Introduction au calcul GP-GPU

Partie 2 : Découverte de CUDA

Laurent Risser

Institut de mathématiques de Toulouse
lrisser@math.univ-toulouse.fr

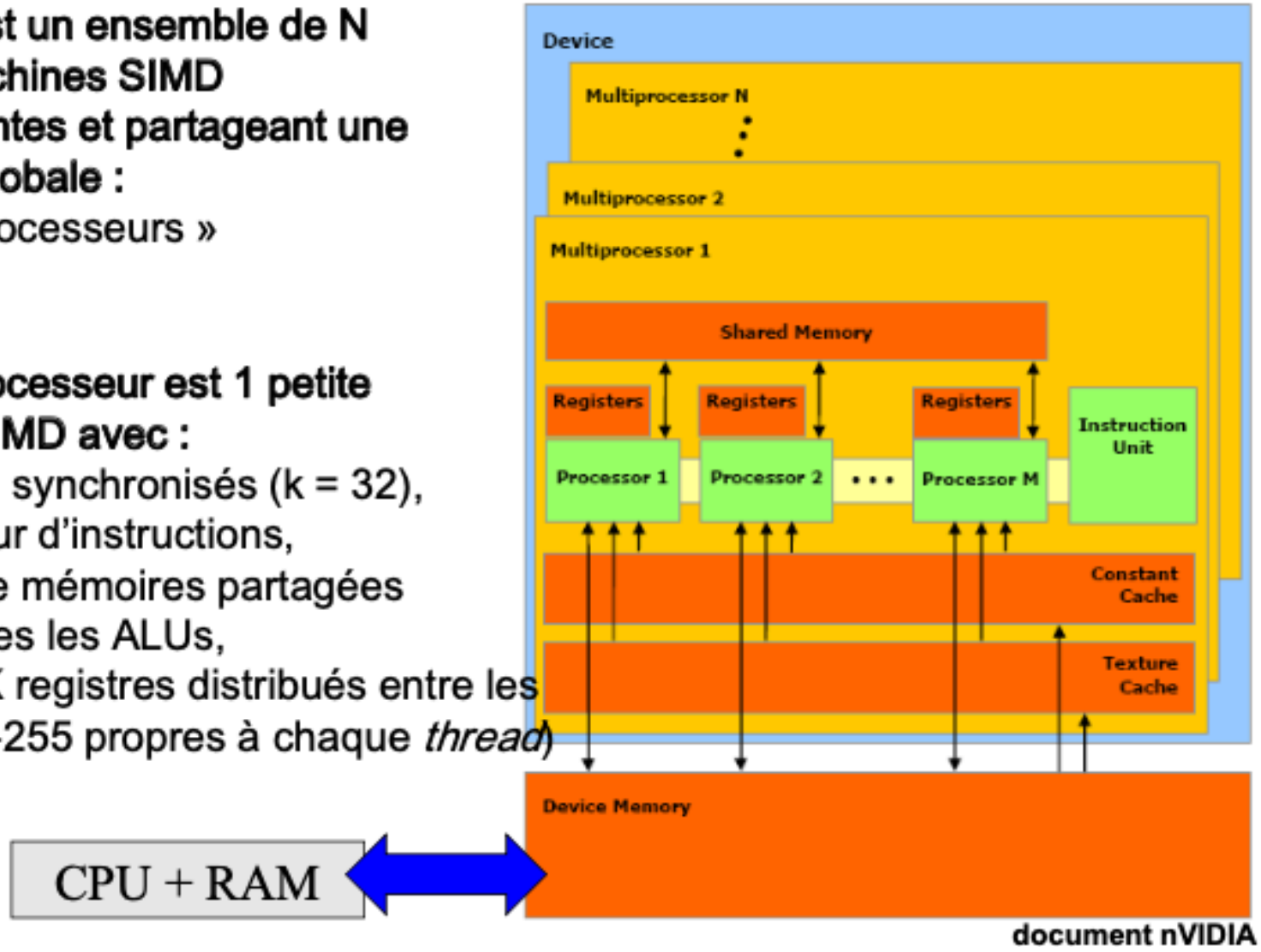
Présentation largement inspirée du cours de Stéphane Vialle

Plan de la présentation

- Architecture matérielle GPU et modèle CUDA
- Programmation CUDA avec PyCUDA

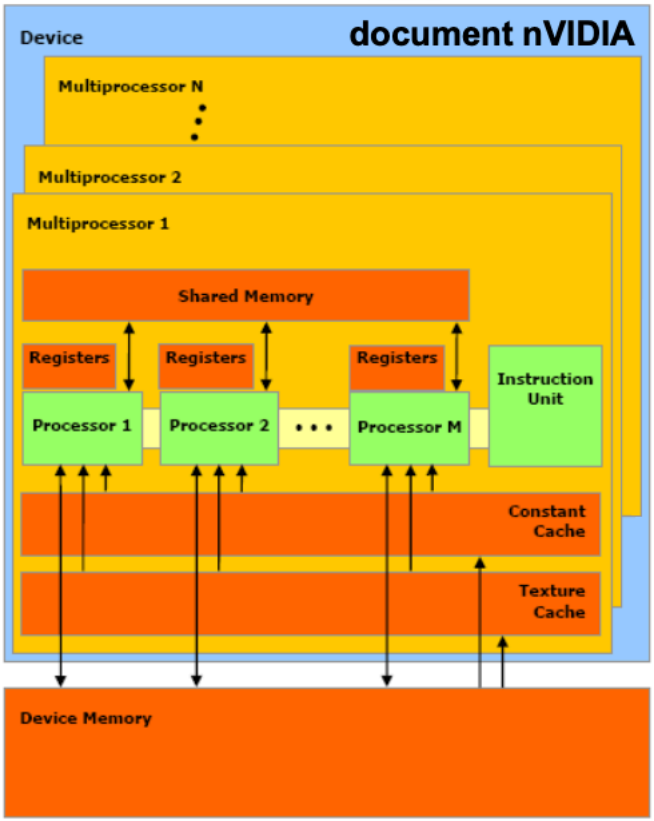
Un GPU est un ensemble de N petites machines SIMD indépendantes et partageant une mémoire globale :
N « multiprocesseurs »

Un multiprocesseur est 1 petite machine SIMD avec :
- k « ALU » synchronisés (k = 32),
- 1 décodeur d'instructions,
- 3 types de mémoires partagées entre toutes les ALUs,
- 32K-128K registres distribués entre les ALUs (63-255 propres à chaque *thread*)

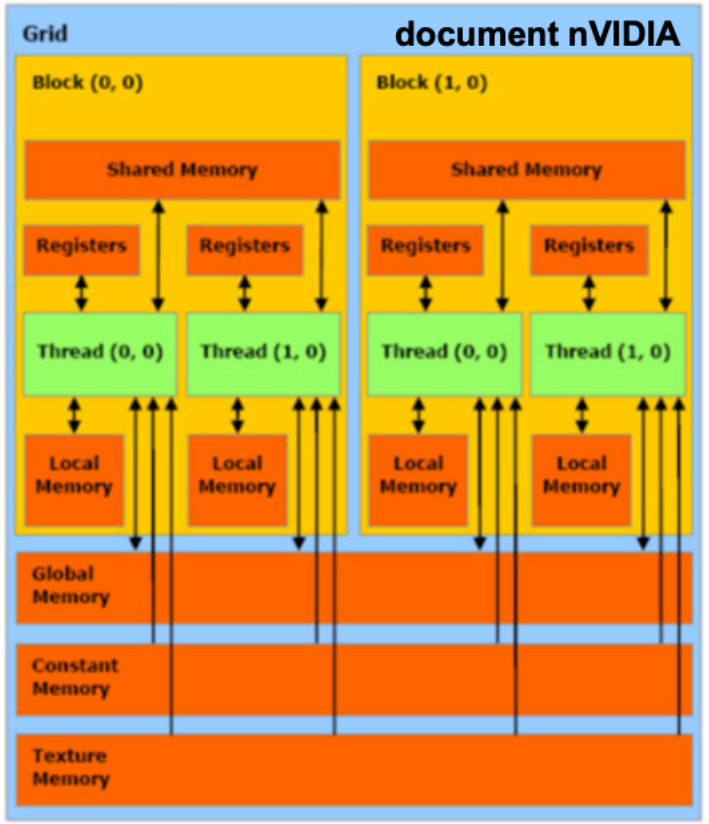


document nVIDIA

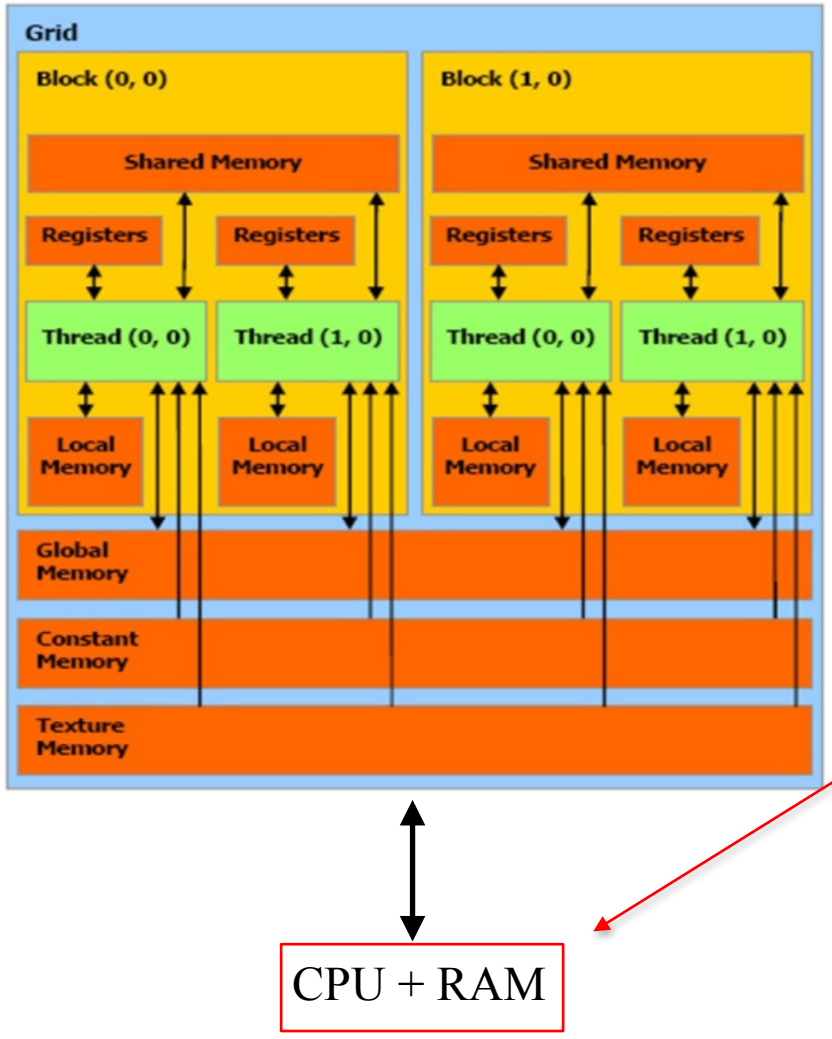
Modélisation de cette architecture dans CUDA



Machines virtuelles



Perception dans le langage



Définition et compilation d'une fonction CUDA

```
my_kernel="""
__global__ void addition(float *a, float *b, float *result)
{
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    result[index] = a[index] + b[index];
}
"""

mod = SourceModule(my_kernel)

addition = mod.get_function("addition")
```

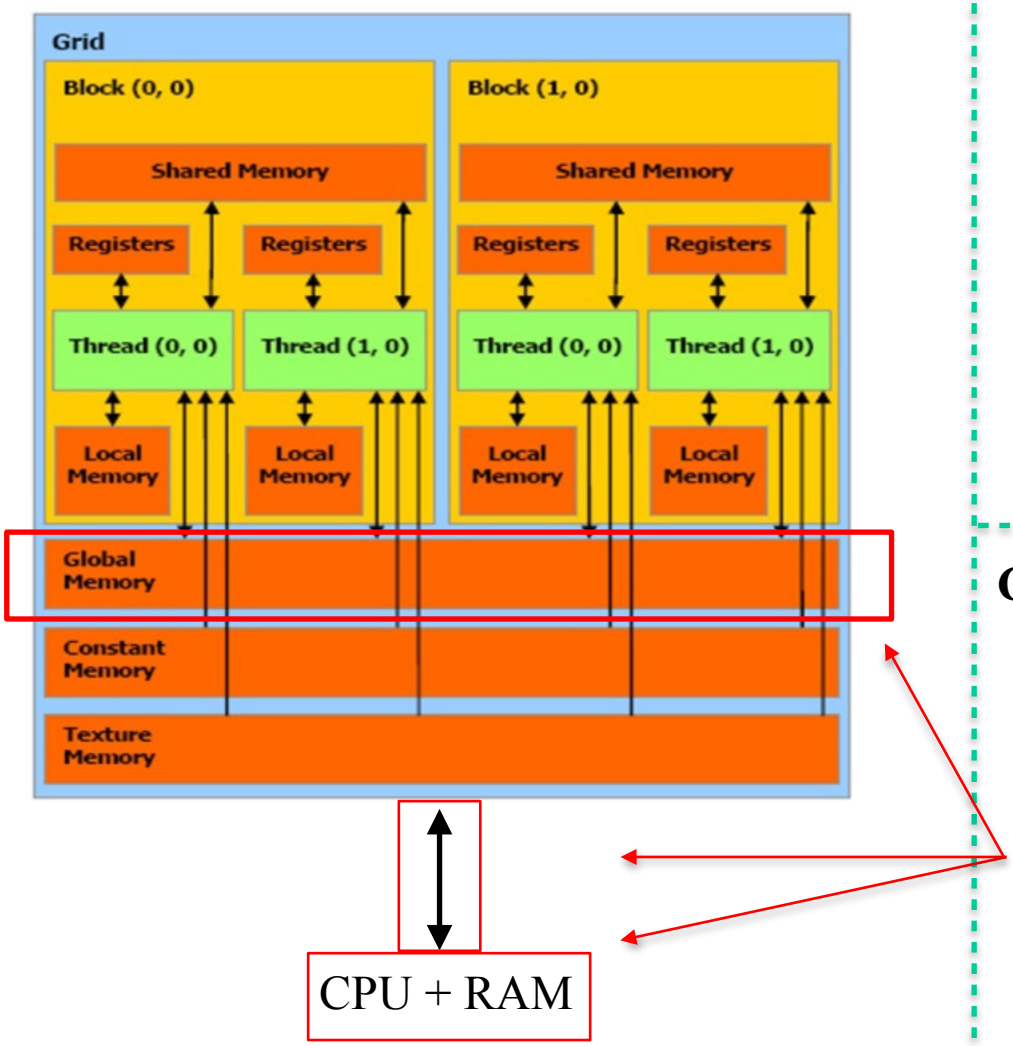
Code python appelant la fonction CUDA

```
a_cpu = np.random.randn(1024).astype(np.float32)
b_cpu = np.random.randn(1024).astype(np.float32)
```

```
# transfer host (CPU) memory to device (GPU) memory
a_gpu = gpuarray.to_gpu(a_cpu)
b_gpu = gpuarray.to_gpu(b_cpu)

# create empty gpu array for the result (C = A + B)
c_gpu = gpuarray.empty((1024), np.float32)
```

```
addition(a_gpu , b_gpu , c_gpu , block = (1024, 1, 1))
```

```
c_cpu = gpuarray.GPUArray.get(c_gpu)
```



Définition et compilation d'une fonction CUDA

```
my_kernel="""
__global__ void addition(float *a, float *b, float *result)
{
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    result[index] = a[index] + b[index];
}
"""

mod = SourceModule(my_kernel)

addition = mod.get_function("addition")
```

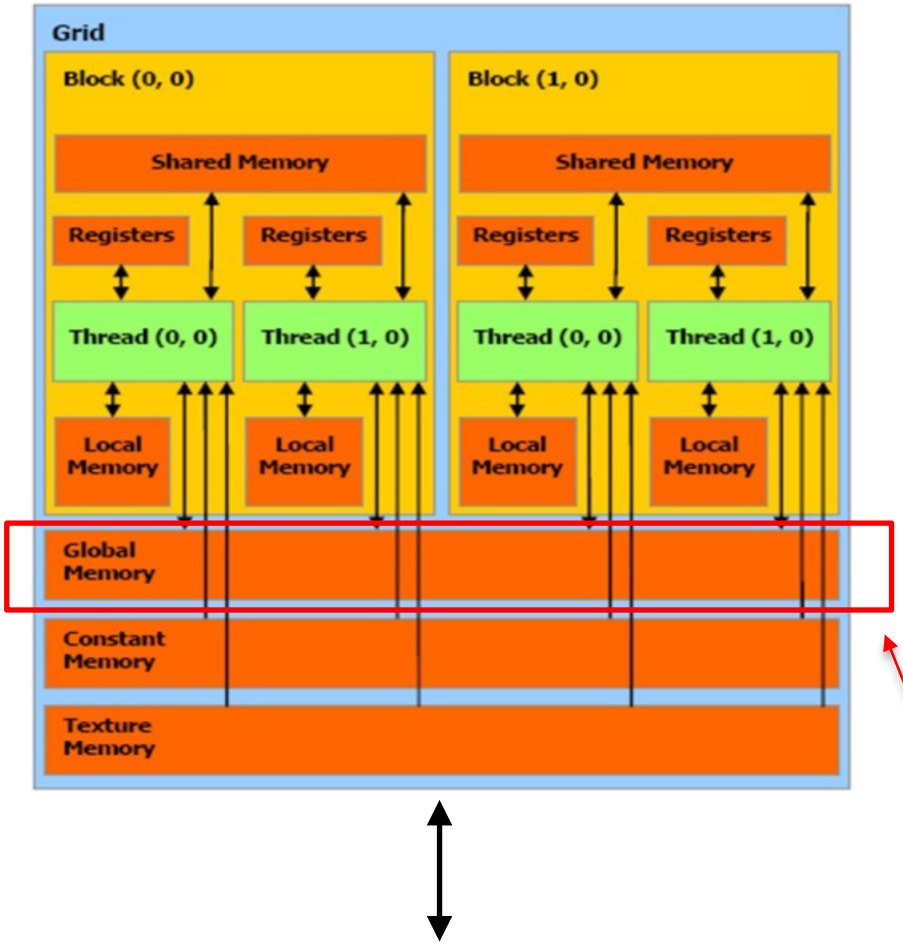
Code python appelant la fonction CUDA

```
a_cpu = np.random.randn(1024).astype(np.float32)
b_cpu = np.random.randn(1024).astype(np.float32)
```

```
# transfer host (CPU) memory to device (GPU) memory
a_gpu = gpuarray.to_gpu(a_cpu)
b_gpu = gpuarray.to_gpu(b_cpu)

# create empty gpu array for the result (C = A + B)
c_gpu = gpuarray.empty((1024), np.float32)
```

```
addition(a_gpu , b_gpu , c_gpu , block = (1024, 1, 1))
```

```
c_cpu = gpuarray.GPUArray.get(c_gpu)
```

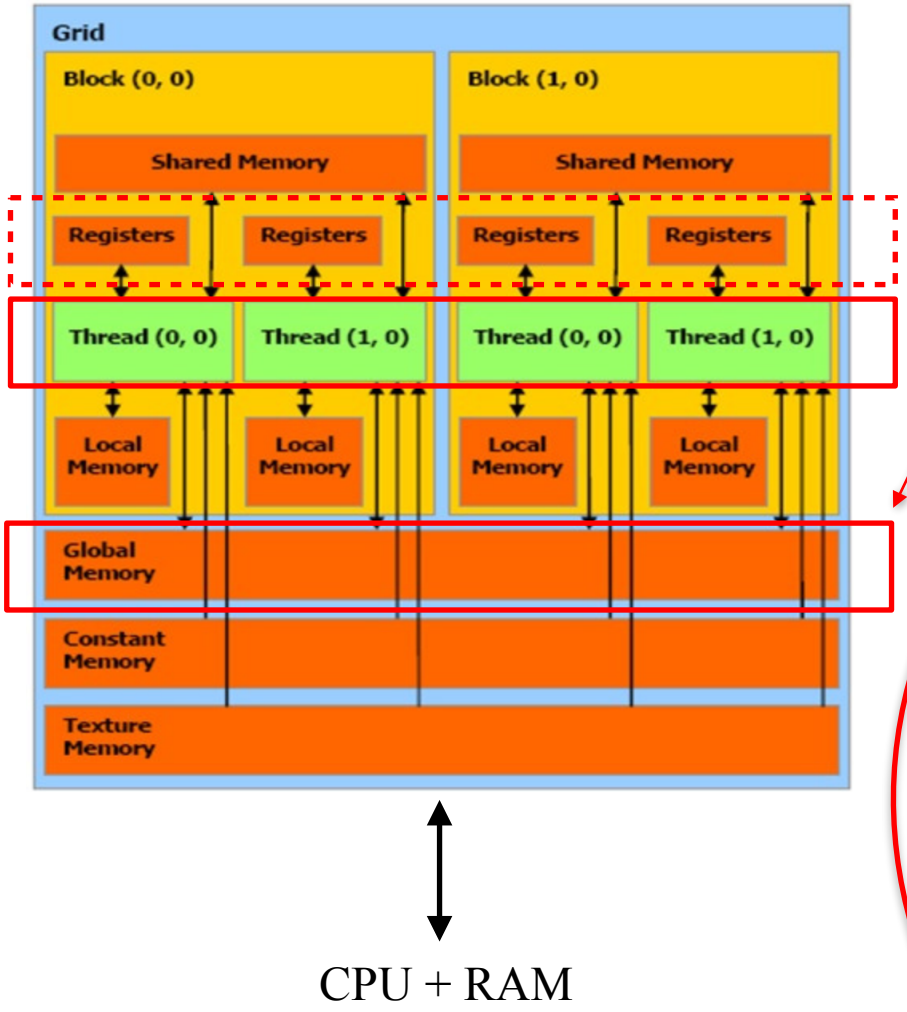


Définition et compilation d'une fonction CUDA

```
my_kernel="""
__global__ void addition(float *a, float *b, float *result)
{
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    result[index] = a[index] + b[index];
}
"""

mod = SourceModule(my_kernel)

addition = mod.get_function("addition")
```

Code python appelant la fonction CUDA

```
a_cpu = np.random.randn(1024).astype(np.float32)
b_cpu = np.random.randn(1024).astype(np.float32)

# transfer host (CPU) memory to device (GPU) memory
a_gpu = gpuarray.to_gpu(a_cpu)
b_gpu = gpuarray.to_gpu(b_cpu)

# create empty gpu array for the result (C = A + B)
c_gpu = gpuarray.empty((1024), np.float32)

addition(a_gpu , b_gpu , c_gpu , block = (1024, 1, 1))

c_cpu = gpuarray.GPUArray.get(c_gpu)
```



CPU + RAM

Définition et compilation d'une fonction CUDA

```
my_kernel="""
__global__ void addition(float *a, float *b, float *result)
{
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    result[index] = a[index] + b[index];
}
"""

mod = SourceModule(my_kernel)


addition = mod.get_function("addition")
```

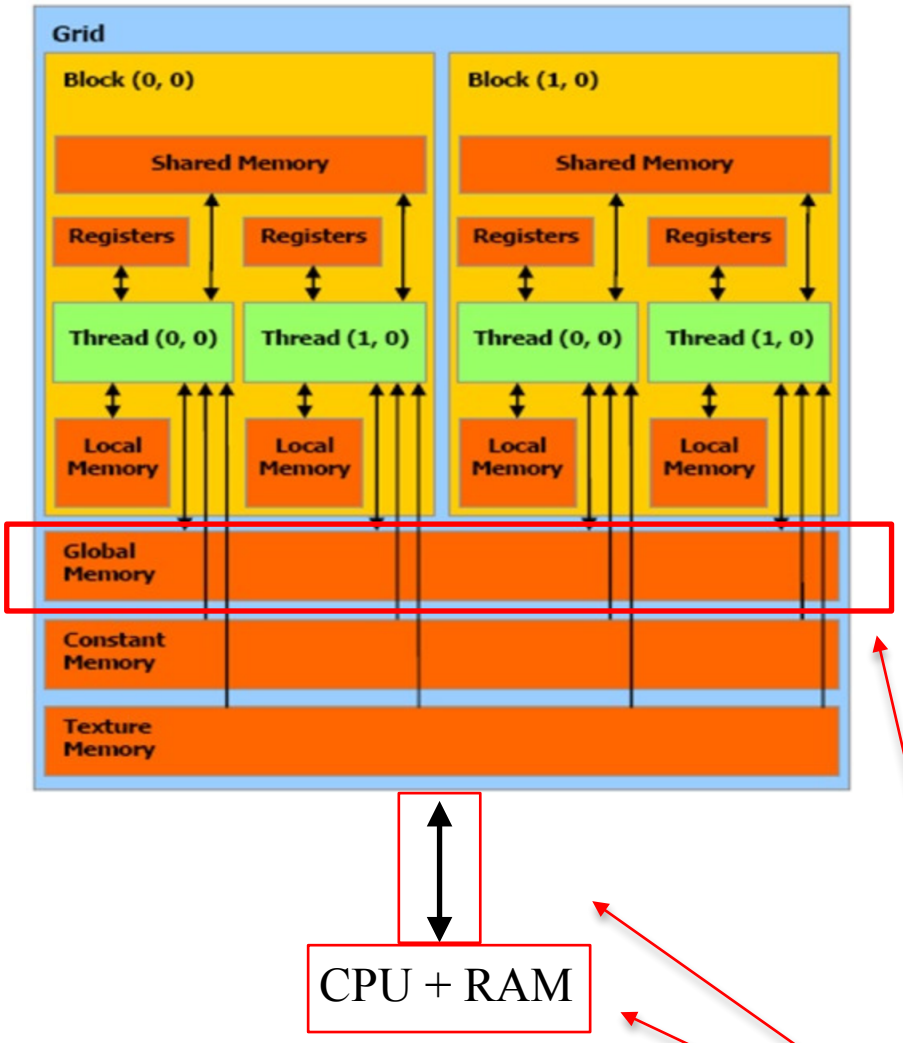
Code python appelant la fonction CUDA

```
a_cpu = np.random.randn(1024).astype(np.float32)
b_cpu = np.random.randn(1024).astype(np.float32)
```

```
# transfer host (CPU) memory to device (GPU) memory
a_gpu = gpuarray.to_gpu(a_cpu)
b_gpu = gpuarray.to_gpu(b_cpu)

# create empty gpu array for the result (C = A + B)
c_gpu = gpuarray.empty((1024), np.float32)
```

```
addition(a_gpu , b_gpu , c_gpu , block = (1024, 1, 1))
```

```
c_cpu = gpuarray.GPUArray.get(c_gpu)
```

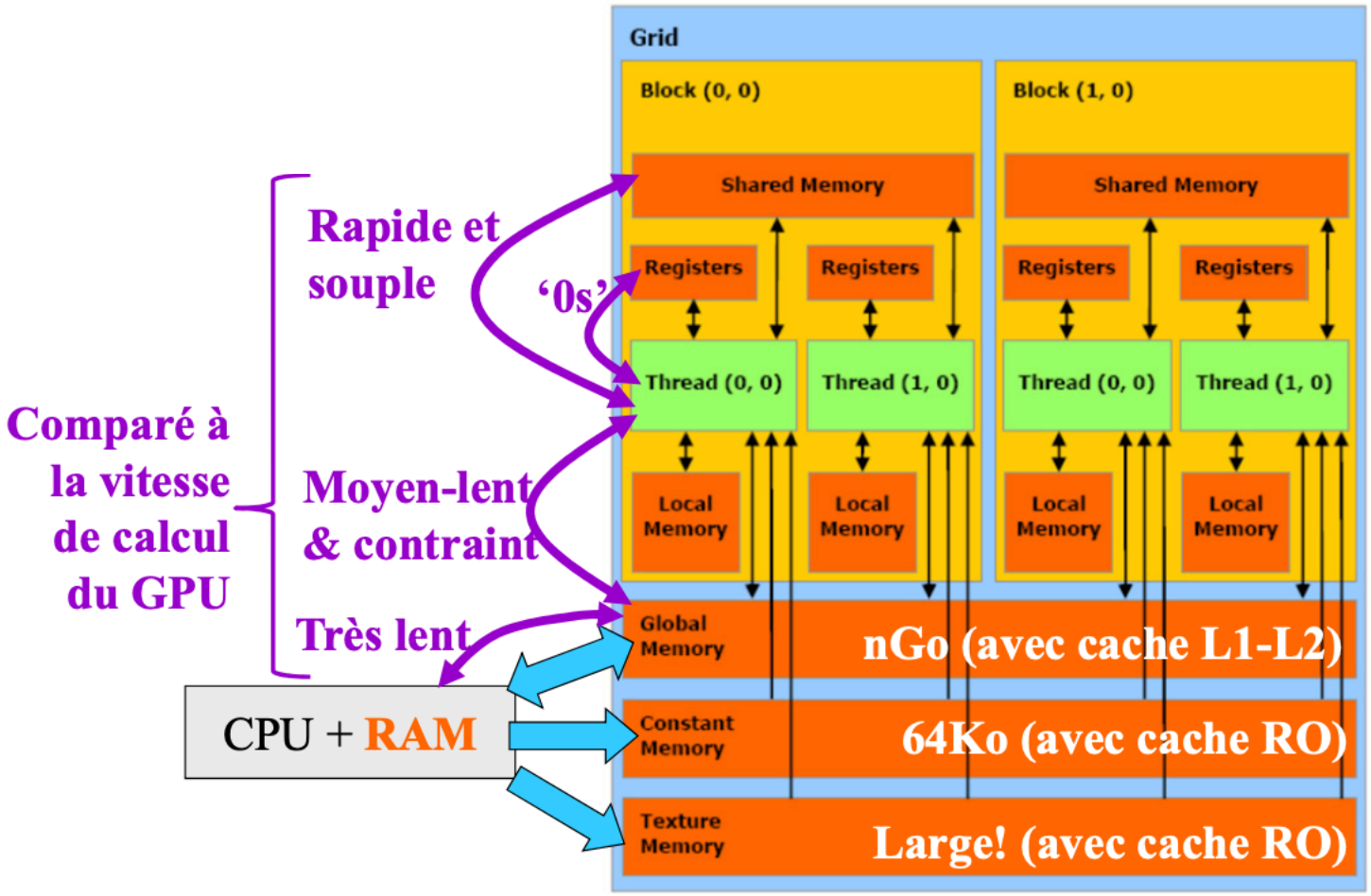


Définition et compilation d'une fonction CUDA

```
my_kernel="""
__global__ void addition(float *a, float *b, float *result)
{
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    result[index] = a[index] + b[index];
}
"""

mod = SourceModule(my_kernel)

addition = mod.get_function("addition")
```

Code python appelant la fonction CUDA

```
a_cpu = np.random.randn(1024).astype(np.float32)
b_cpu = np.random.randn(1024).astype(np.float32)

# transfer host (CPU) memory to device (GPU) memory
a_gpu = gpuarray.to_gpu(a_cpu)
b_gpu = gpuarray.to_gpu(b_cpu)

# create empty gpu array for the result (C = A + B)
c_gpu = gpuarray.empty((1024), np.float32)

addition(a_gpu , b_gpu , c_gpu , block = (1024, 1, 1))

c_cpu = gpuarray.GPUArray.get(c_gpu)
```



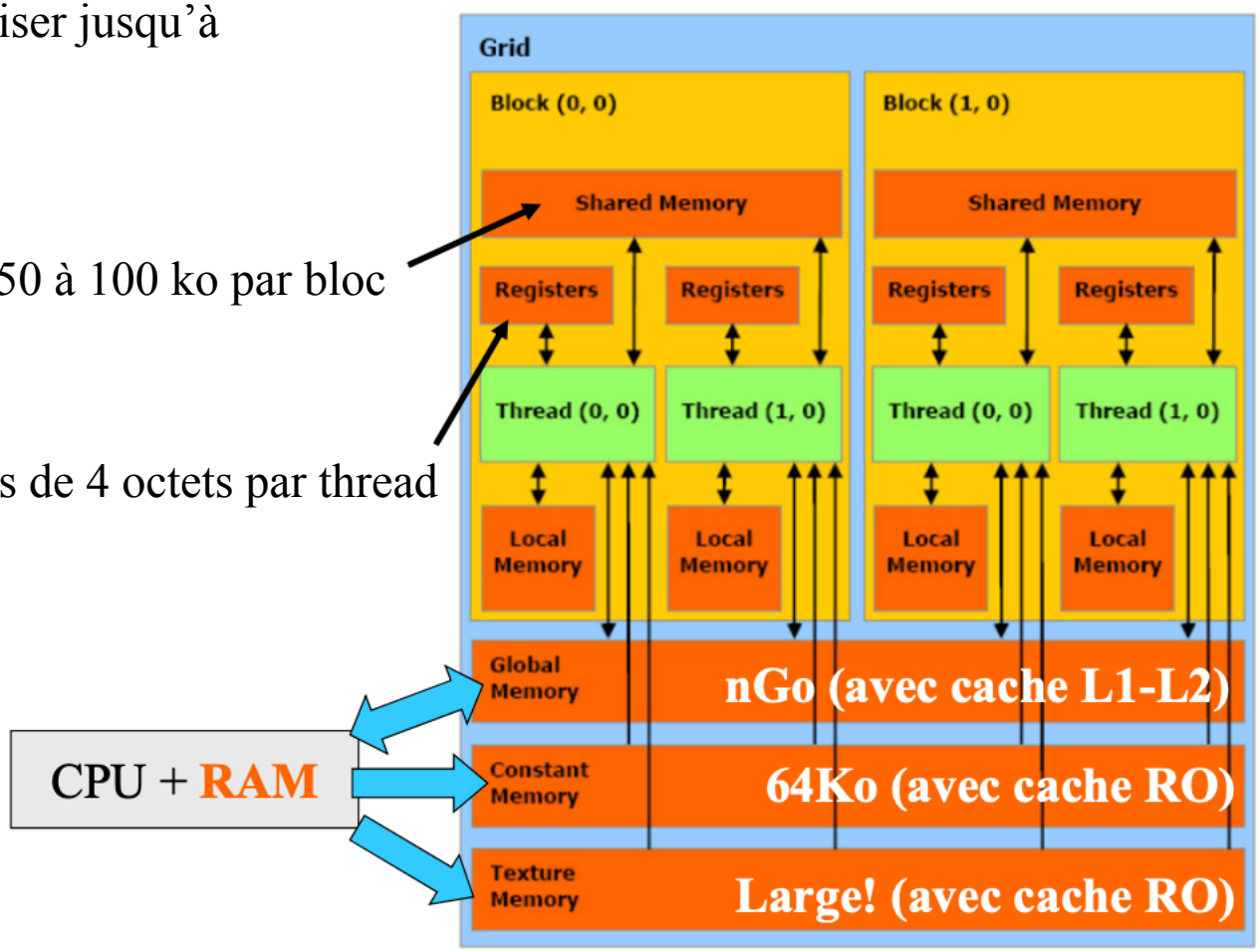
→ Intérêt de la *mémoire shared* et des *registres* si un *thread* utilise plusieurs fois la même case mémoire !

Hiérarchie de tailles mémoires et de temps d'accès très variables

On peut typiquement utiliser jusqu'à
- 1024 threads par bloc
- 65535^3 blocs

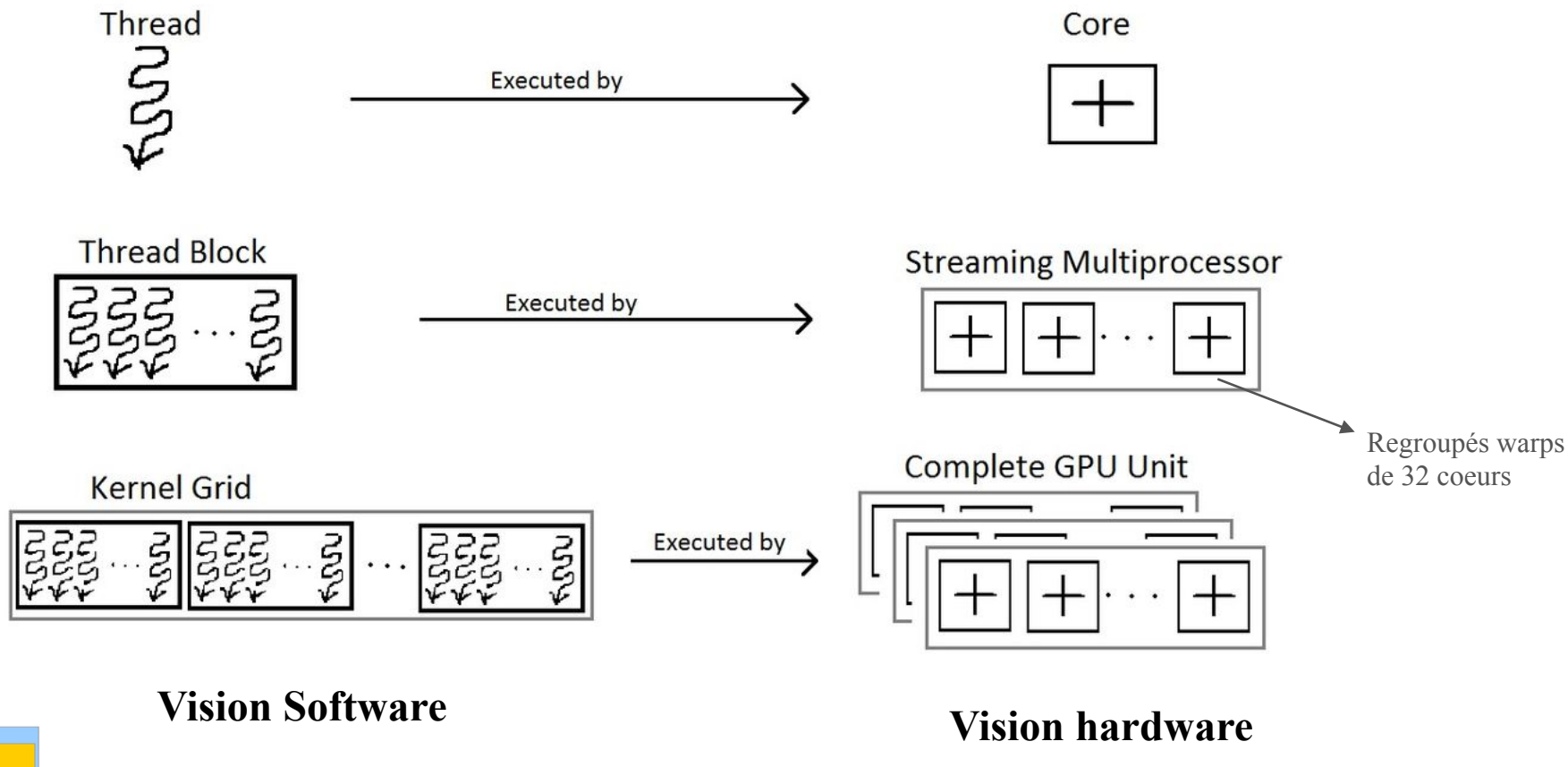
≈ 50 à 100 ko par bloc

≈ 255 registres de 4 octets par thread



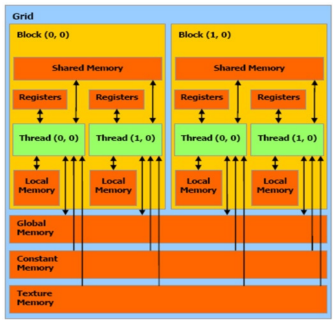
→ Fortes contraintes sur les tailles mémoire et la visibilité des zones mémoire pour chaque thread !

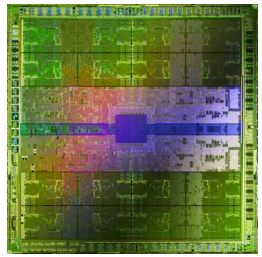
Trois niveaux de parallélisation : Thread / Block / Grid



Vision Software

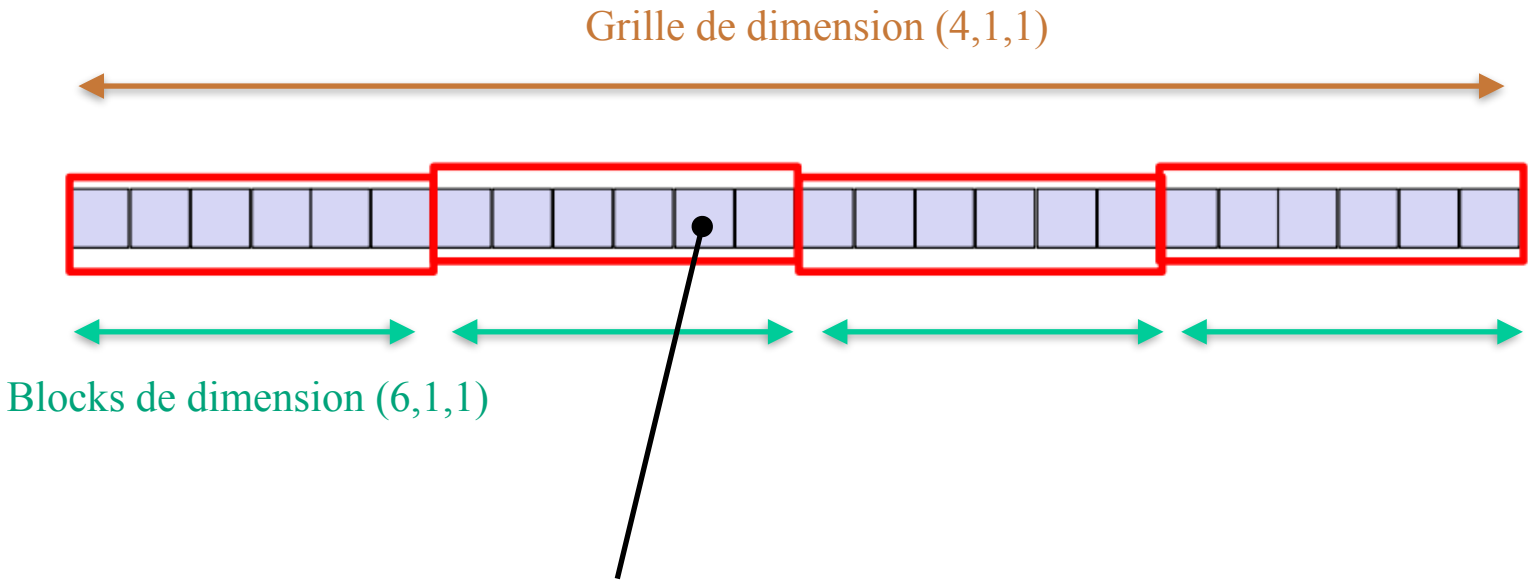
Vision hardware





Source : https://en.wikipedia.org/wiki/Thread_block_(CUDA_programming)#Streaming_multiprocessors

Accès à une case mémoire dans un vecteur



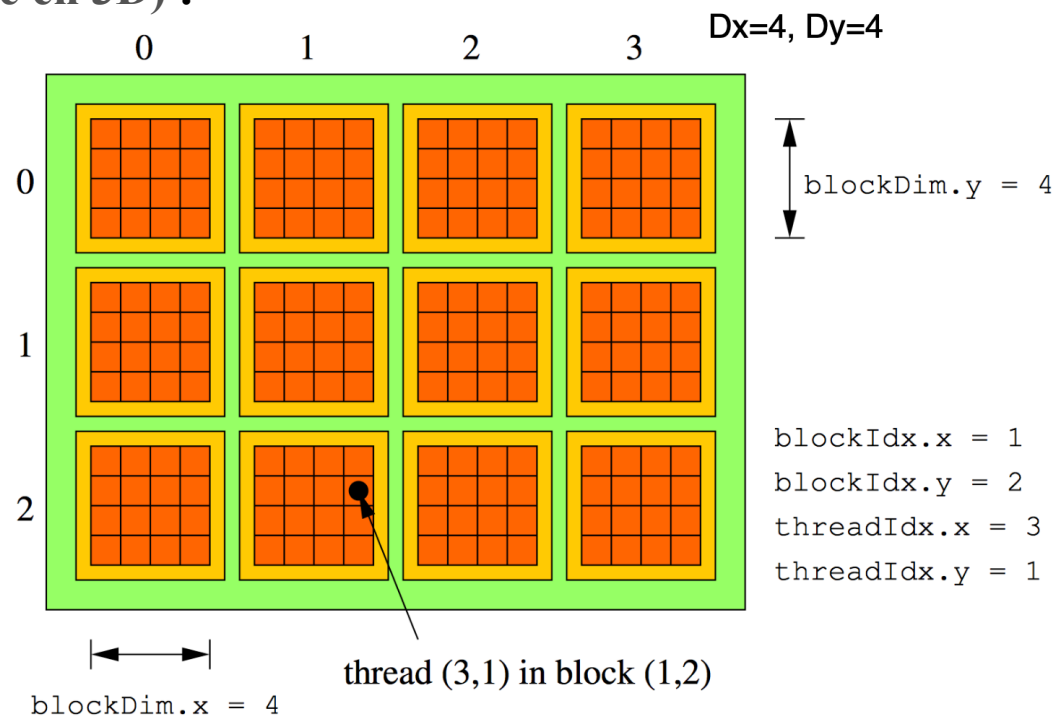
Indice d'une case dans un kernel = $threadId.x + \left(blockId.x \times blockDim.x\right)$

```
my_kernel="""
__global__ void addition(float *a, float *b, float *result)
{
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    result[index] = a[index] + b[index];
}
"""
```

Remarque : Définition de la taille des blocs et de la grille lors de l'appel de la fonction

```
addition(a_gpu , b_gpu , c_gpu , block = (6,1,1) , grid = (4,1,1))
```

En 2D (même principe en 3D) :



threadIdx:

$$x = \texttt{blockIdx.x} \times \texttt{blockDim.x} + \texttt{threadIdx.x} = 1 \times 4 + 3 = 7$$

$$y = \texttt{blockIdx.y} \times \texttt{blockDim.y} + \texttt{threadIdx.y} = 2 \times 4 + 1 = 9$$

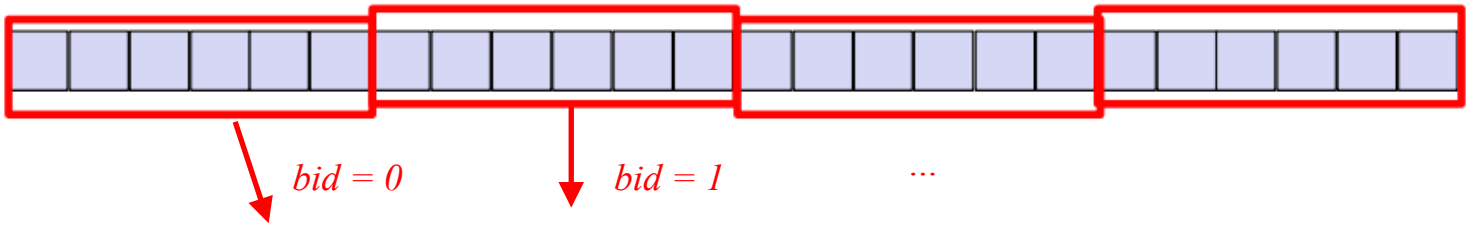
$$ThreadID = x + y \times (blockDim.x \times gridDim.x) = 7 + 9 \times 16$$

Remarque 1 : Définition de la taille des blocs et de la grille lors de l'appel de la fonction

```
addition(a_gpu , b_gpu , c_gpu , block = (4,4,1) , grid = (4,3,1))
```

Remarque 2 : Blocs idéalement de taille d'un multiple de 32 sur chaque dimension (taille des *warps*)

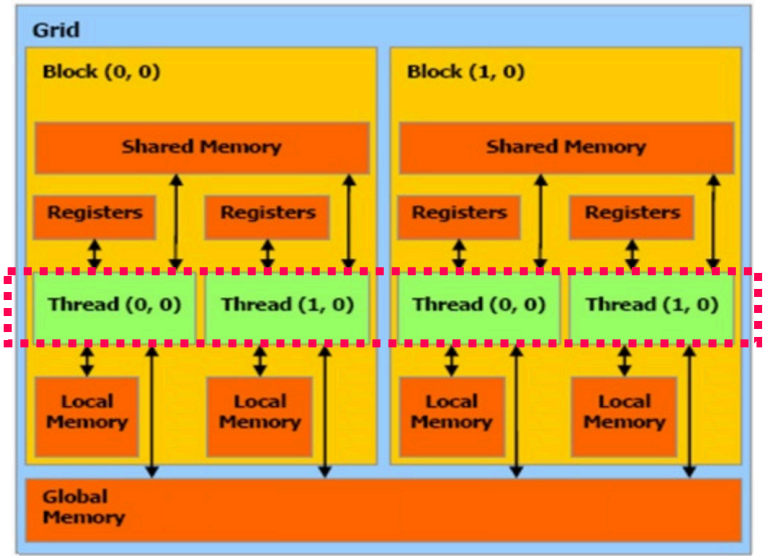
Copies dans la mémoire __shared__ au sein d'un bloc



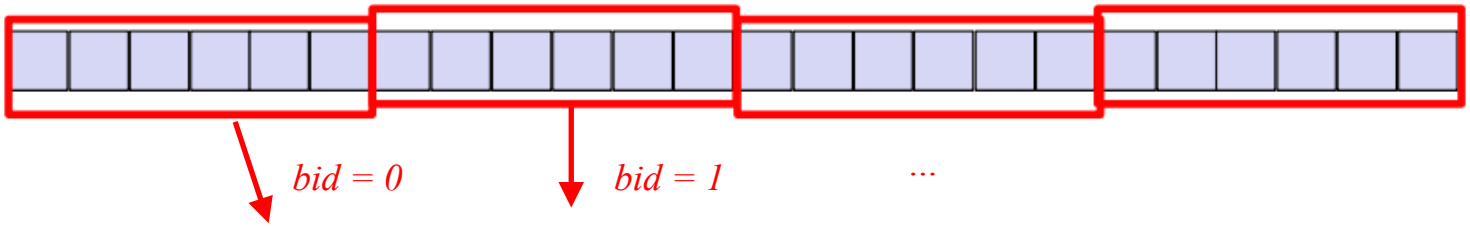
result[bid] est la somme des valeurs au carré dans le bloc *bid*

```
__global__ void sumOfSquares(int* num, int *result)
{
    __shared__ int shared[blockDim.x];
    int i= threadIdx.x + blockIdx.x * blockDim.x;

    int tid = threadIdx.x;
    int bid = blockIdx.x;

    shared[tid] = num[i]*num[i];

    //Synchronisation de tous les threads
    __syncthreads();

    if (tid == 0) {
        for (int i = 1; i < blockDim.x; i++) {
            shared[0] += shared[i];
        }
        result[bid] = shared[0];
    }
}
```

Code pour un *thread*

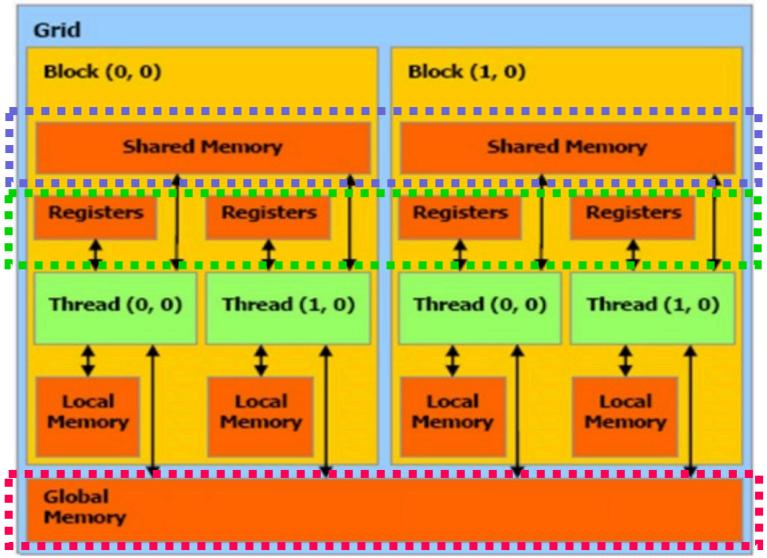


Copies dans la mémoire __shared__ au sein d'un bloc

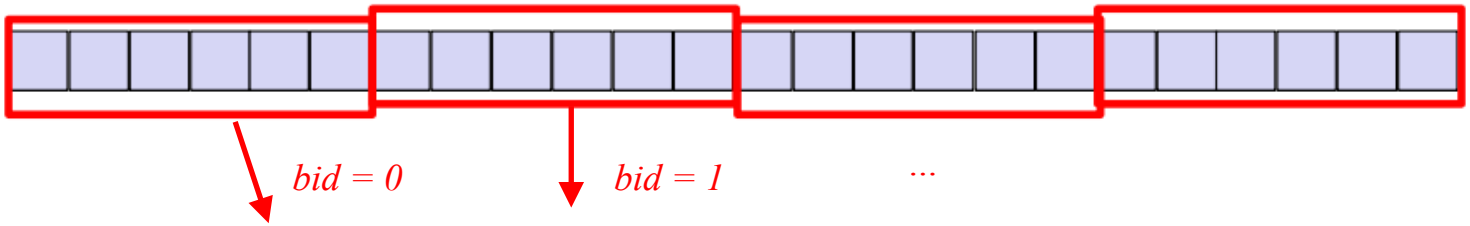


result[bid] est la somme des valeurs au carré dans le bloc *bid*

```
__global__ void sumOfSquares(int* num, int *result)
{
    __shared__ int shared[blockDim.x];
    int i= threadIdx.x + blockIdx.x * blockDim.x;

    int tid = threadIdx.x;
    int bid = blockIdx.x;

    shared[tid] = num[i]*num[i];

    //Synchronisation de tous les threads
    __syncthreads();

    if (tid == 0) {
        for (int i = 1; i < blockDim.x; i++) {
            shared[0] += shared[i];
        }
        result[bid] = shared[0];
    }
}
```



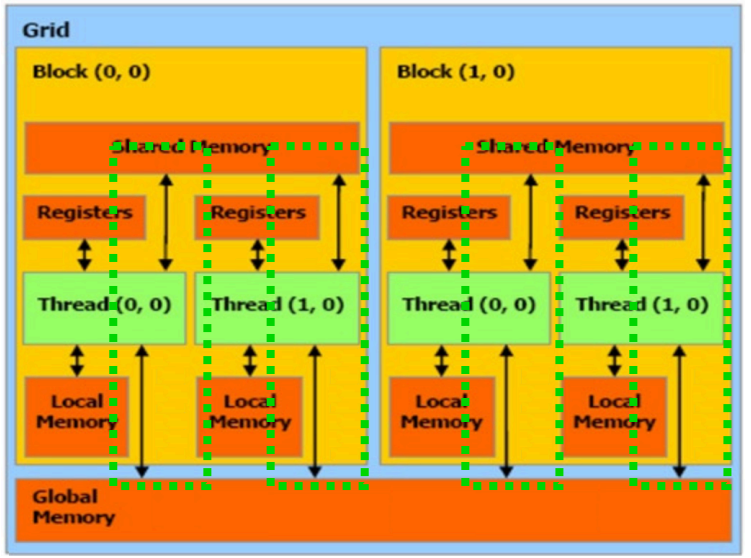
Copies dans la mémoire __shared__ au sein d'un bloc



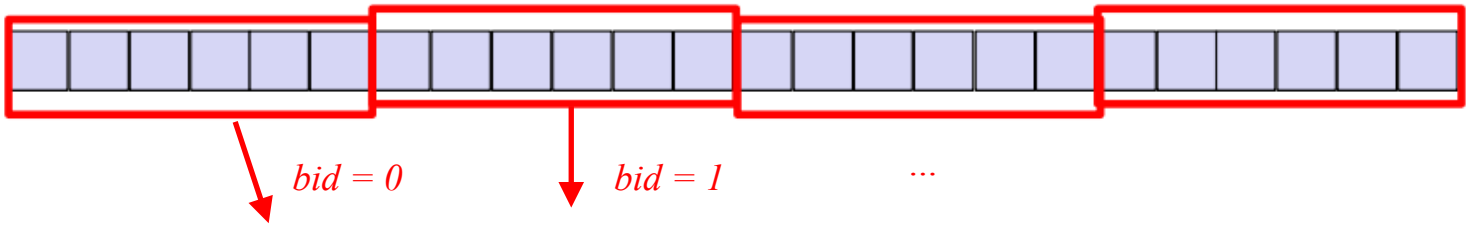
result[bid] est la somme des valeurs au carré dans le bloc *bid*

```
__global__ void sumOfSquares(int* num, int *result)
{
    __shared__ int shared[blockDim.x];
    int i= threadIdx.x + blockIdx.x * blockDim.x;

    int tid = threadIdx.x;
    int bid = blockIdx.x;

    shared[tid] = num[i]*num[i];

    //Synchronisation de tous les threads
    __syncthreads();

    if (tid == 0) {
        for (int i = 1; i < blockDim.x; i++) {
            shared[0] += shared[i];
        }
        result[bid] = shared[0];
    }
}
```

Copie global vers shared



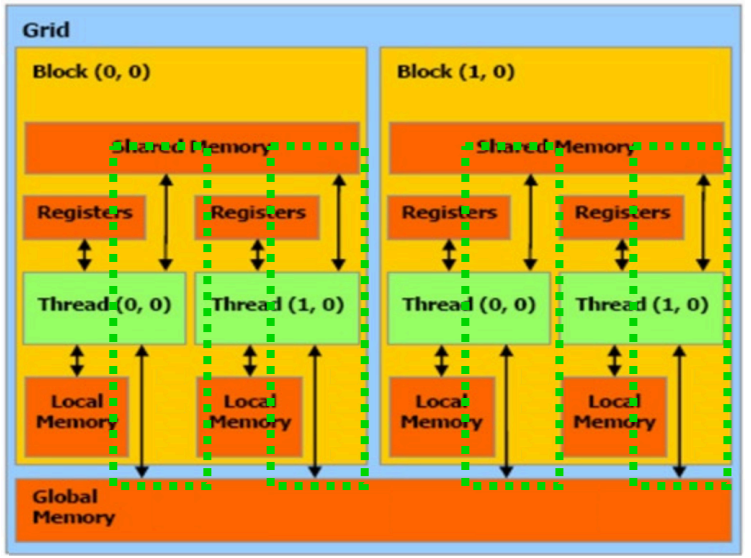
Copies dans la mémoire __shared__ au sein d'un bloc



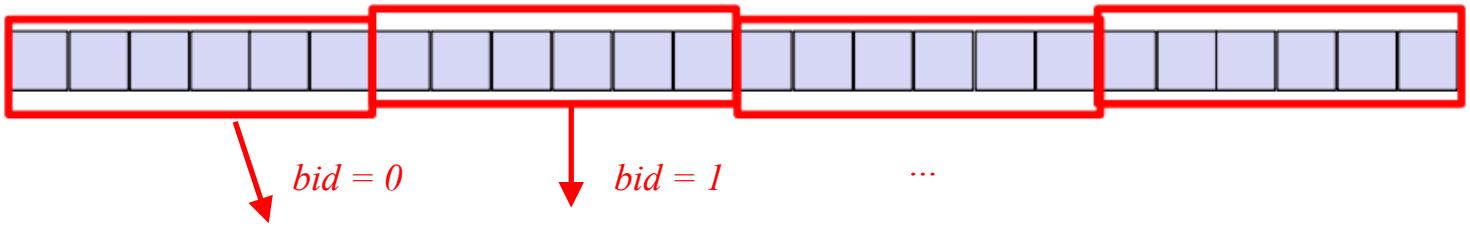
result[bid] est la somme des valeurs au carré dans le bloc *bid*

```
__global__ void sumOfSquares(int* num, int *result)
{
    __shared__ int shared[blockDim.x];
    int i= threadIdx.x + blockIdx.x * blockDim.x;

    int tid = threadIdx.x;
    int bid = blockIdx.x;

    shared[tid] = num[i]*num[i];

    //Synchronisation de tous les threads
    __syncthreads();

    if (tid == 0) {
        for (int i = 1; i < blockDim.x; i++) {
            shared[0] += shared[i];
        }
        result[bid] = shared[0];
    }
}
```

Synchronisation des threads dans chaque bloc



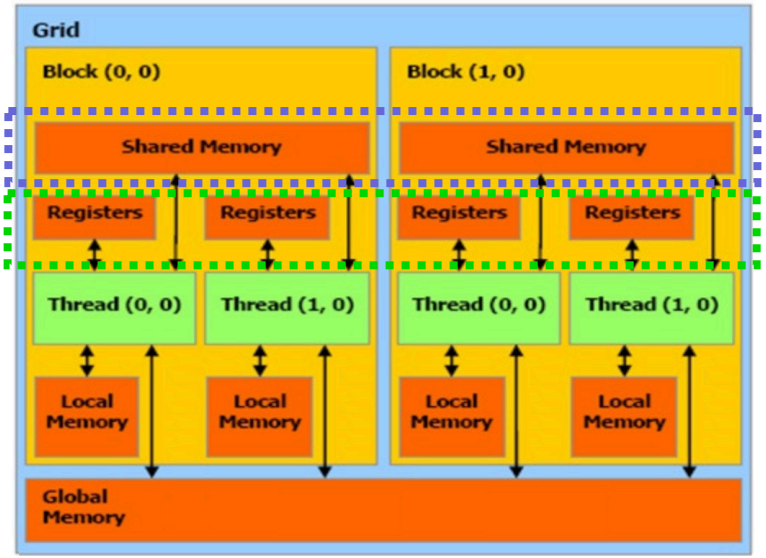
Copies dans la mémoire __shared__ au sein d'un bloc



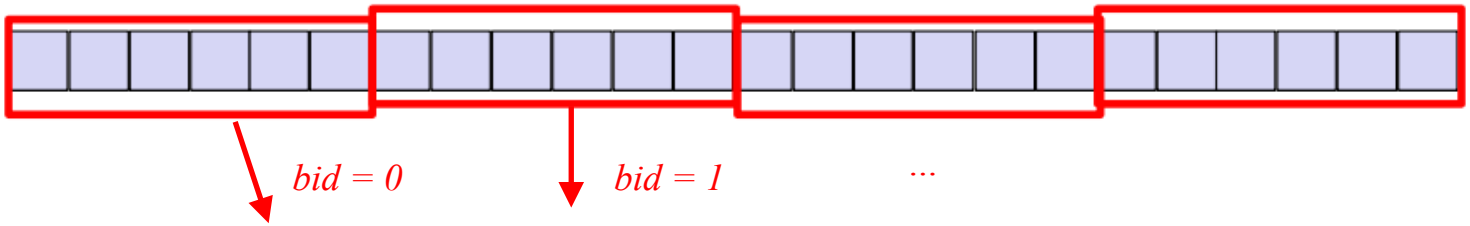
result[bid] est la somme des valeurs au carré dans le bloc *bid*

```
__global__ void sumOfSquares(int* num, int *result)
{
    __shared__ int shared[blockDim.x];
    int i= threadIdx.x + blockIdx.x * blockDim.x;

    int tid = threadIdx.x;
    int bid = blockIdx.x;

    shared[tid] = num[i]*num[i];

    //Synchronisation de tous les threads
    __syncthreads();

    if (tid == 0) {
        for (int i = 1; i < blockDim.x; i++) {
            shared[0] += shared[i];
        }
        result[bid] = shared[0];
    }
}
```

Calculs faisant appel aux mémoires *shared* et *registers*



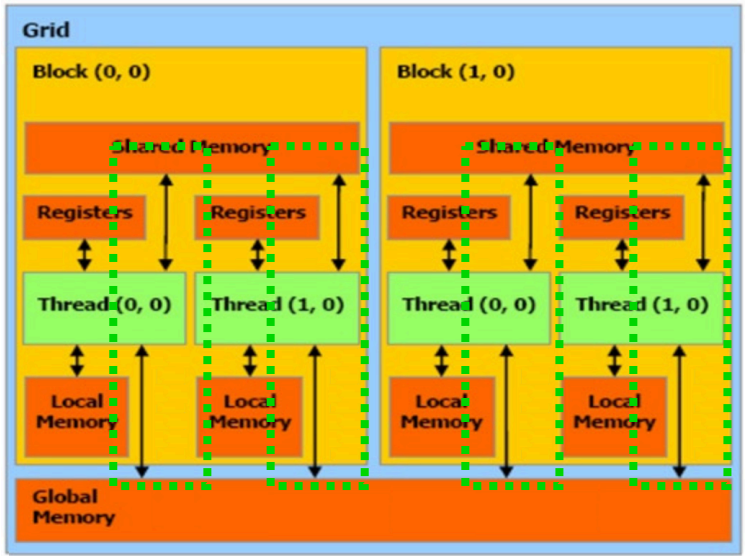
Copies dans la mémoire __shared__ au sein d'un bloc



result[bid] est la somme des valeurs au carré dans le bloc *bid*

```
__global__ void sumOfSquares(int* num, int *result)
{
    __shared__ int shared[blockDim.x];
    int i= threadIdx.x + blockIdx.x * blockDim.x;

    int tid = threadIdx.x;
    int bid = blockIdx.x;

    shared[tid] = num[i]*num[i];

    //Synchronisation de tous les threads
    __syncthreads();

    if (tid == 0) {
        for (int i = 1; i < blockDim.x; i++) {
            shared[0] += shared[i];
        }
        result[bid] = shared[0];
    }
}
```

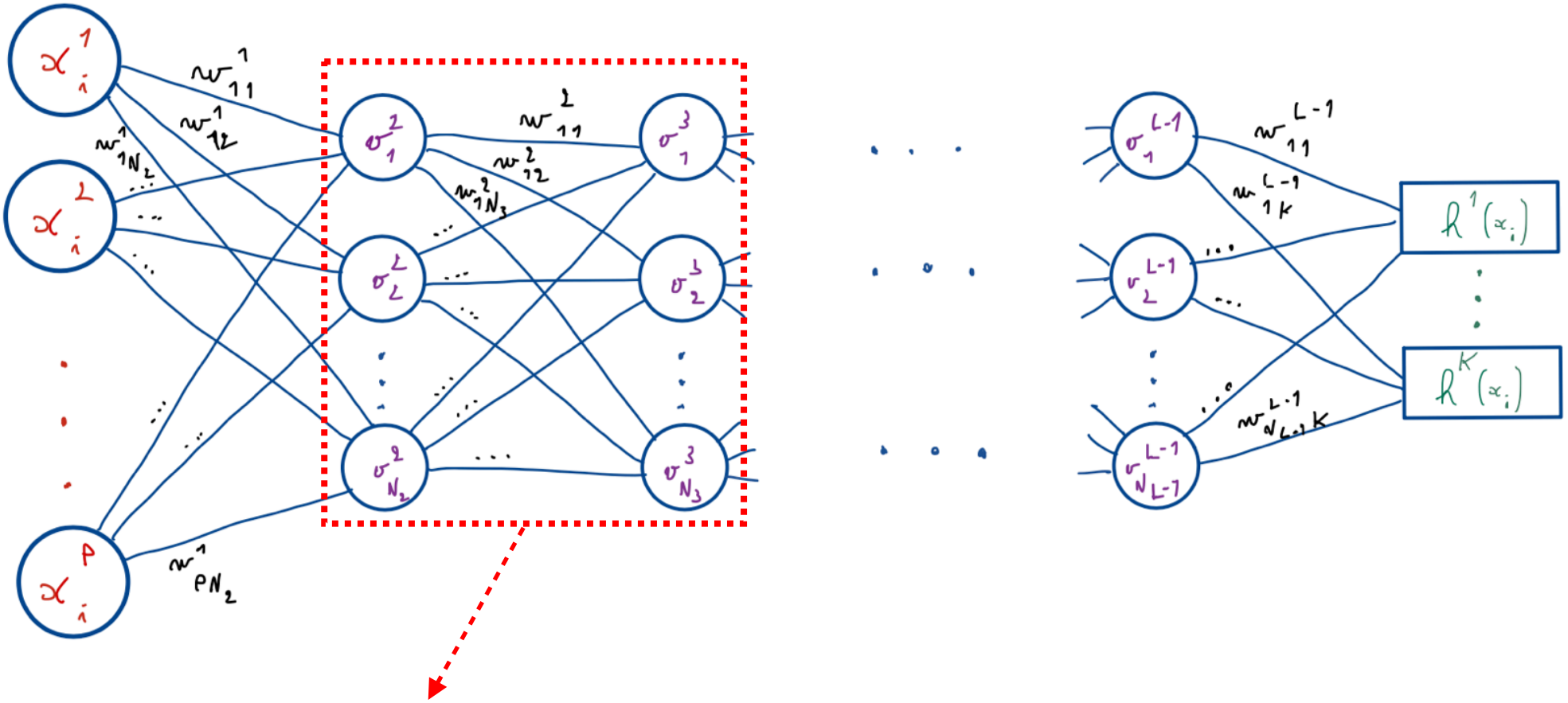
← Copie shared vers global



Plus d'informations sur les différentes contraintes : https://en.wikipedia.org/wiki/CUDA

	2006				2012					2015			2016			2017			2020	
Technical specifications	**Compute capability (version)**																			
	1.0	**1.1**	**1.2**	**1.3**	**2.x**	**3.0**	**3.2**	**3.5**	**3.7**	**5.0**	**5.2**	**5.3**	**6.0**	**6.1**	**6.2**	**7.0**	**7.2**	**7.5**	**8.0**	**8.6**
Maximum number of resident grids per device (concurrent kernel execution)	t.b.d.				16		4		32			16	128	32	16	128	16	128		
Maximum dimensionality of grid of thread blocks	2				3															
Maximum x-dimension of a grid of thread blocks	65535				$2^{31}-1$															
Maximum y-, or z-dimension of a grid of thread blocks	65535																			
Maximum dimensionality of thread block	3																			
Maximum x- or y-dimension of a block	512				1024															
Maximum z-dimension of a block	64																			
Maximum number of threads per block	512				1024															
Warp size	32																			
Maximum number of resident blocks per multiprocessor	8					16				32								16	32	16
Maximum number of resident warps per multiprocessor	24		32		48	64												32	64	48
Maximum number of resident threads per multiprocessor	768		1024		1536	2048												1024	2048	1536
Number of 32-bit registers per multiprocessor	8 K		16 K		32 K	64 K			128 K	64 K										
Maximum number of 32-bit registers per thread block	N/A				32 K	64 K	32 K		64 K			32 K	64 K		32 K	64 K				
Maximum number of 32-bit registers per thread	124				63	255														
Maximum amount of shared memory per multiprocessor	16 KB				48 KB				112 KB	64 KB	96 KB	64 KB		96 KB	64 KB	96 KB (of 128)		64 KB (of 96)	164 KB (of 192)	100 KB (of 128)
Maximum amount of shared memory per thread block	48 KB															96 KB	48 KB	64 KB	163 KB	99 KB
Number of shared memory banks	16				32															
Amount of local memory per thread	16 KB				512 KB															
Constant memory size	64 KB																			
Cache working set per multiprocessor for constant memory	8 KB												4 KB	8 KB						

Passage forward d'une couche dense d'un réseau de neurones :



$$o_j^3 = \sum_{i=1}^{N_2} o_i^2 w_{i,j}^2, \forall j \in \{1,\dots,N_3\}$$

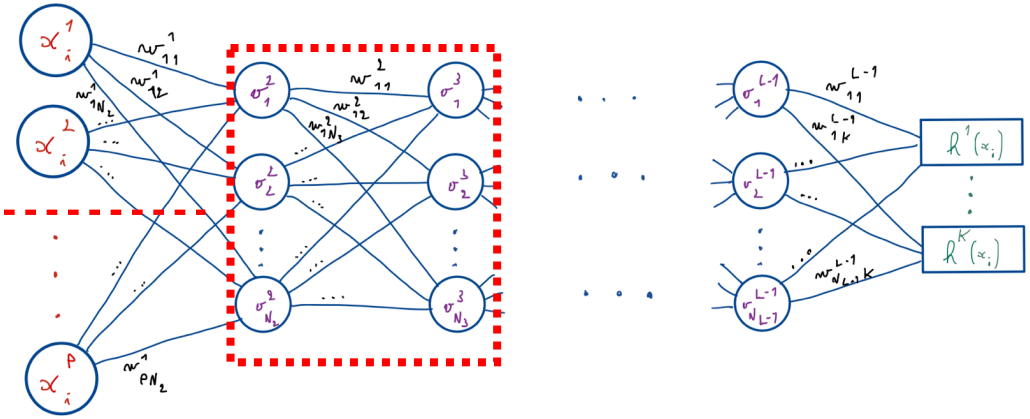
… pour chaque observation b
d'un mini-batch $b = 1,\dots,B$

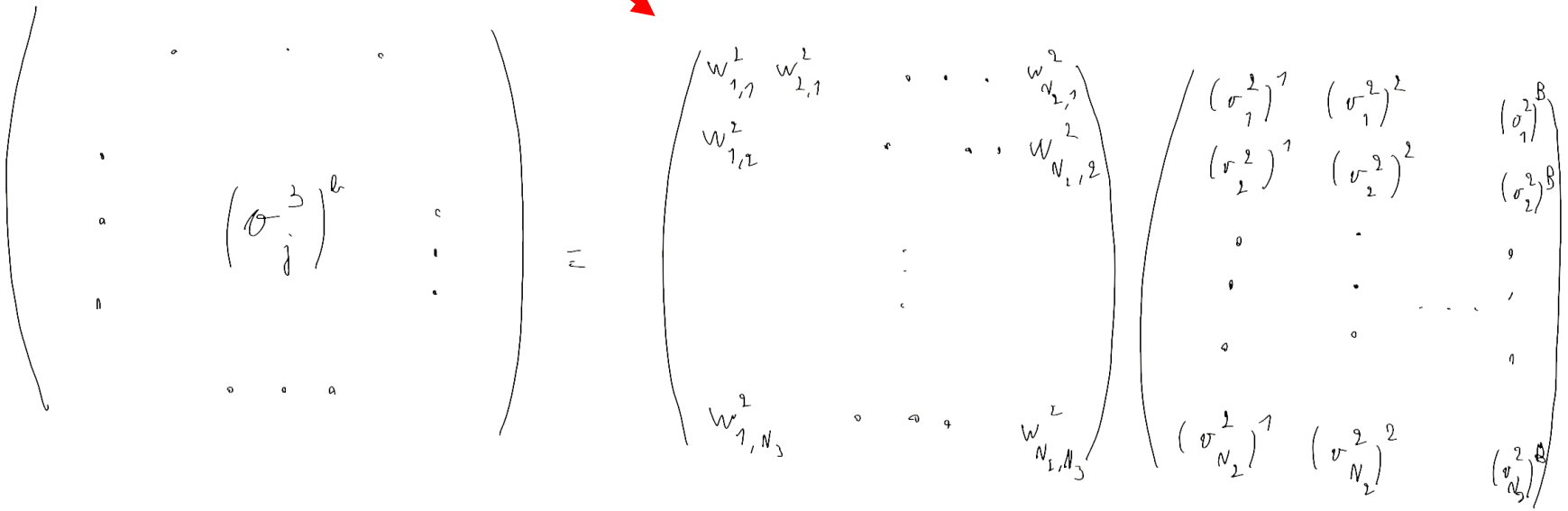
Passage forward d'une couche dense d'un réseau de neurones :

$$\left(o_j^3\right)^b = \sum_{i=1}^{N_2} \left(o_i^2\right)^b w_{i,j}^2$$

$$\forall j \in \{1,\ldots,N_3\}$$

$$\forall b \in \{1,\ldots,B\}$$





→ Complexité algorithmique = $\mathcal{O}(N_2 \times N_3 \times B)$ … mais se parallélise très bien sur un GPU !

TP1