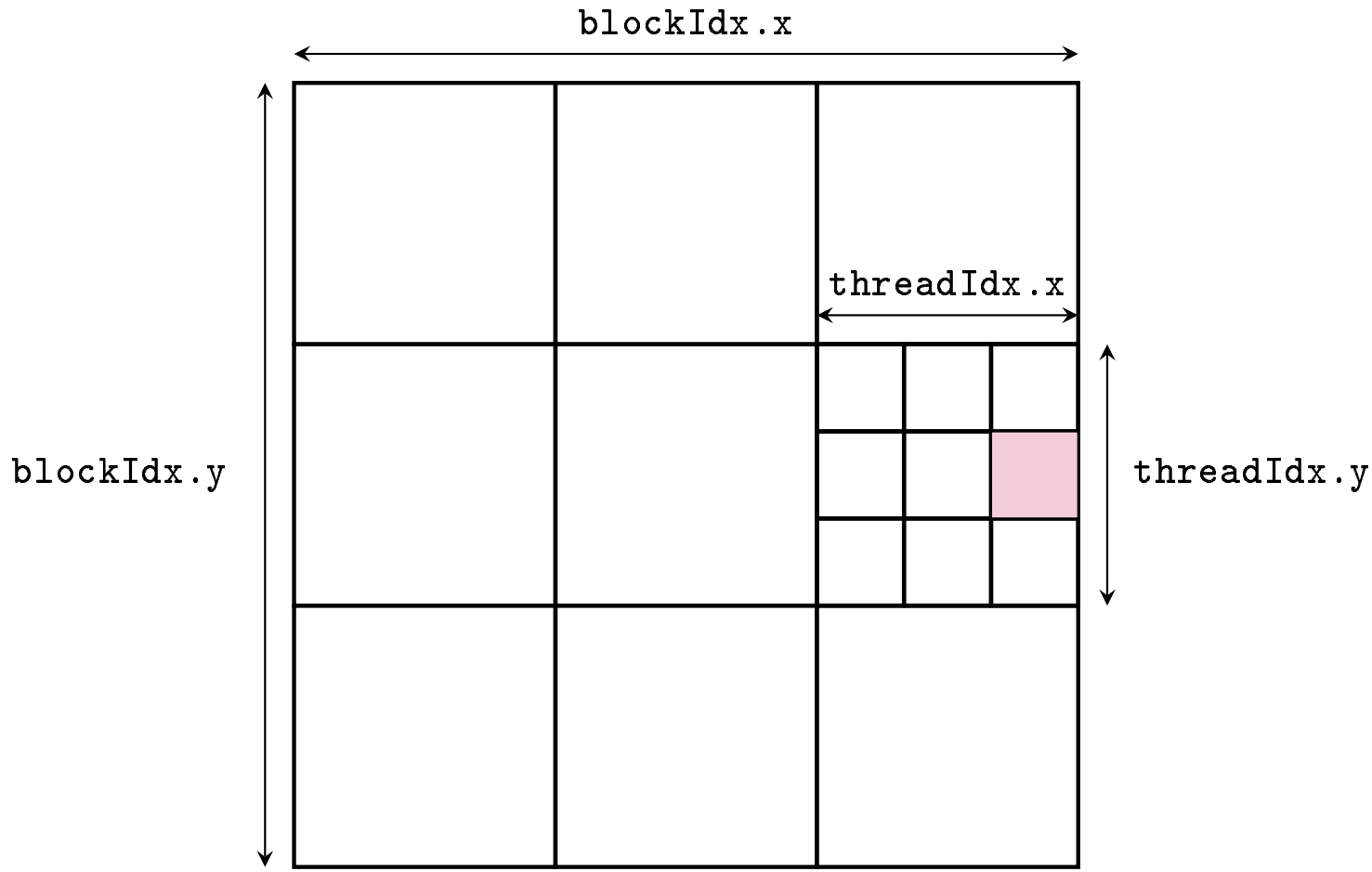
Des blocks et des threads



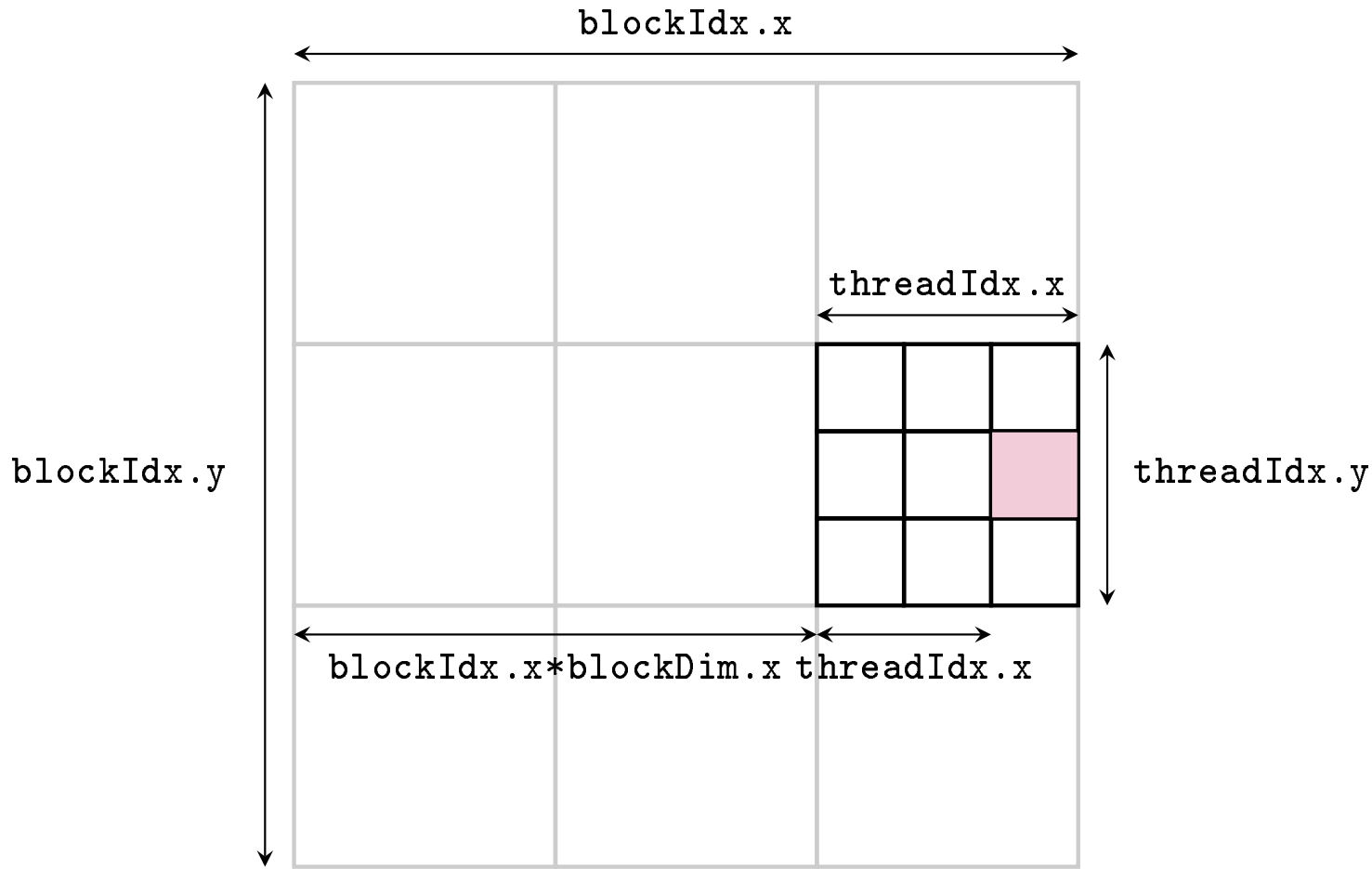
Des blocks et des threads



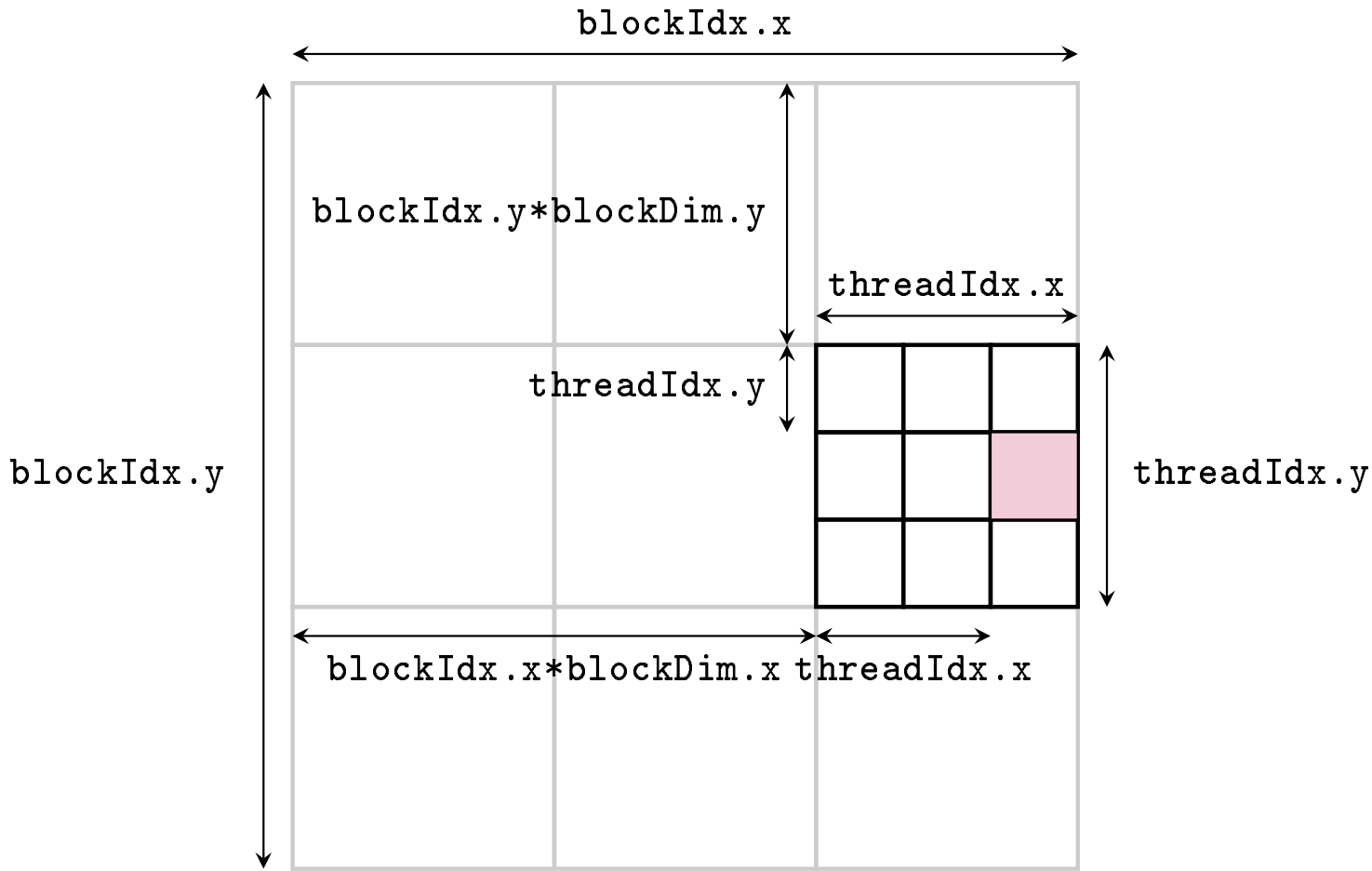
Des blocks et des threads



Des blocks et des threads

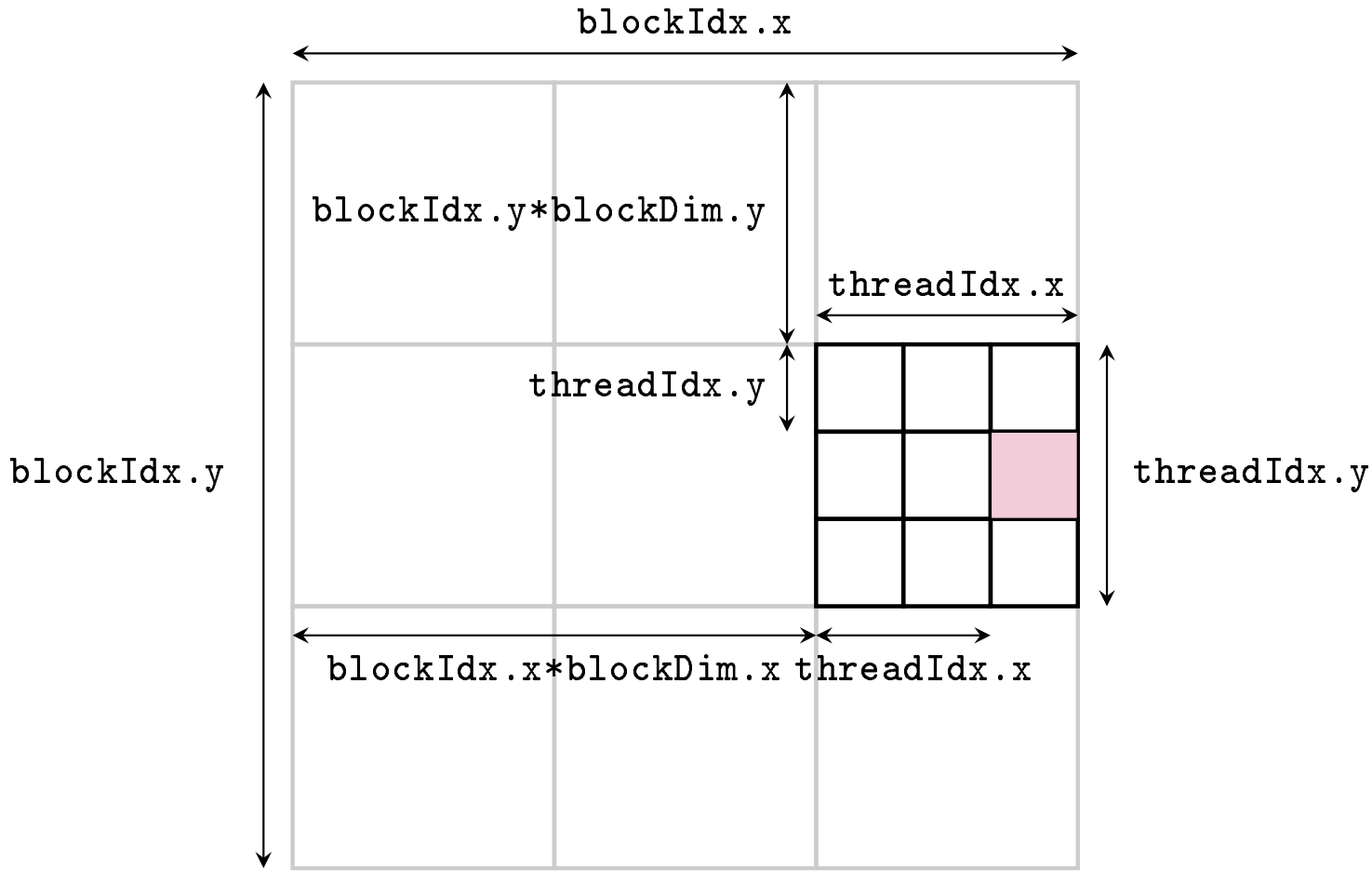


Des blocks et des threads



Addition de 2 matrices

- `colonne = blockIdx.x*blockDim.x+theadIdx.x`
- `ligne = blockIdx.y*blockDim.y+theadIdx.y`

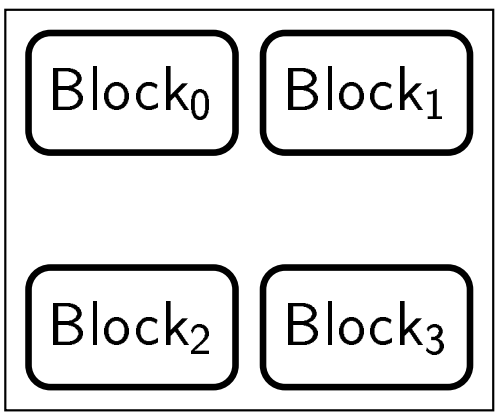


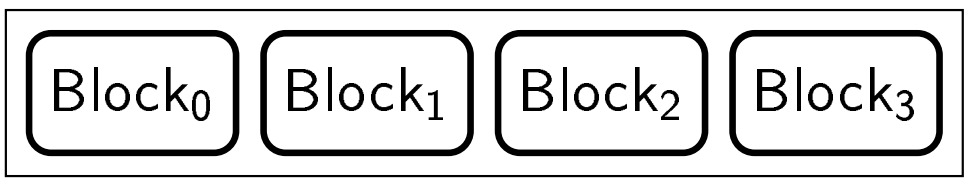
Addition de deux matrices

```
#define N 2048 //taille de la matrice
#define THREAD_PER_BLOCK 512 //nombre de threads

__global__ void add (int *a, int *b, int *c){
  int colonne = blockIdx.x*blockDim.x+theadIdx.x;
  int ligne = blockIdx.y*blockDim.y+theadIdx.y;
  int indice = ligne * N + colonne;
  // N = gridDim.x * blockDim.x;
  c[indice] = a[indice] + b[indice];
}

int main (int argc, char** argv)
{
  ...
  dim3 block(THREAD_PER_BLOCK, THREAD_PER_BLOCK);
  dim3 grid(N/blockDim.x, N/blockDim.y);
  add <<<grid, block >>> (dev_a, dev_b, dev_c);
  ...
}
```

Pourquoi s'embêter avec des threads et des blocks ?

- 1 thread = 1 coeur
- 1 block = 1SM
- 1 block est exécuté sur 1SM
- Blocks :
  - Les blocks sont exécutés dans n'importe quel ordre, séquentiellement ou en parallèle
  - L'avantage est que ça scale automatiquement avec le nombre de SM





Pourquoi s'embêter avec des threads et des blocks ?

- 1 thread = 1 coeur
- 1 block = 1SM
- 1 block est exécuté sur 1SM
- Blocks :
  - Les blocks sont exécutés dans n'importe quel ordre, séquentiellement ou en parallèle
  - L'avantage est que ça scale automatiquement avec le nombre de SM
- Threads
  - Contrairement aux blocks, les threads peuvent
    - Communiquer
    - Se synchroniser
  - Ces opérations sont à l'intérieur d'un block

Les paramètres du kernel

- Les blocks :
  - Le nombre de blocks doit être supérieur au nombre de SM (pour que tous travaillent)
  - Il devrait y avoir plusieurs blocks par SM, afin que d'autres blocks s'exécutent pendant une synchronisation

  → Si une synchronisation est utilisée, il vaut mieux utiliser plusieurs petits blocks qu'un grand
- Les threads :
  - Les SM ordonnancent les threads par groupe SIMD de 32 (warp) sur Quadro 620
    - Les threads d'un warp sont synchronisés

  → Les threads dans un block sont exécutés par groupe de 32
  - Un SM peut exécuter plusieurs blocks de manière concurrente

Les paramètres du kernel

- Les blocks :
  - Le nombre de blocks doit être supérieur au nombre de SM (pour que tous travaillent)
  - Il devrait y avoir plusieurs blocks par SM, afin que d'autres blocks s'exécutent pendant une synchronisation

  → Si une synchronisation est utilisée, il vaut mieux utiliser plusieurs petits blocks qu'un grand
- Les threads :
  - Les SM ordonnancent les threads par groupe SIMD de 32 (warp) sur Quadro 620
    - Les threads d'un warp sont synchronisés

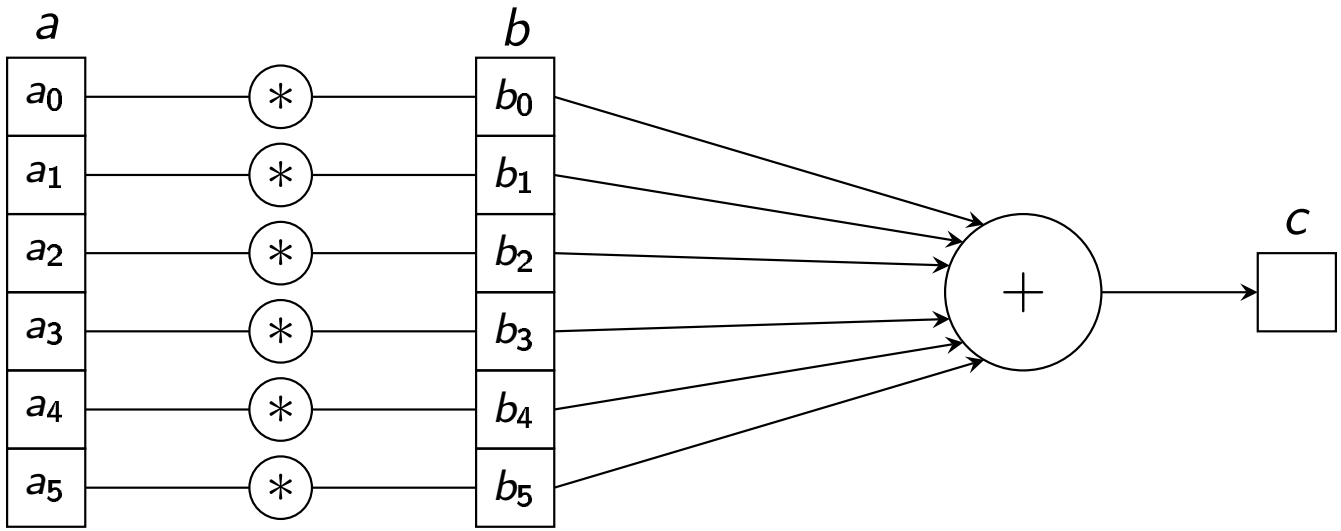
  → Les threads dans un block sont exécutés par groupe de 32
  - Un SM peut exécuter plusieurs blocks de manière concurrente
  - Utiliser des blocks de taille multiple de la taille du warp

Les instructions de contrôles dans les warps

- Les instructions de contrôle (if, while, for, switch, do) affectent les performances, car les thread d'un warp vont diverger
- Les différents chemins sont sérialisés
- Lorsque toutes les exécutions sur les différents chemins sont finies, les threads convergent vers le même chemin
- Les conditions doivent minimiser les divergences
  - Par exemple une condition dépendant de $(threadIdx/warp\_size)$
- Il faut utiliser plus de threads CUDA que de nombre de cœurs CUDA pour augmenter le parallélisme

Exercice 4

Produit scalaire (dot product)



$$c = (a_0, a_1, a_2, a_3, a_4, a_5).(b_0, b_1, b_2, b_3, b_4, b_5)$$
$$= a_0 b_0 + a_1 b_1 + a_2 b_2 + a_3 b_3 + a_4 b_4 + a_5 b_5$$

Produit scalaire (dot product) : 1 seul block

```
__global__ void dot (int *a, int *b, int *c){
  // chaque thread calcule le produit d'une paire
  int tmp = a[threadIdx.x] * b[threadIdx.x];
}
```

Produit scalaire (dot product) : 1 seul block

```
__global__ void dot (int *a, int *b, int *c){
  // chaque thread calcule le produit d'une paire
  int tmp = a[threadIdx.x] * b[threadIdx.x];
}
```

- Le calcul est local au processus

→ Les variables `temp` ne sont pas accessibles aux autres processus

→ Mais il faut partager les données pour faire la somme finale

Partager les données entre les threads (d'un même block)

- Les threads d'un block partagent une zone mémoire appelée shared memory
- Caractéristiques
  - Extrêmement rapide
  - *on-chip*
  - Déclarée avec `__shared__`
- Des blocks sur le même SM partage la même mémoire partagée globale. Donc pour une mémoire de 48KB avec N blocks sur le même SM, chaque block possédera 48/N de mémoire partagée

Produit scalaire (dot product) : 1 seul block

```
#define N 512 //taille du tableau
__global__ void dot (int *a, int *b, int *c){
  __shared__ int tmp[N]
  // chaque thread calcule le produit d'une paire
  tmp[threadIdx.x] = a[threadIdx.x] * b[threadIdx.x
      ];

  //Le thread 0 effectue la somme
  if (0 == threadIdx.x){
    int sum = 0;
    for (int i = 0; i < N; i++)
      sum = sum + temp[i];
    *c = sum;
  }
}
```

Synchronisation des threads d'un même block

- Grâce à la fonction `__syncthreads()`
- Synchronise uniquement les threads d'un même block

Produit scalaire (dot product) : 1 seul block

```
#define N 512 //taille du tableau
__global__ void dot (int *a, int *b, int *c){
   __shared__ int tmp[N]
   // chaque thread calcule le produit d'une paire
   tmp[threadIdx.x] = a[threadIdx.x] * b[threadIdx.x
       ];

   //Synchronisation pour etre sur que tout les
       threads ont fini
   __syncthreads();

   //Le thread 0 effectue la somme
   if (0 == threadIdx.x){
      int sum = 0;
      for (int i = 0; i < N; i++)
         sum = sum + temp[i];
      *c = sum;
   }
}
```

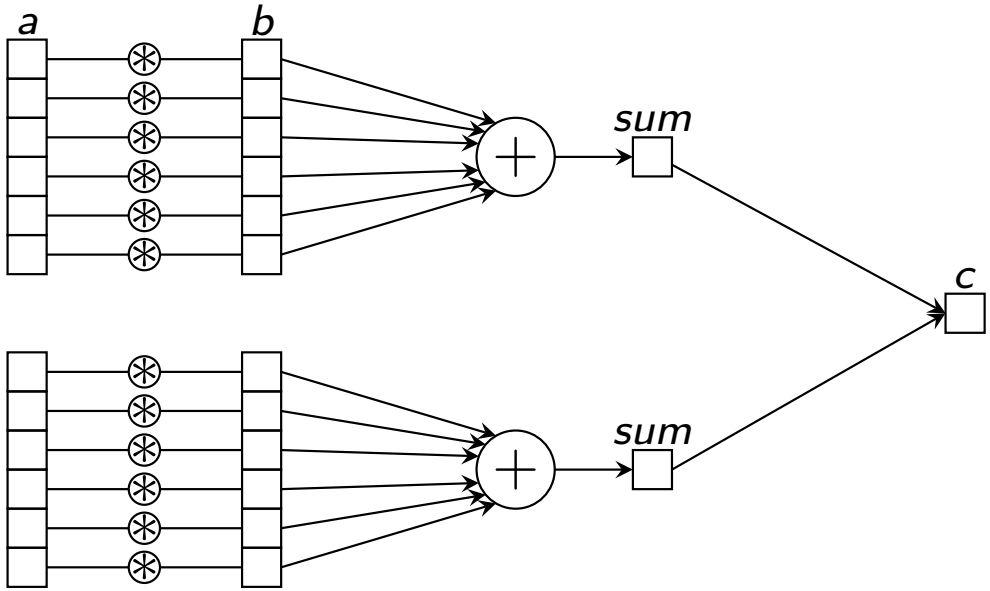
Produit scalaire (dot product) : 1 seul block

```
int main (void){
  int *a, *b, *c;
  int *gpu_a, *gpu_b, *gpu_c;
  int size = N * sizeof(int);
  // allocation de l'espace pour le device}
  cudaMalloc((void **)&gpu_a, size);
  cudaMalloc((void **)&gpu_b, size);
  cudaMalloc((void **)&gpu_c, sizeof(int));

  a=(int*) malloc (size);
  b=(int*) malloc (size);
  random_ints(a, N);
  random_ints(b, N);
```

Programmation parallèle en CUDA : le main

```
// Copie des donnees vers le Device
cudaMemcpy (gpu_a,a,size,cudaMemcpyHostToDevice);
cudaMemcpy (gpu_b,b,size,cudaMemcpyHostToDevice);

//Lancement de l'operation avec N threads et un
    seul block
dot <<< 1, N >>> (gpu_a, gpu_b, gpu_c);

//Copie du resultat
cudaMemcpy(&c, gpu_c, sizeof(int),
    cudaMemcpyDeviceToHost);

free(a); free(b);
cudaFree(gpu_a);
cudaFree(gpu_b);
cudaFree(gpu_c);
return 0;
}
```

Plus de parallélisme : plusieurs blocks



Produit scalaire (dot product)

```
#defin N (2048 * 2048)
#define THREAD_PER_BLOCK 512 //taille du tableau
__global__ void dot (int *a, int *b, int *c){
  __shared__ int tmp[THREADS_PER_BLOCK]
  // chaque thread calcule le produit d'une paire
  int indice = threadIdx.x + blockIdx.x * blockDim.
      x;
  tmp[threadIdx.x] = a[indice] * b[indice];

  //Synchronisation (dans le block)
  __syncthreads();

  //Le thread 0 effectue la somme
  if (0 == threadIdx.x){
    int sum = 0;
    for (int i = 0; i < THREADS_PER_BLOCK; i++)
      sum = sum + temp[i];
    *c = sum;
  }
}
```

Race condition

- c est dans la mémoire globale
- plusieurs thread 0 peuvent y accéder en même temps

Race condition

- c est dans la mémoire globale
- plusieurs thread 0 peuvent y accéder en même temps
- **Les opérations atomiques** :
  - Les opération de lecture, modification et écriture sont ininterruptibles
  - Plusieurs opérations atomiques possibles avec CUDA :

- `atomicAdd()`
- `atomicSub()`
- `atomicMin()`
- `atomicMax()`
- `atomicInc()`
- `atomicDec()`
- `atomicExch()`
- `atomicCAS()`

Produit scalaire (dot product)

```
#defin N (2048 * 2048)
#define THREAD_PER_BLOCK 512 //taille du tableau
__global__ void dot (int *a, int *b, int *c){
  __shared__ int tmp[THREADS_PER_BLOCK]
  // chaque thread calcule le produit d'une paire
  int indice = threadIdx.x + blockIdx.x * blockDim.
      x;
  tmp[threadIdx.x] = a[indice] * b[indice];

  //Synchronisation (dans le block)
  __syncthreads();

  //Le thread 0 effectue la somme
  if (0 == threadIdx.x){
    int sum = 0;
    for (int i = 0; i < THREADS_PER_BLOCK; i++)
      sum = sum + temp[i];
    atomicAdd(*c, sum);
  }
}
```

Produit scalaire (dot product)

```
int main (void){
  int *a, *b, *c;
  int *gpu_a, *gpu_b, *gpu_c;
  int size = N * sizeof(int);
  // allocation de l'espace pour le device}
  cudaMalloc((void **)&gpu_a, size);
  cudaMalloc((void **)&gpu_b, size);
  cudaMalloc((void **)&gpu_c, size);

  a=(int*) malloc (size);
  b=(int*) malloc (size);
  c=(int*) malloc (sizeof(int));
  random_ints(a, N);
  random_ints(b, N);
```

Produit scalaire (dot product)

```
// Copie des donnees vers le Device
cudaMemcpy (gpu_a,a,size,cudaMemcpyHostToDevice);
cudaMemcpy (gpu_b,b,size,cudaMemcpyHostToDevice);

//Lancement de l'operation avec THREAD_PER_BLOCK
    par block
add <<< N/THREAD_PER_BLOCK,THREAD_PER_BLOCK >>> (
    gpu_a, gpu_b, gpu_c);

//Copie du resultat
cudaMemcpy(c, gpu_c, size, cudaMemcpyDeviceToHost);

free(a); free(b);
cudaFree(gpu_a);
cudaFree(gpu_b);
cudaFree(gpu_c);
return 0;
}
```

Mesure du temps

- Les transferts de données sont synchrones
- L'appel au kernel ne l'est pas

```
cudaMemcpy(d_x, x, N*sizeof(float),
    cudaMemcpyHostToDevice);
cudaMemcpy(d_y, y, N*sizeof(float),
    cudaMemcpyHostToDevice);

t1 = myCPUTimer();
saxpy<<<(N+255)/256, 256>>>(N, 2.0, d_x, d_y);
cudaDeviceSynchronize();
t2 = myCPUTimer();

cudaMemcpy(y, d_y, N*sizeof(float),
    cudaMemcpyDeviceToHost);
```

Mesure du temps

- CUDA event API
- Les opérations sont séquentielles sur le GPU

```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
...
cudaEventRecord(start);
saxpy<<<(N+255)/256, 256>>>(N, 2.0f, d_x, d_y);
cudaEventRecord(stop);

cudaEventSynchronize(stop);//Garantit que l'
    evenement s'est execute
...
float milliseconds = 0;
cudaEventElapsedTime(&milliseconds, start, stop);
```

Exercice 5

Les optimisations de base

`http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/`

- ✓ Les paramètres du kernel
- ✓ Les instructions de contrôle
- × Les mémoires

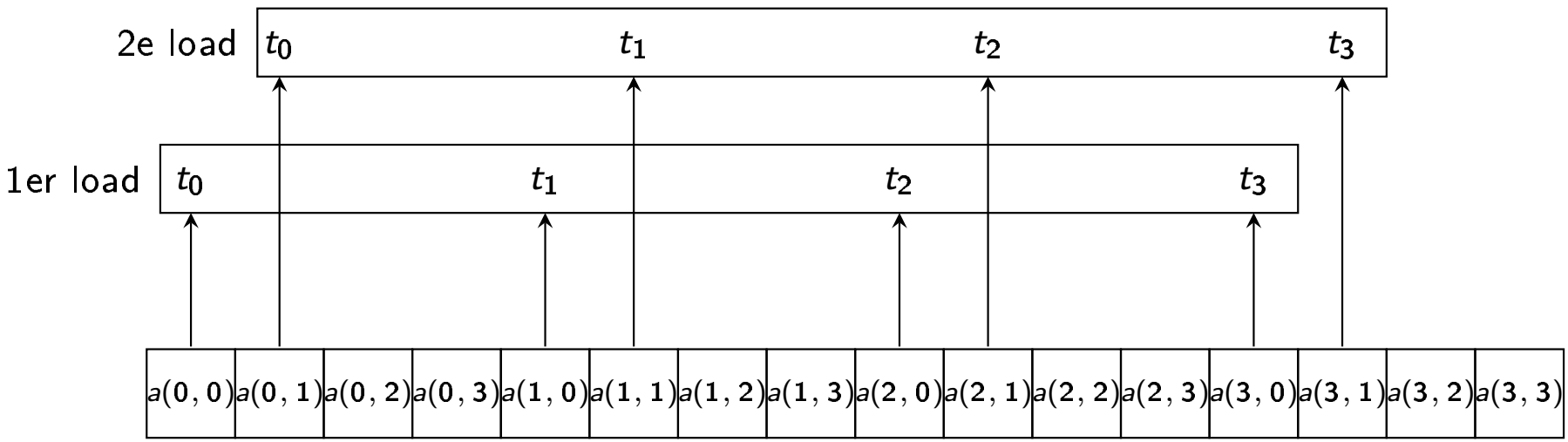
Throughput de la mémoire

1. Minimiser les transferts de faible BW
   → Minimiser les transfert Host<->Device

2. Minimiser les transferts mémoire globale<->Device
   → Favoriser la mémoire partagée et les caches
   - La mémoire partagée est équivalente à un cache géré par l'utilisateur

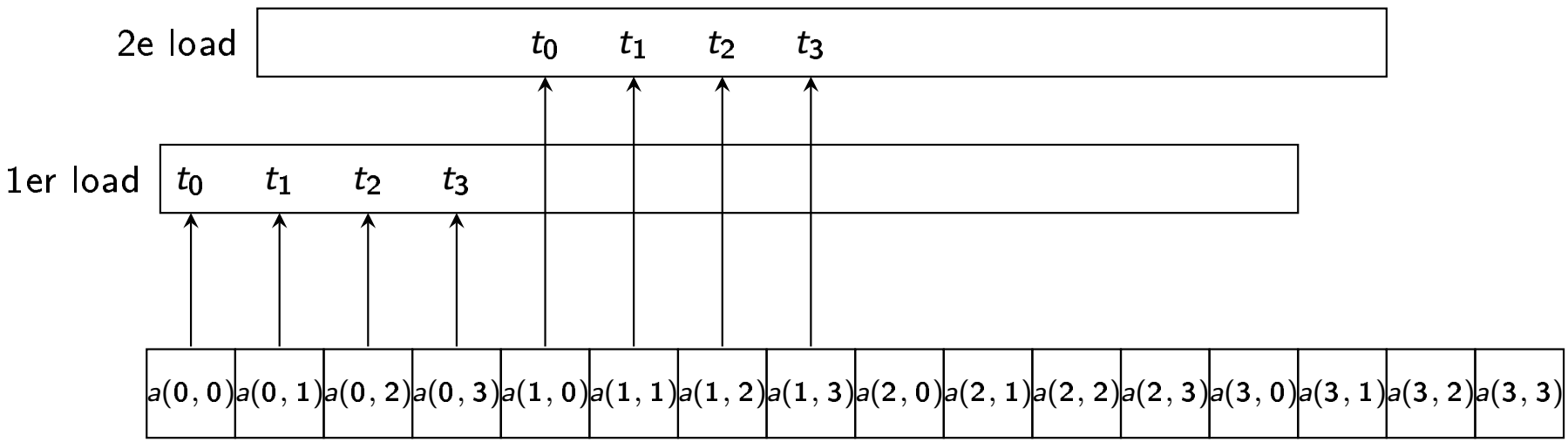
La mémoire globale

- Lorsque tous les threads font un load, le hardware détecte si les threads accèdent à un espace mémoire contigu
- Dans ce cas, le hardware groupe (*coalesces*) les accès en un seul accès à différentes location de la DRAM



La mémoire globale

- Lorsque tous les threads font un load, le hardware détecte si les threads accèdent à un espace mémoire contigu
- Dans ce cas, le hardware groupe (*coalesces*) les accès en un seul accès à différentes location de la DRAM

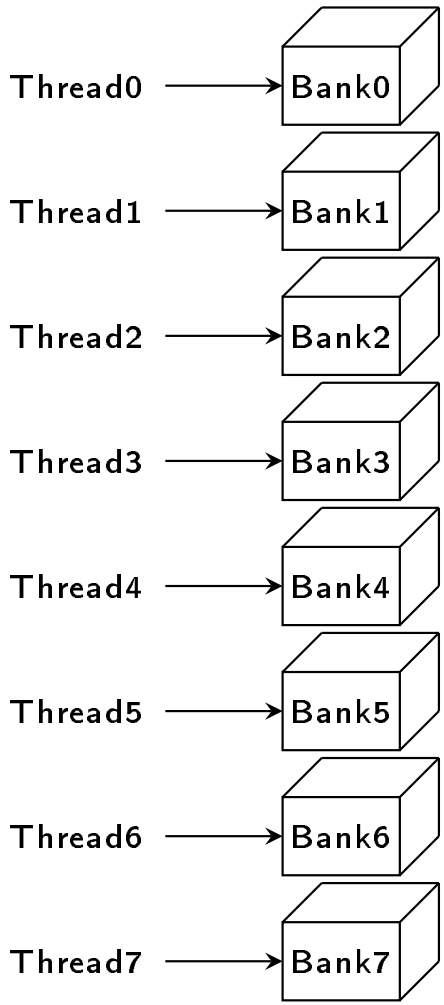


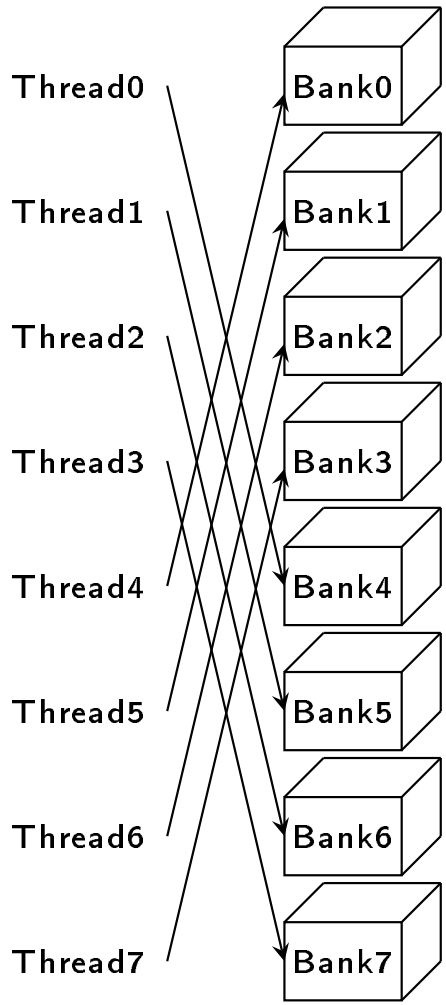
La mémoire partagée

- À utiliser pour éviter les accès non alignés
- La mémoire est partagée en modules de taille égale, appelés *bank*
- Il y a autant de *bank* que de threads dans un *warp*
- L'accès aux *bank* est simultané

→ Toute lecture/écriture de n adresses dans n bank différents est simultané

La mémoire partagée





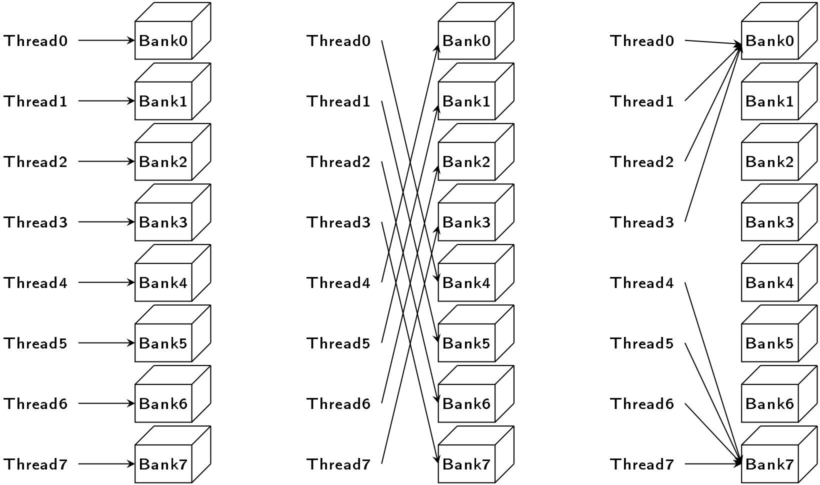
Bank conflict

- Si deux accès sont dans différentes adresses du même *bank*, il y a conflit
- ☹ L'accès en cas de conflit est sérialisé

- La mémoire partagée est rapide tant qu'il n'y a pas de *bank conflict*

- Le hardware divise un accès mémoire avec conflit en autant d'accès nécessaire pour ne plus avoir de conflit
- ☹ La BW est réduite par un facteur égal au nombre d'accès créés pour éviter les conflits

Bank conflict



Exercice 6

Nombre de threads dans les blocks pour le produit de matrices

- Chaque SM du Quadro K620 possède 48KB de mémoire partagée par block
- TILE_WIDTH = 16 :
  - 2 vecteur de mémoire partagée de float d'une taille de

  → 2 x 4B x 16 x 16 = 2KB pour un block
  - Donc jusqu'à 24 blocks pouvant s'exécuter sur le même SM
- TILE_WIDTH = 32
  - 2 vecteur de mémoire partagée de float d'une taille de

  → 2 x 4B x 32 x 32 = 8KB pour un block
  - Donc jusqu'à 6 blocks pouvant s'exécuter sur le même SM
- Mais dans la vraie vie, on utilise des bibliothèques (cuBlas)

Accès à la mémoire

Différents accès à la mémoire

- Accès via des `memcpy` explicites
  - ☺ Bonnes performances (car les données sont disponibles)
  - ☹ Code assez lourd et erreurs fréquentes
  - ☹ Accès uniquement à la mémoire du GPU
- Accès via un mécanisme *Zero Copy* (non abordé dans ce cours) : les thread GPU accèdent directement à l'espace mémoire du CPU (via les fonctions `cudaHostAlloc` et `cudaHostGetDevicePointer`).
  - ☺ Accès à toute la mémoire
  - ☹ Vitesse limitée par le bus PCI et NVLink
  - ☹ Pas de localité des données
- Mémoire unifiée : une même zone mémoire accessible directement depuis le CPU et le GPU (le meilleur des deux mondes)

Référence : `https://developer.nvidia.com/blog/maximizing-unified-memory-performance-cuda/`

Exemple d'utilisation de la mémoire unifiée

```
int *a, *b;
int *gpu_a, *gpu_b;
cudaMalloc((void **)&gpu_a, N*sizeof(int));
cudaMalloc((void **)&gpu_b, N*sizeof(int));
a=(int*) malloc (N*sizeof(int));
b=(int*) malloc (N*sizeof(int));
for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }
cudaMemcpy (gpu_a,a,N*sizeof(int),cudaMemcpyHostToDevice);
cudaMemcpy (gpu_b,b,N*sizeof(int),cudaMemcpyHostToDevice);
add <<< N, 1 >>> (gpu_a, gpu_b);
cudaMemcpy(b, gpu_b, N*sizeof(int), cudaMemcpyDeviceToHost);
free(a); free(b);
cudaFree(gpu_a);
cudaFree(gpu_b);
```

Exemple d'utilisation de la mémoire unifiée

```
int *a, *b;
int *gpu_a, *gpu_b;
cudaMalloc((void **)&gpu_a, N*sizeof(int));
cudaMalloc((void **)&gpu_b, N*sizeof(int));
a=(int*) malloc (N*sizeof(int));
b=(int*) malloc (N*sizeof(int));
for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }
cudaMemcpy (gpu_a,a,N*sizeof(int),cudaMemcpyHostToDevice);
cudaMemcpy (gpu_b,b,N*sizeof(int),cudaMemcpyHostToDevice);
add <<< N, 1 >>> (gpu_a, gpu_b);
cudaMemcpy(b, gpu_b, N*sizeof(int), cudaMemcpyDeviceToHost);
free(a); free(b);
cudaFree(gpu_a);
cudaFree(gpu_b);
```

Exemple d'utilisation de la mémoire unifiée

```
int *a, *b;
cudaMallocManaged(a, N*sizeof(int));
cudaMallocManaged(b, N*sizeof(int));
for (int i = 0; i < N; i++) { // operations effectuees
    par le CPU
    x[i] = 1.0f;
    y[i] = 2.0f;
  }
add <<< N, 1 >>> (a, b);

cudaFree(a);
cudaFree(b);
```

Référence : https://developer.nvidia.com/blog/unified-memory-cuda-beginners/

Exemple d'utilisation de la mémoire unifiée

```
int *a, *b;
cudaMallocManaged(a, N*sizeof(int));
cudaMallocManaged(b, N*sizeof(int));
for (int i = 0; i < N; i++) { // operations effectuees
    par le CPU
    x[i] = 1.0f;
    y[i] = 2.0f;
  }
add <<< N, 1 >>> (a, b);
cudaDeviceSynchronize();
cudaFree(a);
cudaFree(b);
```

Référence : https://developer.nvidia.com/blog/unified-memory-cuda-beginners/

Pourquoi on s'est embêtés à parler des `memcpy`

- La mémoire unifiée n'est disponible que depuis CUDA 6.0
- Selon les architectures, le fonctionnement n'est pas le même (plus d'explication au slide suivant)

Fonctionnement de la mémoire unifiée

1. Pré-Pascal
   - Allocation de tous l'espaces (les pages) sur le GPU
   - Lorsque le CPU rempli le tableau (la boucle for), les pages doivent être chargées sur le CPU (donc défauts de page)
   - Lorsque le kernel est appelé, les pages sont de nouveau rechargées sur le GPU (les défauts de page de sont pas possibles sur les anciennes architectures)
2. Pascal
   - Les pages peuvent n'être allouées que lors de l'accès
   - Les défauts de page sont possibles (donc de l'overhead)

   $\Rightarrow$ Pour éviter cela, nous pouvons utiliser des techniques de prefetching, ou initialiser le tableau dans un kernel GPU

Mémoire unifiée : le prefetching

- La migration on-demand permet un meilleur overlap entre le transfert des données et le calcul, mais il y les défauts de page.
- Si le schéma d'accès aux données est connu, il est possible d'utiliser `cudaMemPrefetchAsync`.

Référence : `https://developer.nvidia.com/blog/maximizing-unified-memory-performance-cuda/`

Parallélisme de tâches avec des GPU : les streams

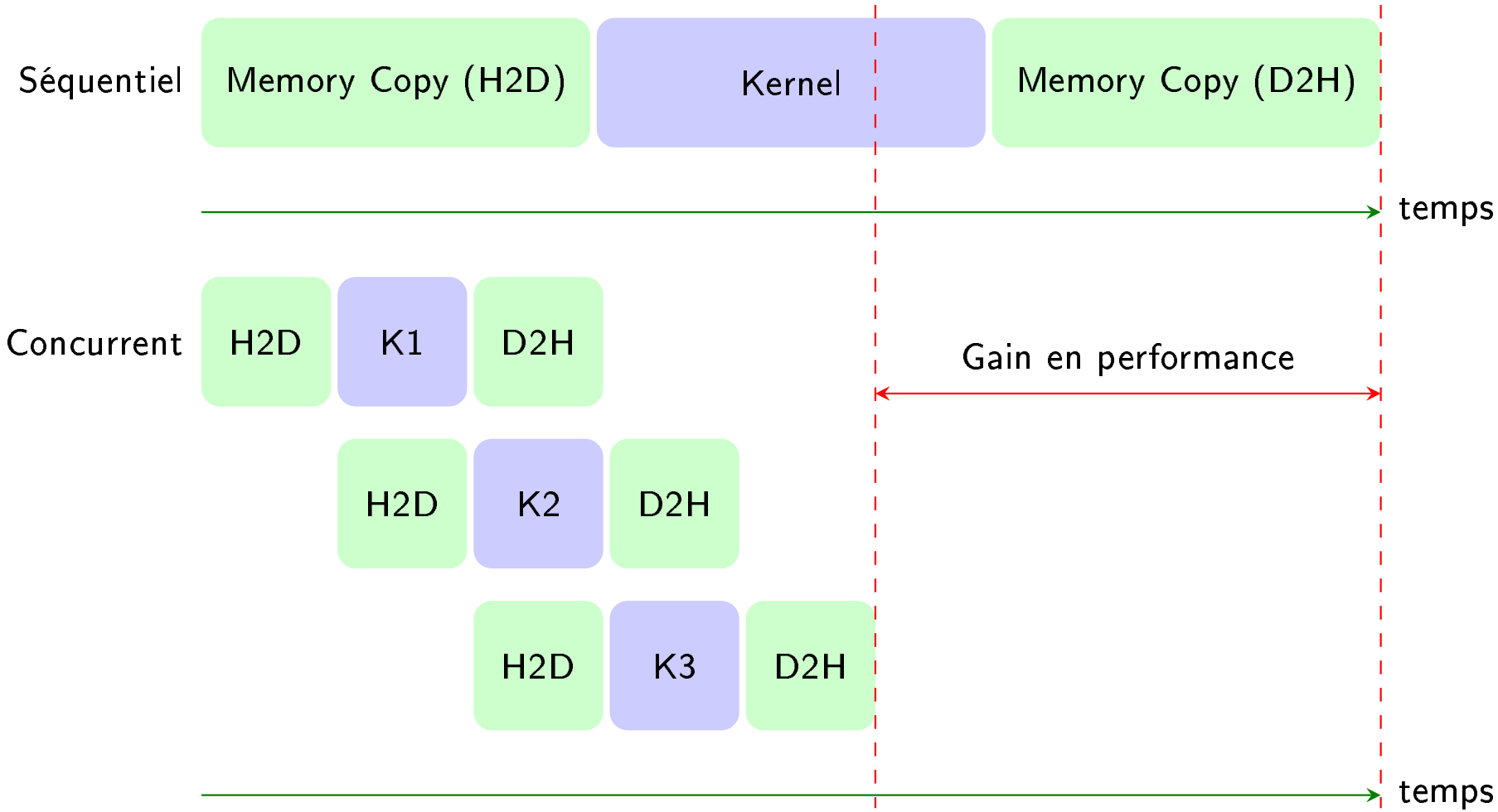
Les streams

- Un stream : Une séquence d'opérations (comme une file d'attente pour le device)
- Stream par défaut : stream 0
  - Opérations de lecture et d'écriture synchrone (*HostToDevice* et *DeviceToHost*) avec le host
  - Les *kernels* sont asynchrones avec le host par défaut (des opérations CPU sont possibles)

Les streams

- Stream différents du stream 0
- Les opérations sur un même stream sont ordonnées (FIFO) et syncrhones
- Les opérations dans des streams différents peuvent s'exécuter en parallèle
- Les opérations entre les différents streams peuvent s'intercaler

Exemple d'exécution

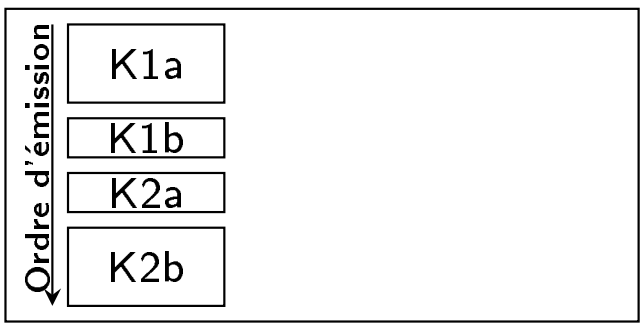


Exemple de code avec streams

```
cudaStream_T stream1, stream2, stream3, stream4;
cudaStreamCreate(&stream1);
...
cudaMalloc(&data_dev1, size);
...
cudaMemcpyAsync(data_dev1, data_host1, size,
    cudaMemcpyHostToDevice, stream1);
kernel2 <<< grid, block, 0, stream2 >>> (...,
    data_dev2, ...);
kernel3 <<< grid, block, 0, stream3 >>> (...,
    data_dev3, ...);
cudaMemcpyAsync(data_host4, data_dev4, size,
    cudaMemcpyDeviceToHost, stream4);
...
```

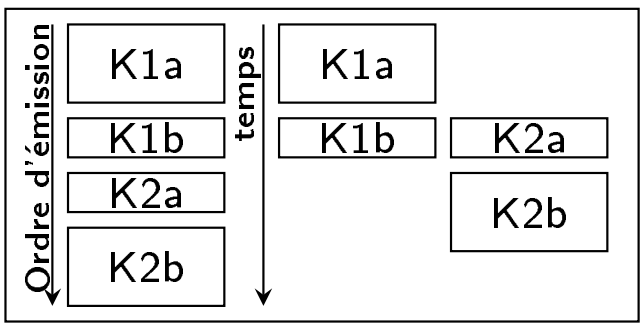
L'ordre d'émission des opérations est important

- Stream 1 : K1a, K1b
- Stream 2 : K2a, K2b



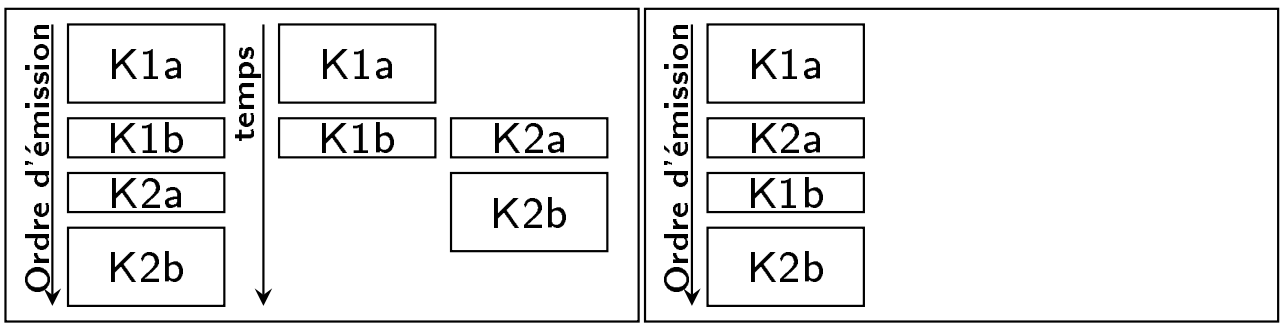
L'ordre d'émission des opérations est important

- Stream 1 : K1a, K1b
- Stream 2 : K2a, K2b



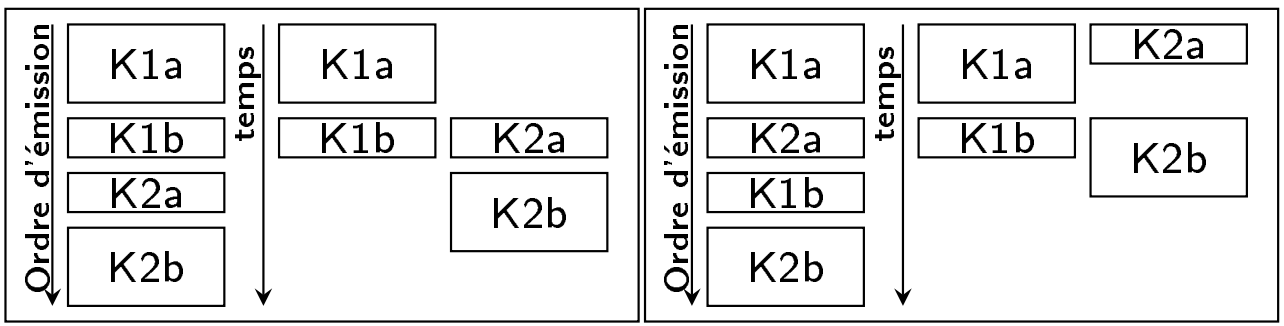
L'ordre d'émission des opérations est important

- Stream 1 : K1a, K1b
- Stream 2 : K2a, K2b



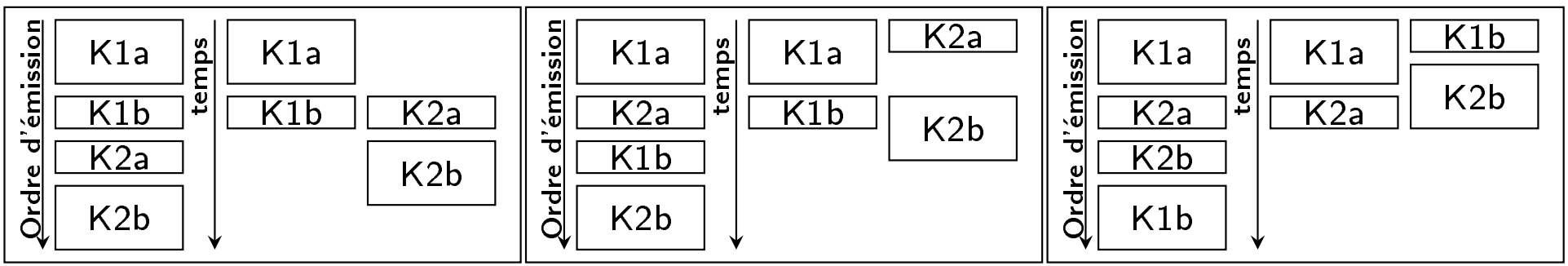
L'ordre d'émission des opérations est important

- Stream 1 : K1a, K1b
- Stream 2 : K2a, K2b



L'ordre d'émission des opérations est important

- Stream 1 : K1a, K1b
- Stream 2 : K2a, K2b

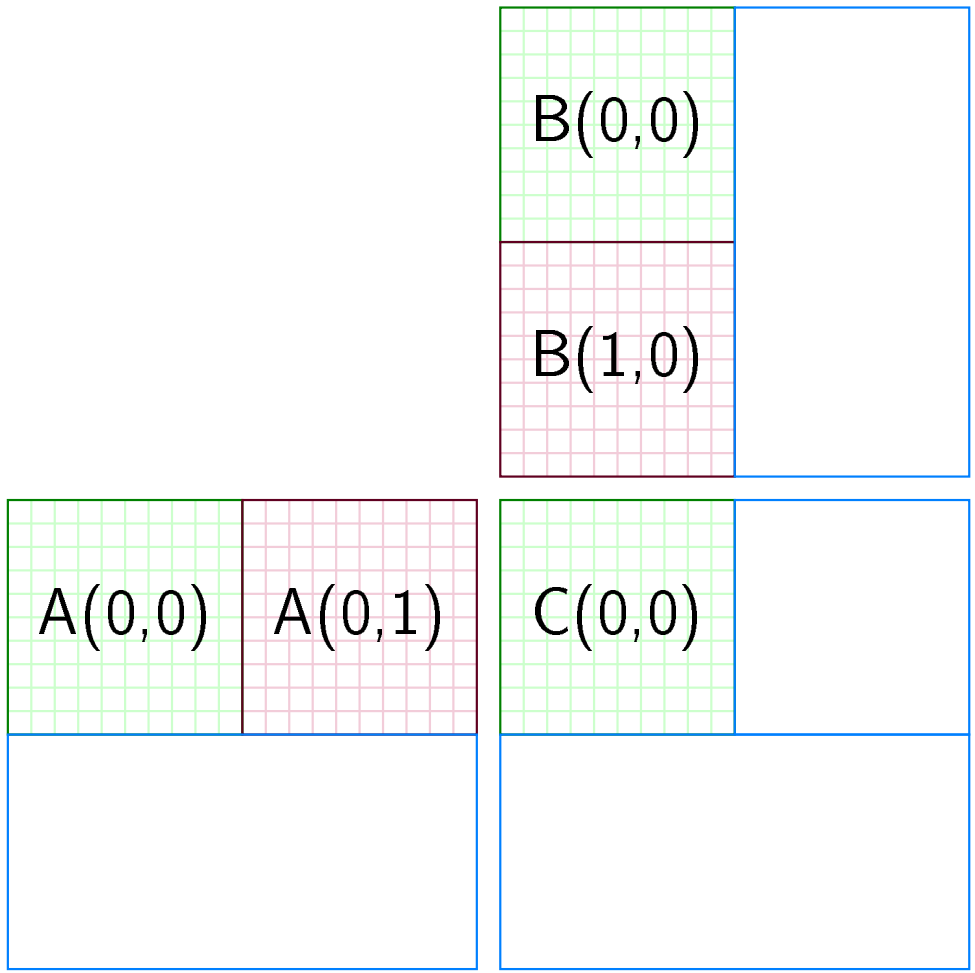


Pour aller plus loin

- MPI + CUDA :
  `http://on-demand.gputechconf.com/gtc/2014/presentations/S4236-multi-gpu-programming-mpi.pdf`
- Mémoire unifiée CPU + GPU :
  `http://www.drdobbs.com/parallel/unified-memory-in-cuda-6-a-brief-overvie/240169095?pgno=1`

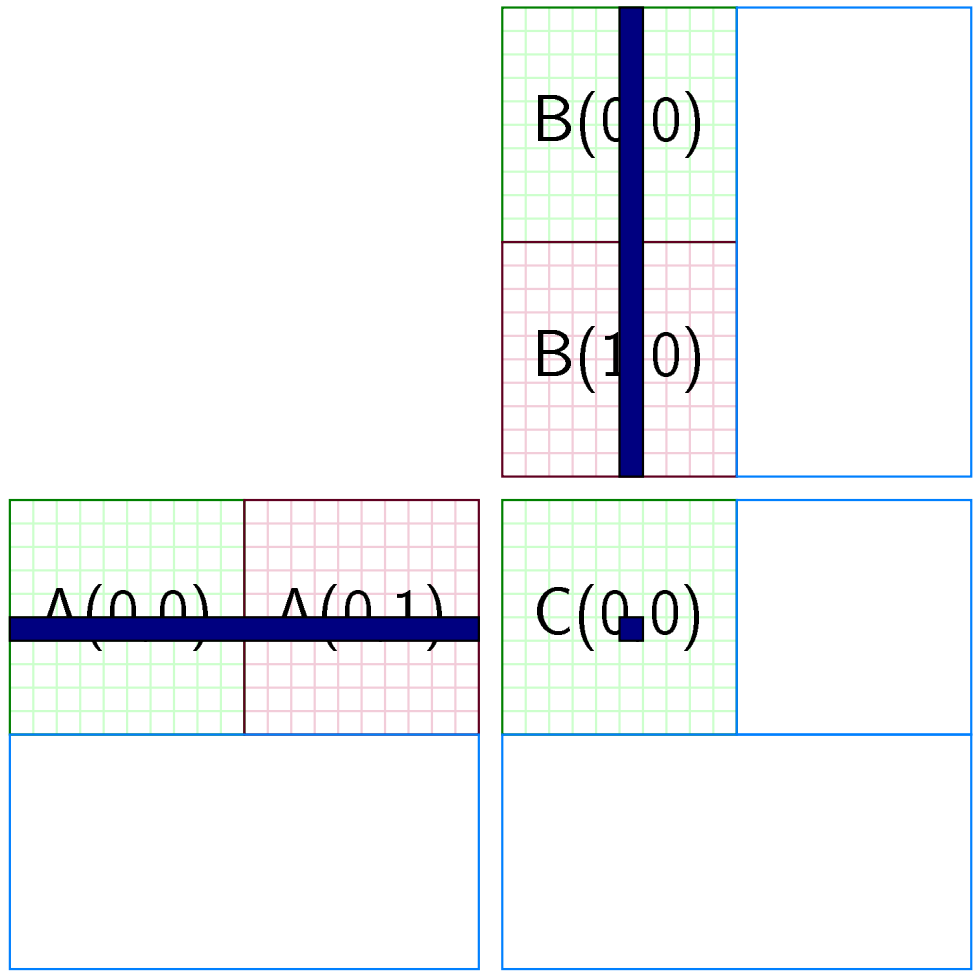
Multiplication de matrice

- C(0,0) -> blockIdx.x=0,blockIdx.y=0
- C(0,1) -> blockIdx.x=1,blockIdx.y=0
- C(1,0) -> blockIdx.x=0,blockIdx.y=1
- C(1,1) -> blockIdx.x=1,blockIdx.y=1



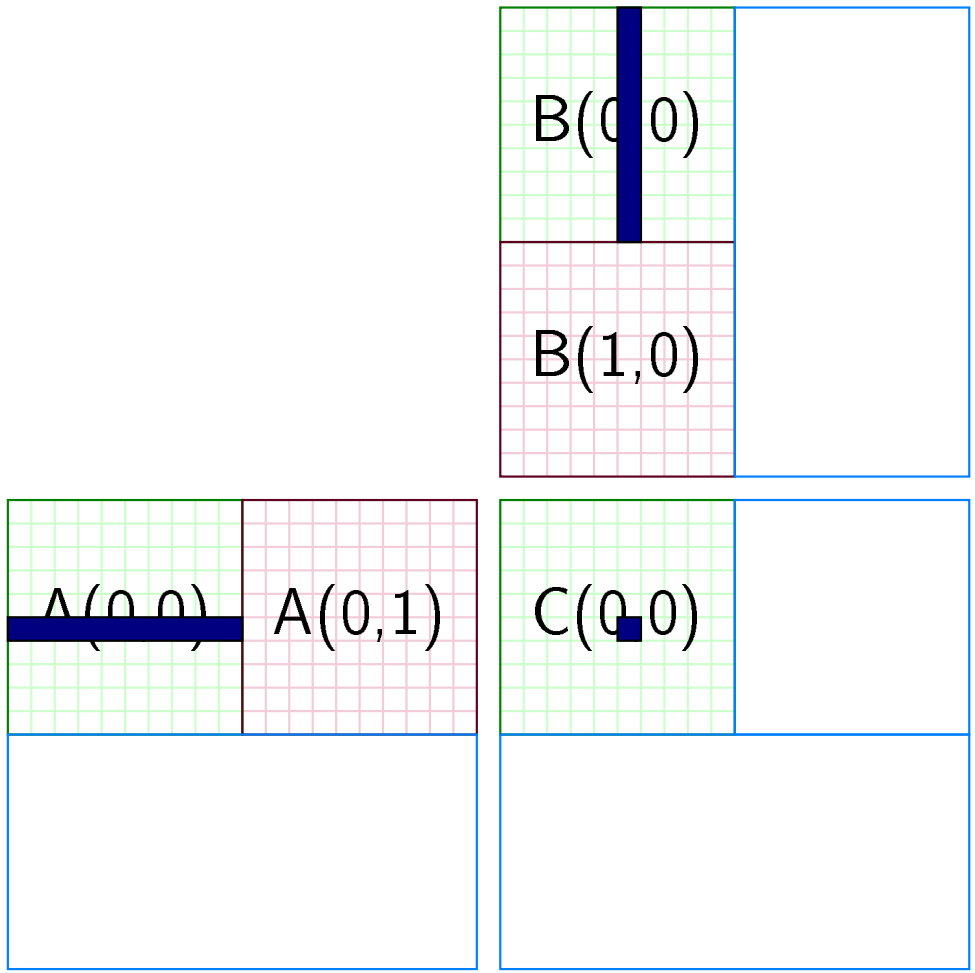
Multiplication de matrice

- 1 block calcule une tuile
- 1 thread calcule un élément de la tuile



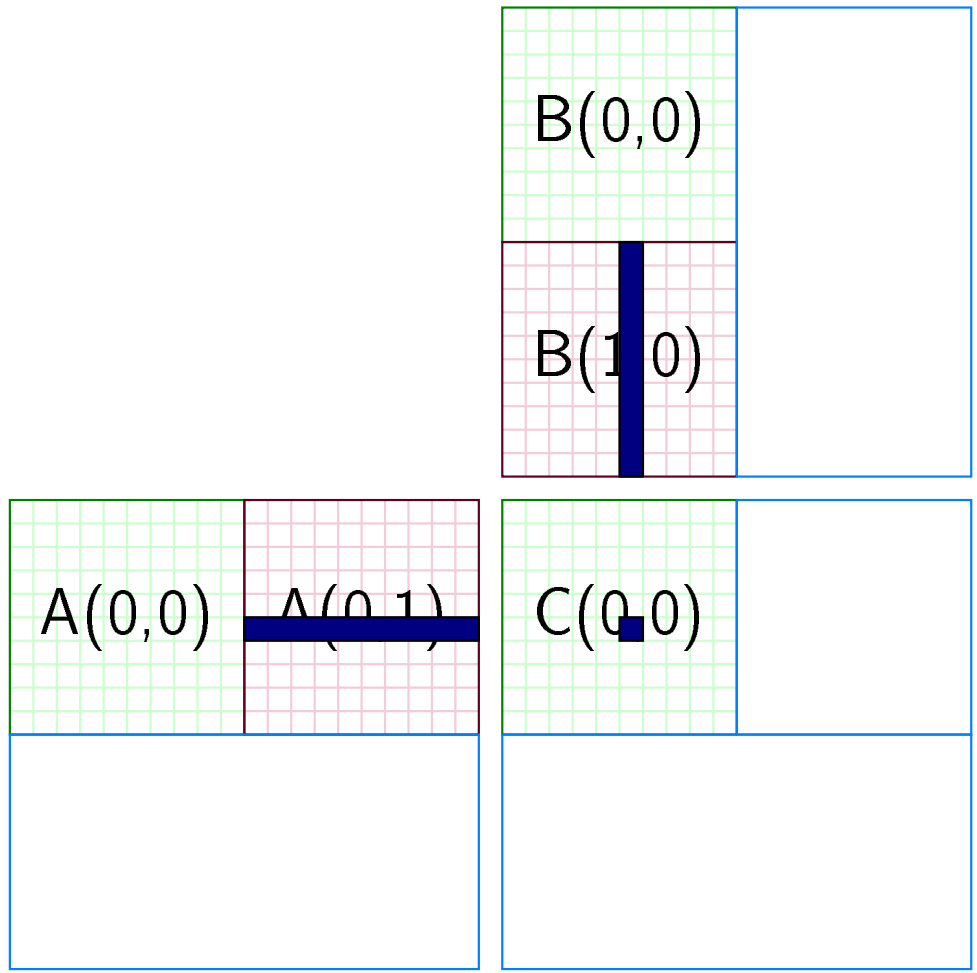
Multiplication de matrice

- Les blocks A(0,0) et B(0,0) sont chargés en mémoire partagée de façon collaborative par les threads (chacun charge un élément de chaque tuile)
- Les threads font le calcul partiel de C
- Chaque valeur est gardée dans le registre du thread



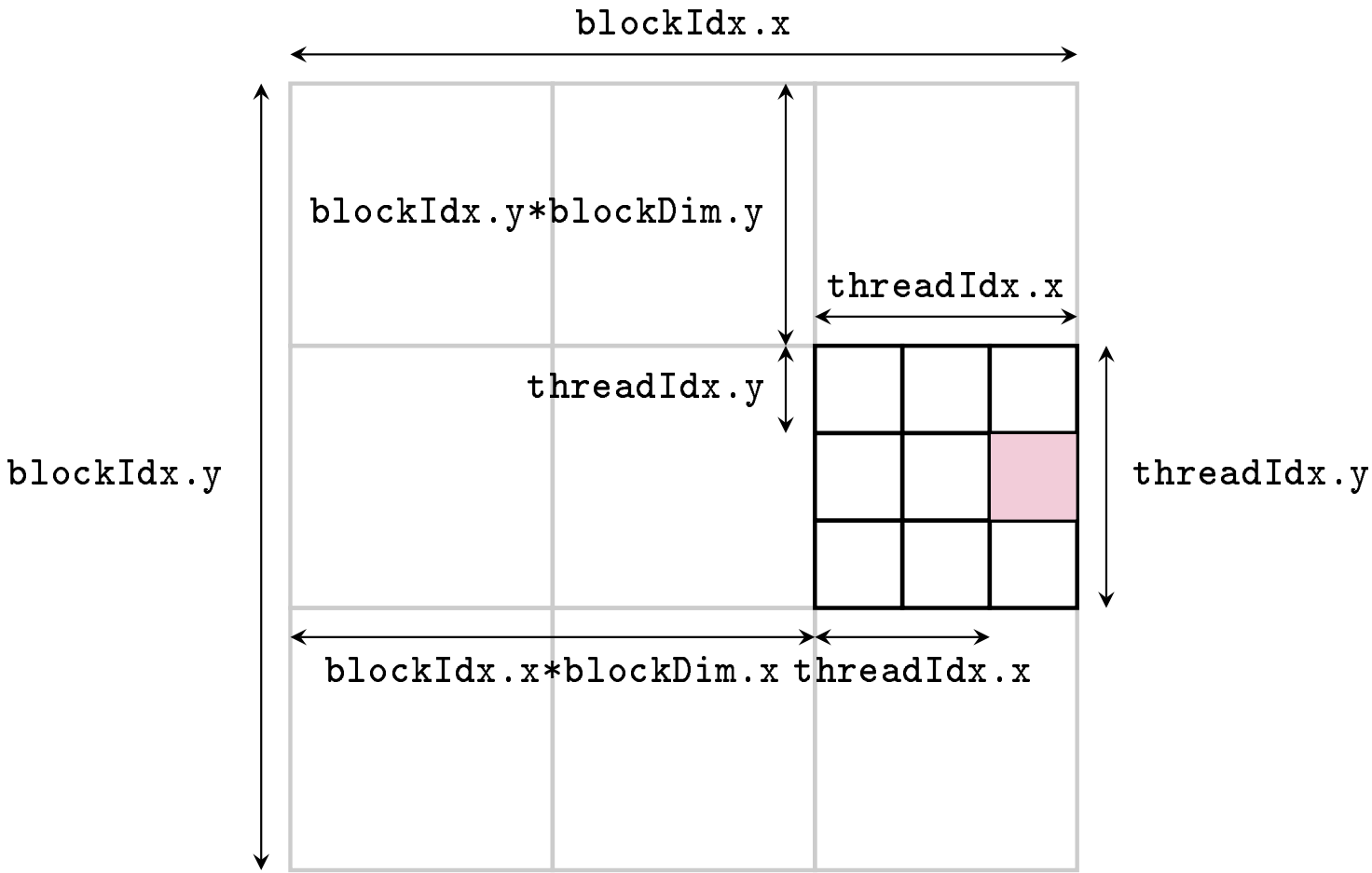
Multiplication de matrice

- Les blocks A(0,1) et B(1,0) sont chargés en mémoire partagée
- Les threads terminent le calcul de C
- À la fin, C est stockée en mémoire globale



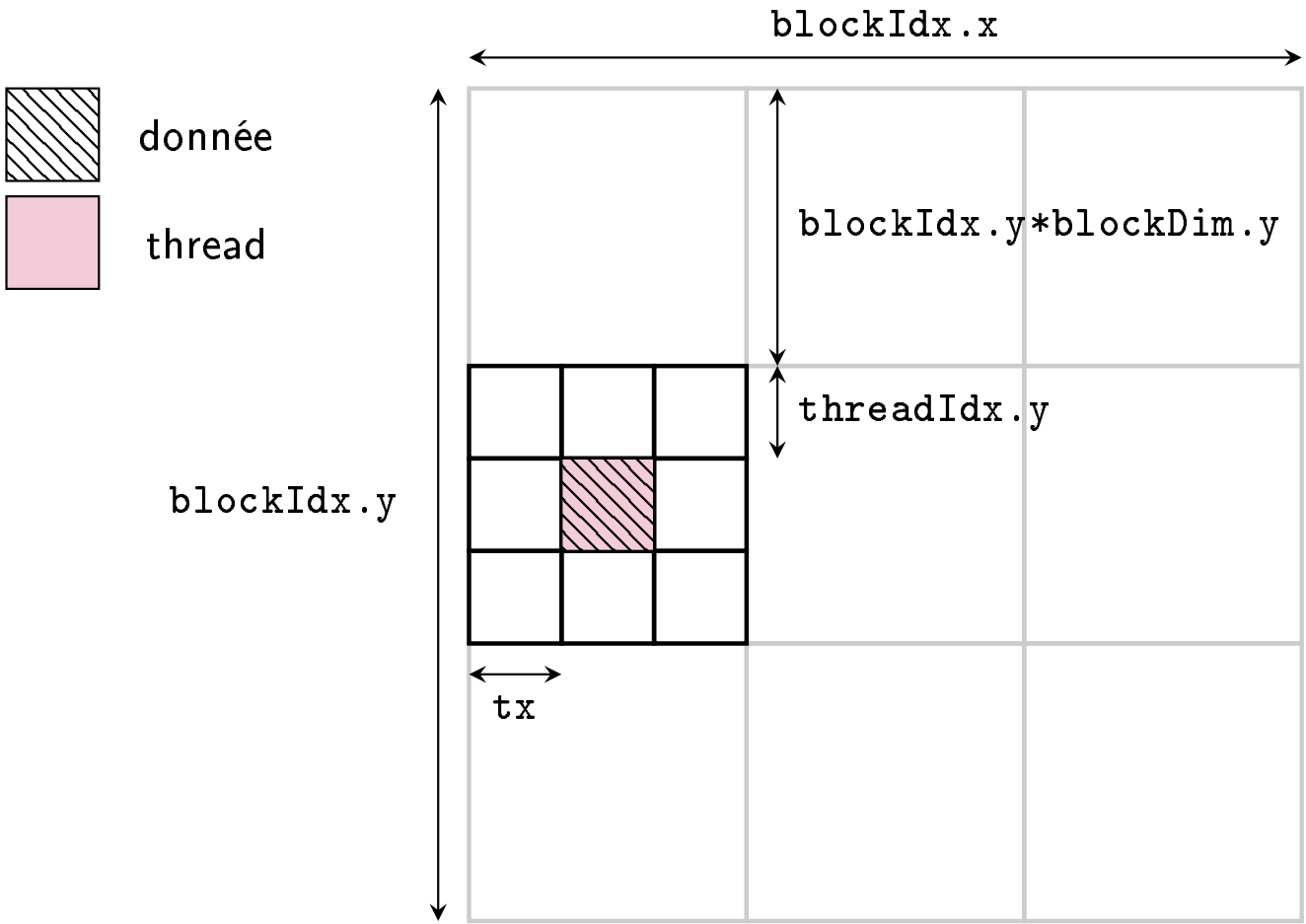
Les indices dans le produit de matrices par tuile : A

- `col = blockIdx.x*blockDim.x+theadIdx.x`
- `ligne = blockIdx.y*blockDim.y+theadIdx.y`



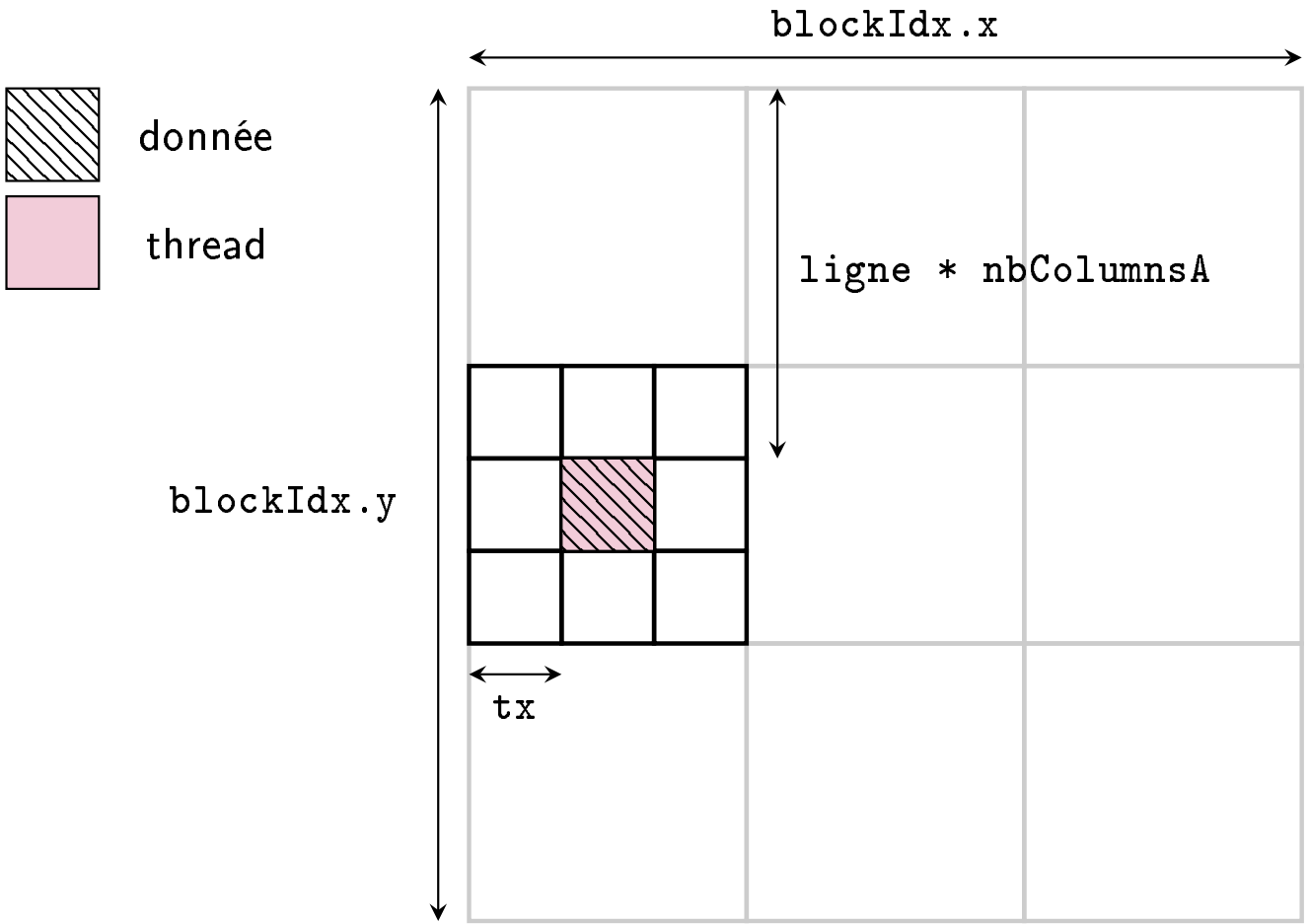
Les indices dans le produit de matrices par tuile : A

- `col = blockIdx.x*blockDim.x+theadIdx.x`
- `ligne = blockIdx.y*blockDim.y+theadIdx.y`
- `TILE_WIDTH = blockDim.x`



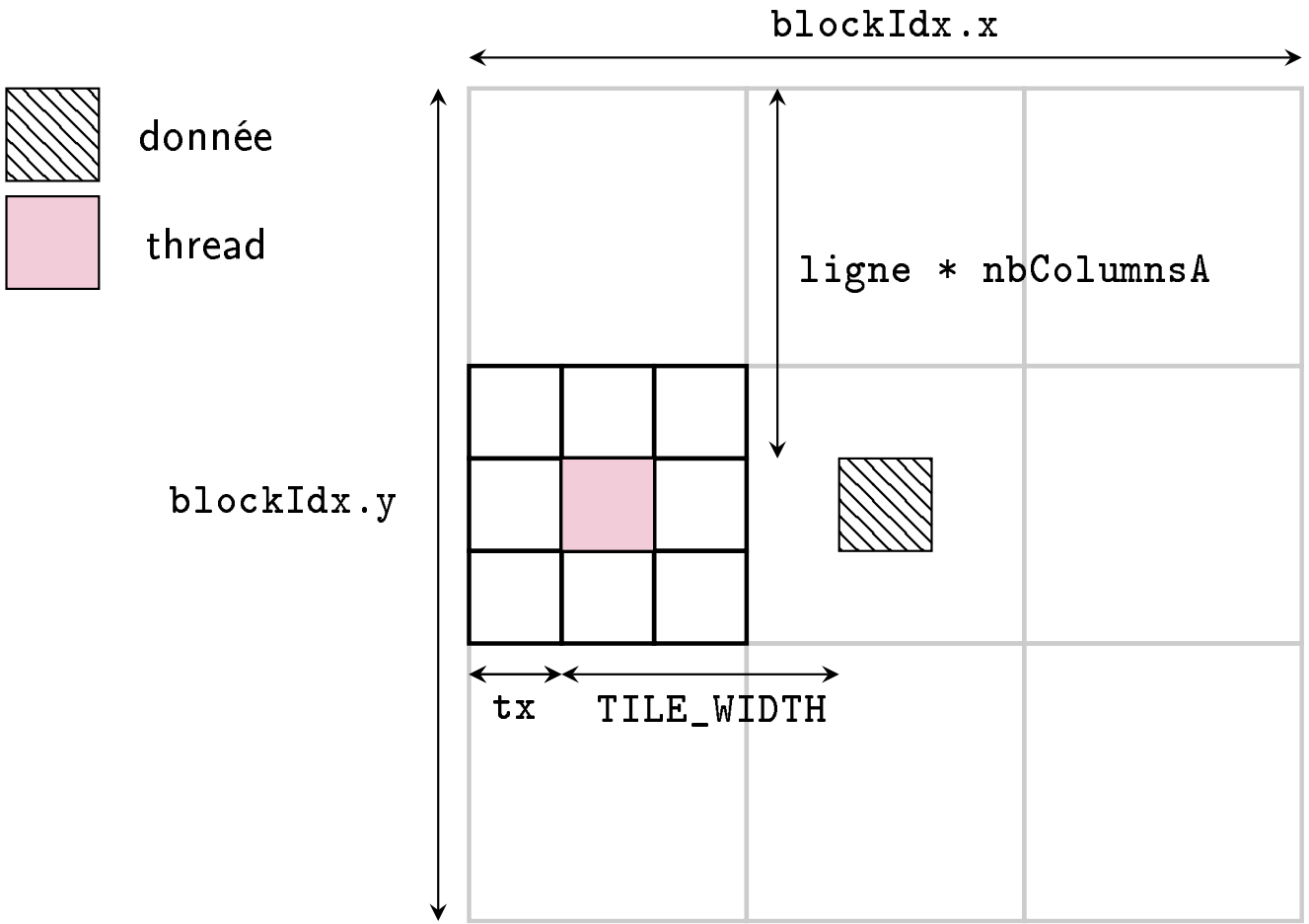
Les indices dans le produit de matrices par tuile : A

- `col = blockIdx.x*blockDim.x+theadIdx.x`
- `ligne = blockIdx.y*blockDim.y+theadIdx.y`
- `TILE_WIDTH = blockDim.x`



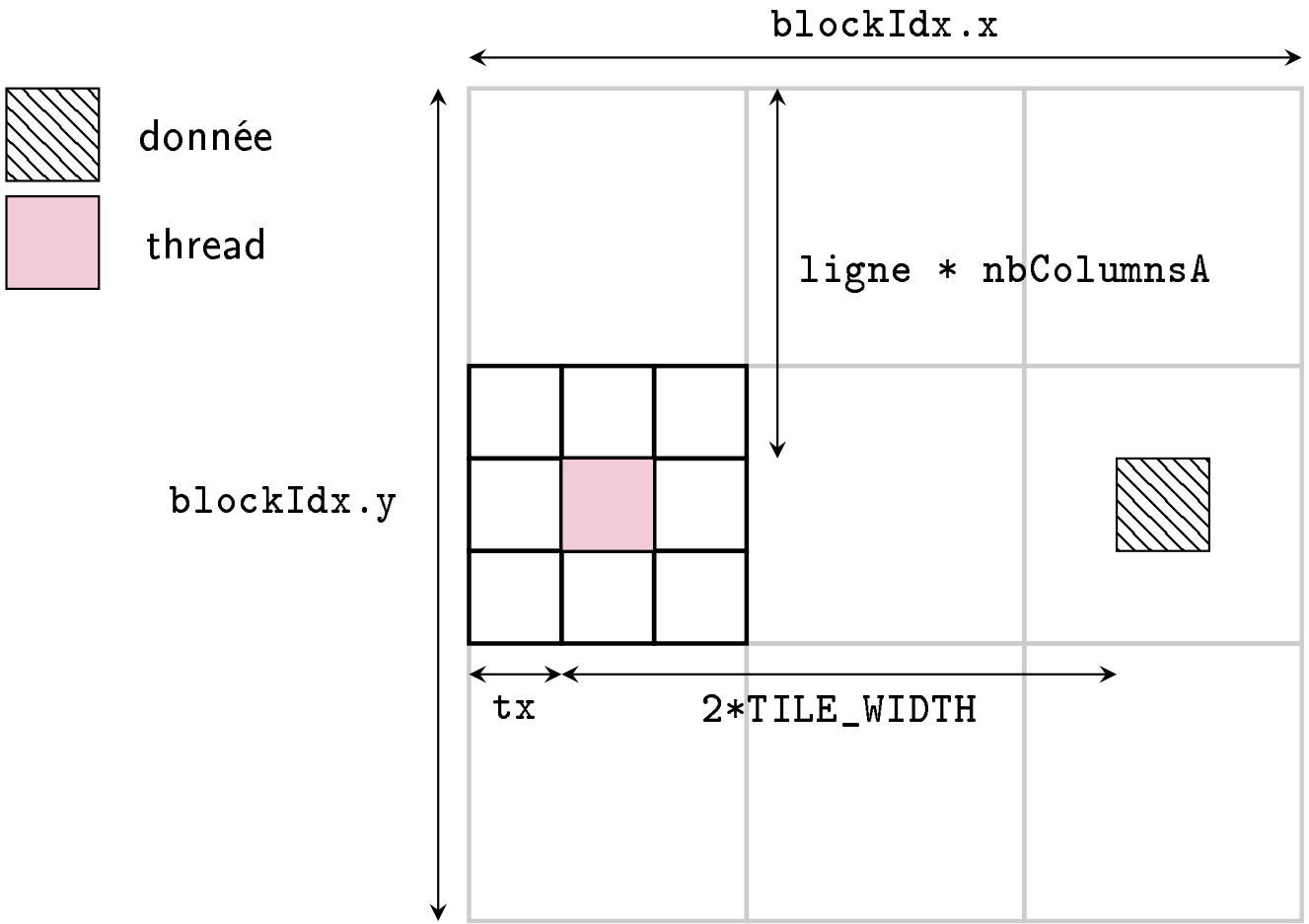
Les indices dans le produit de matrices par tuile : A

- `col = blockIdx.x*blockDim.x+theadIdx.x`
- `ligne = blockIdx.y*blockDim.y+theadIdx.y`
- `TILE_WIDTH = blockDim.x`



Les indices dans le produit de matrices par tuile : A

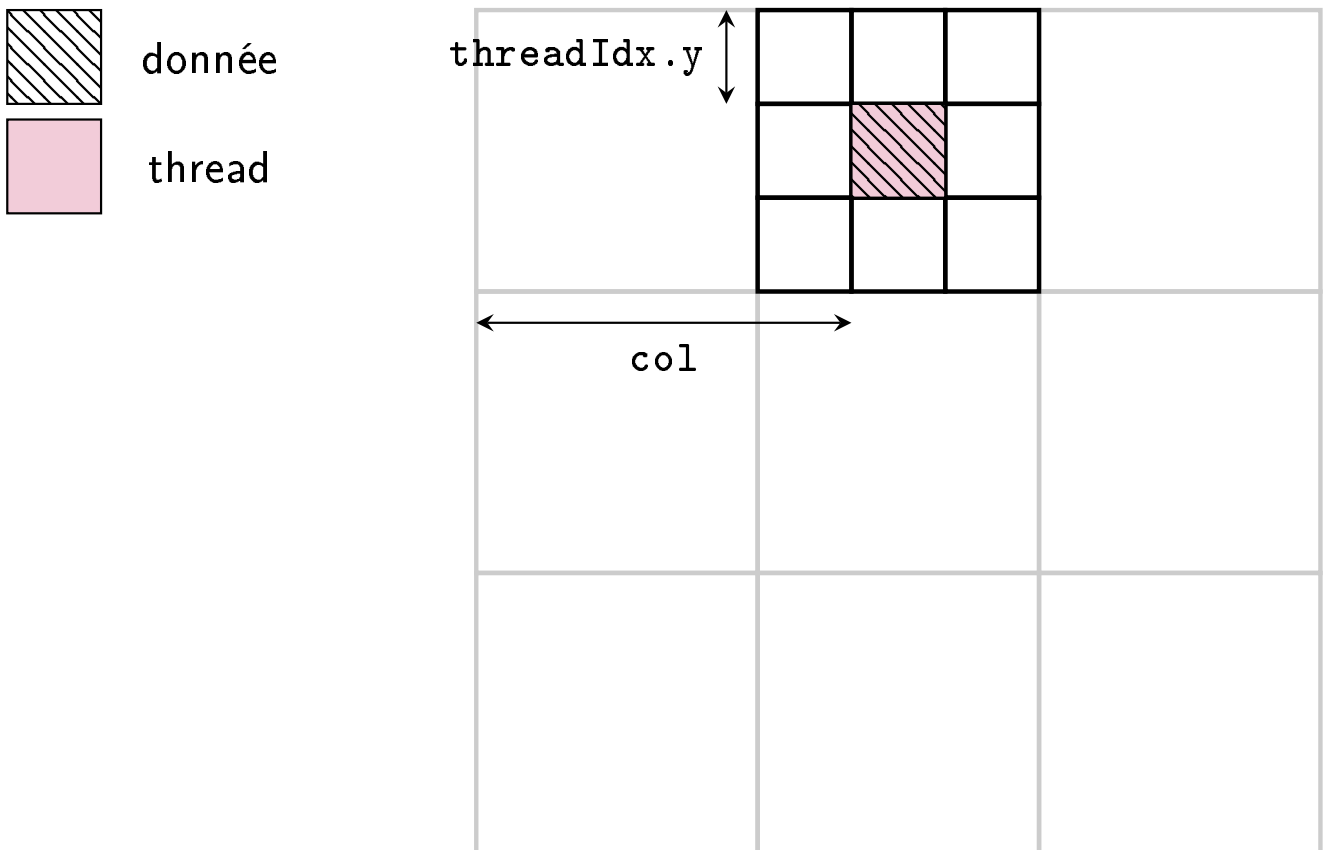
- `col = blockIdx.x*blockDim.x+theadIdx.x`
- `ligne = blockIdx.y*blockDim.y+theadIdx.y`
- `TILE_WIDTH = blockDim.x`



Les indices dans le produit de matrices par tuile : A

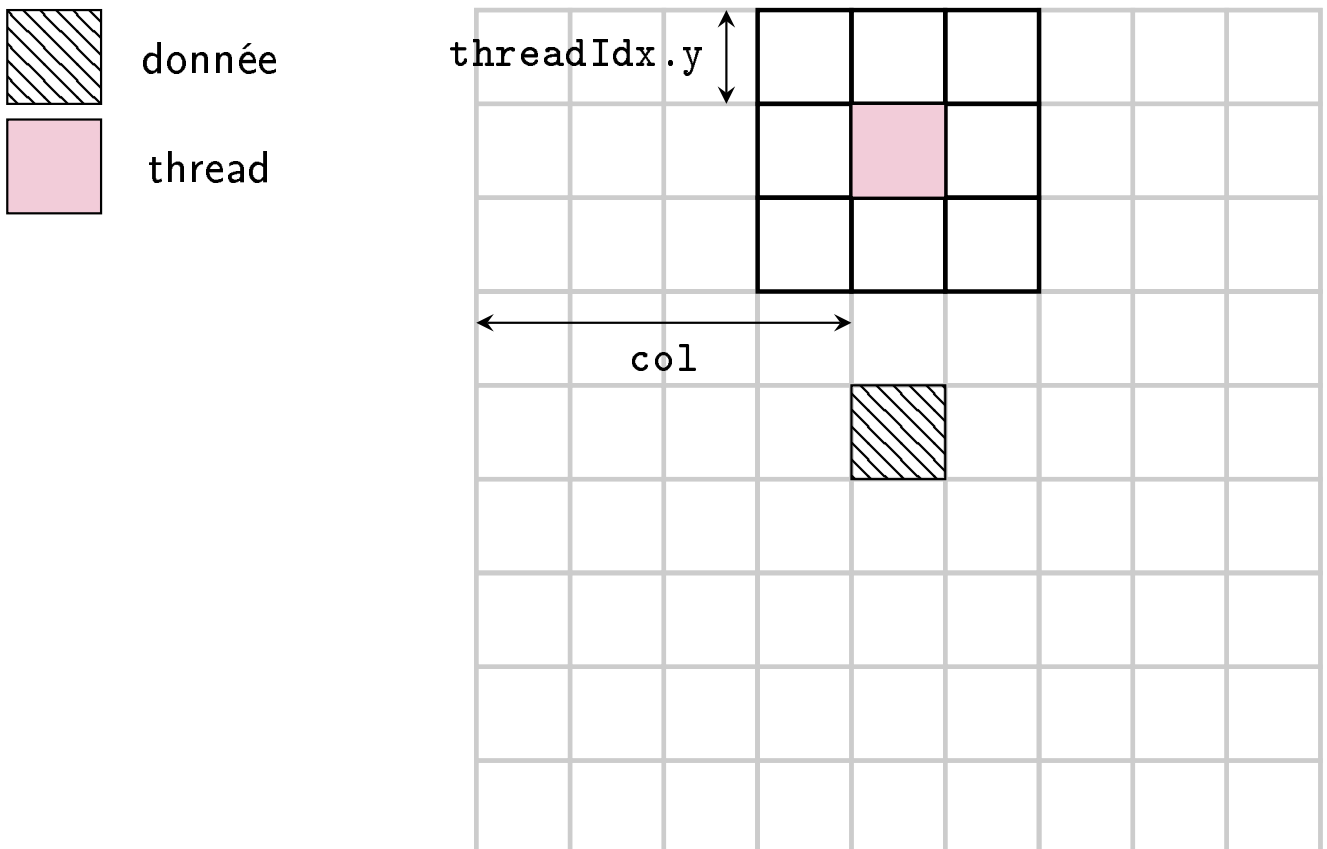
- `col = blockIdx.x*blockDim.x+theadIdx.x`
- `ligne = blockIdx.y*blockDim.y+theadIdx.y`
- `TILE_WIDTH = blockDim.x`
- `indice = ligne * nb_colA + TILE_WIDTH * tileId + threadIdx.x`

Les indices dans le produit de matrices par tuile : B



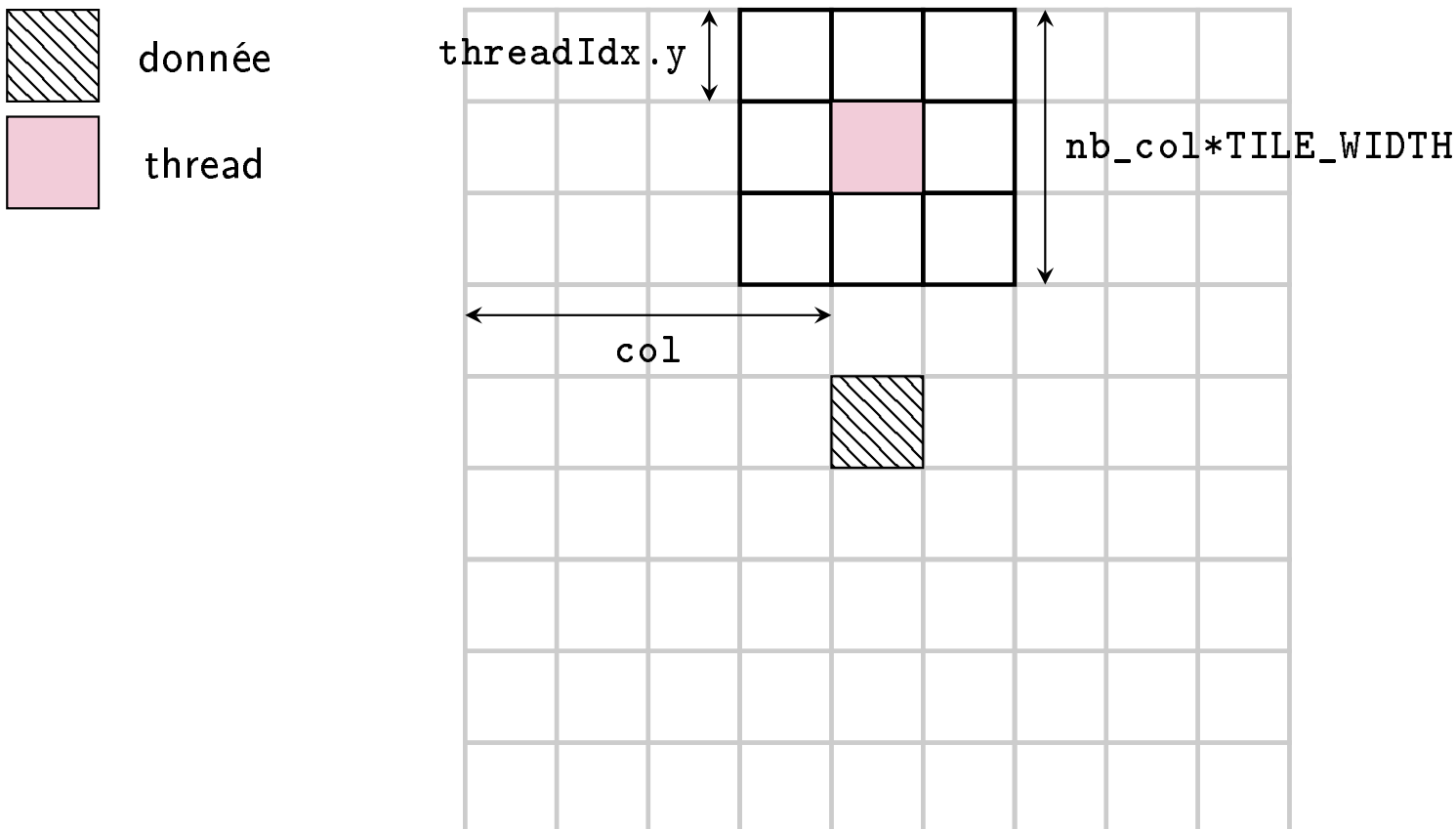
Les indices dans le produit de matrices par tuile : B

- La matrice est stockée par ligne :
  - Les 2 éléments sont séparés par le nombre d'éléments dans une ligne multiplié par la taille de la tuile
  - Une fois dans la bonne tuile, il faut accéder à la bonne ligne, en n'oubliant pas que le stockage est fait par ligne (`ty*nb_col`)



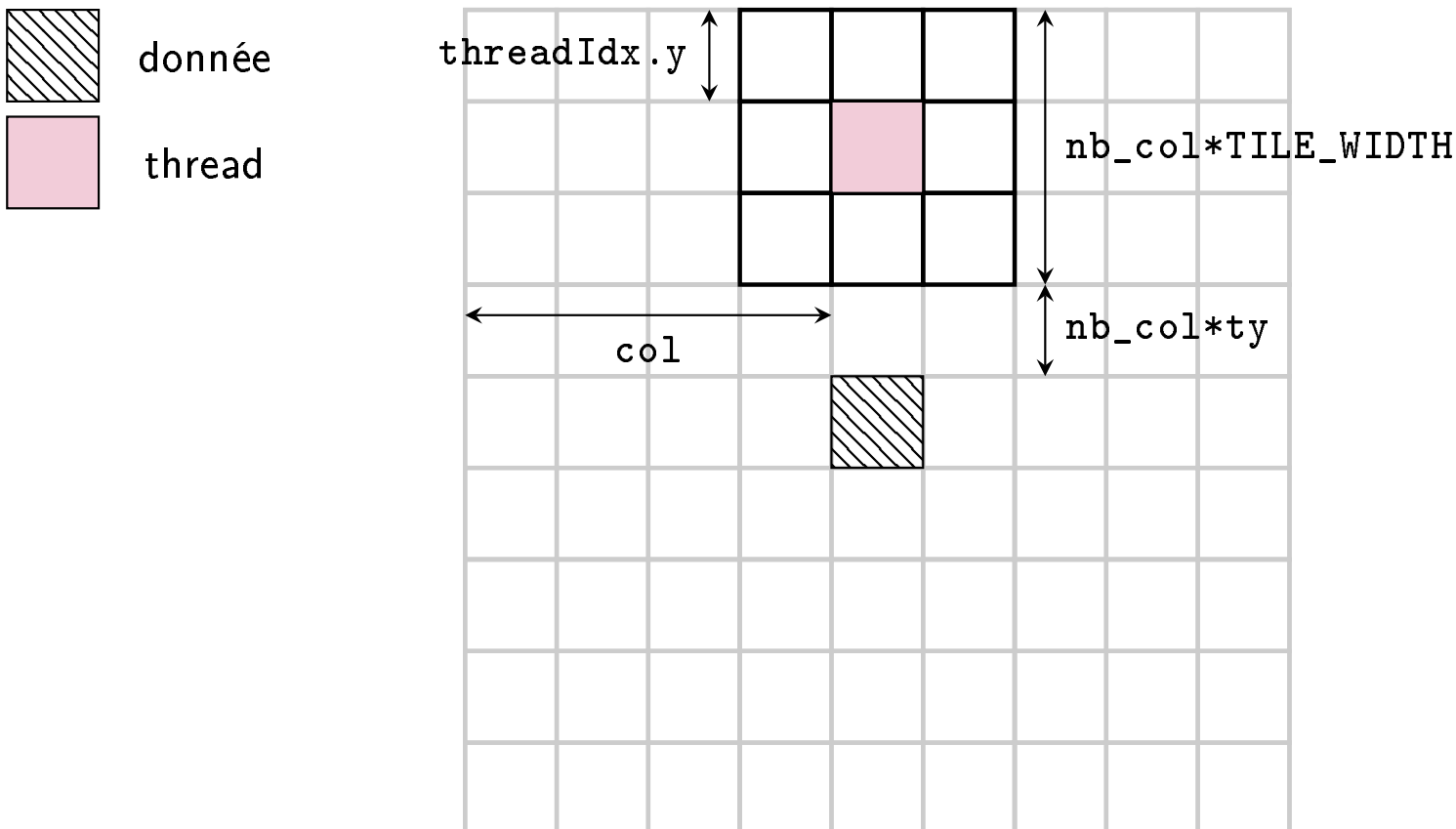
Les indices dans le produit de matrices par tuile : B

- La matrice est stockée par ligne :
  - Les 2 éléments sont séparés par le nombre d'éléments dans une ligne multiplié par la taille de la tuile
  - Une fois dans la bonne tuile, il faut accéder à la bonne ligne, en n'oubliant pas que le stockage est fait par ligne (`ty*nb_col`)



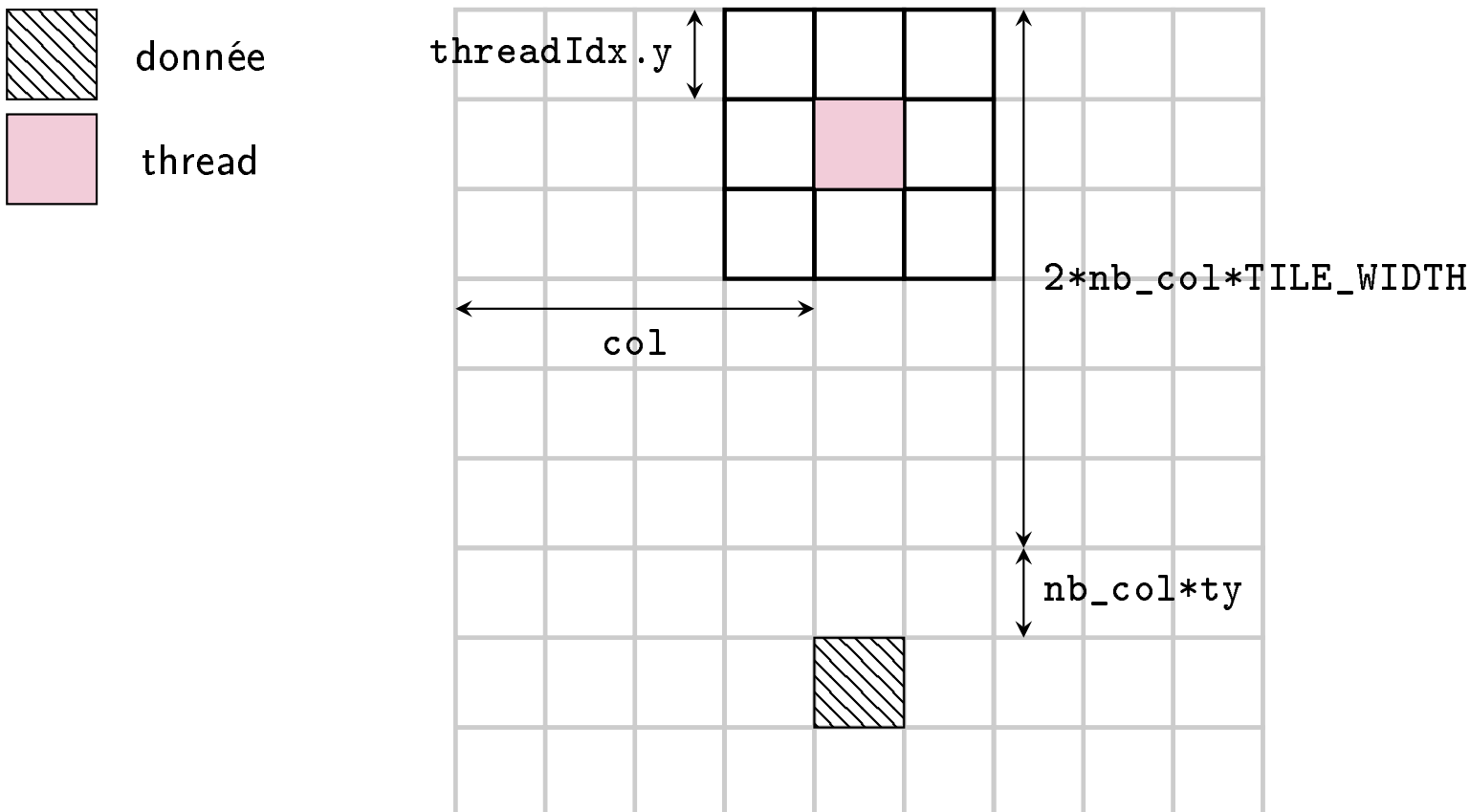
Les indices dans le produit de matrices par tuile : B

- La matrice est stockée par ligne :
  - Les 2 éléments sont séparés par le nombre d'éléments dans une ligne multiplié par la taille de la tuile
  - Une fois dans la bonne tuile, il faut accéder à la bonne ligne, en n'oubliant pas que le stockage est fait par ligne (`ty*nb_col`)



Les indices dans le produit de matrices par tuile : B

- La matrice est stockée par ligne :
  - Les 2 éléments sont séparés par le nombre d'éléments dans une ligne multiplié par la taille de la tuile
  - Une fois dans la bonne tuile, il faut accéder à la bonne ligne, en n'oubliant pas que le stockage est fait par ligne (`ty*nb_col`)



Les indices dans le produit de matrices par tuile : B

- `col = blockIdx.x*blockDim.x+theadIdx.x`
- `ligne = blockIdx.y*blockDim.y+theadIdx.y`
- `TILE_WIDTH = blockDim.x`
- `indice = col + (nb_colB * TILE_WIDTH * tileId + ty * nb_col)`

Les indices dans le produit de matrices par tuile : C

- `col = blockIdx.x*blockDim.x+theadIdx.x`
- `ligne = blockIdx.y*blockDim.y+theadIdx.y`
- `indice = nbCol * ligne + col`