UF I5AISE51
Dimensionnement et évaluation des architectures
Introduction à la programmation massivement parallèle

—

Partie 4
Mémoire partagée et synchronisation

—

P.-E. Hladik

INSA Toulouse

6 janvier 2021

Plan

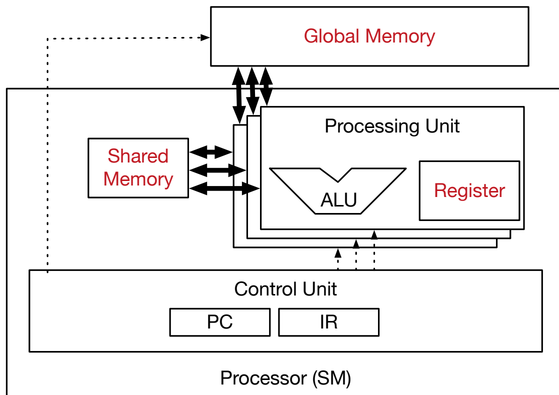
1. Introduction à la programmation parallèle
2. Allocation des données et exécution d'un kernel
3. Kernel à plusieurs dimensions
4. Mémoire partagée et synchronisation

Comment bien utiliser la mémoire du GPU

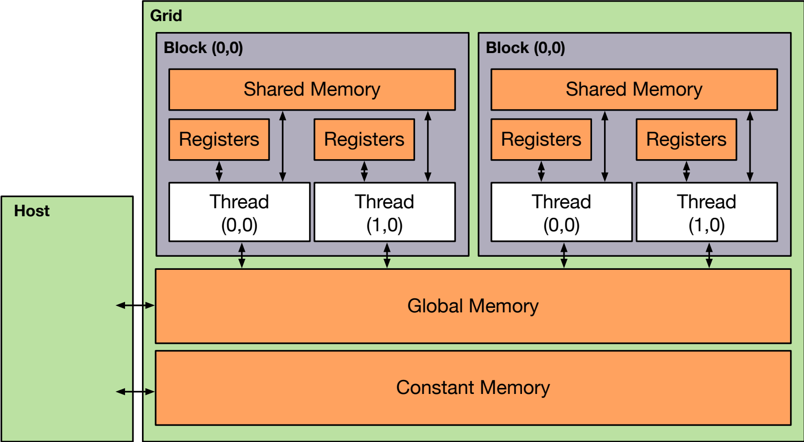
Pour apprendre à utiliser efficacement les types de mémoire CUDA dans un programme parallèle, il faut :

- comprendre l'importance de l'efficacité de l'accès à la mémoire
- connaître la distinction entre registres, mémoire partagée et mémoire globale
- comprendre la portée et la durée de vie des variables

Mémoire CUDA vue par le matériel



Mémoire CUDA vue par le programmeur



Déclaration des variables en CUDA pour un kernel

		Mémoire	Portée	Durée de vie
	int LocalVar;	registre	thread	thread
__device__ __shared__	int SharedVar;	shared	block	block
__device__	int GlobalVar;	global	grid	application
__device__ __constant__	int ConstantVar;	constant	grid	application

- __device__ est optionnel avec __shared__ et __constant__

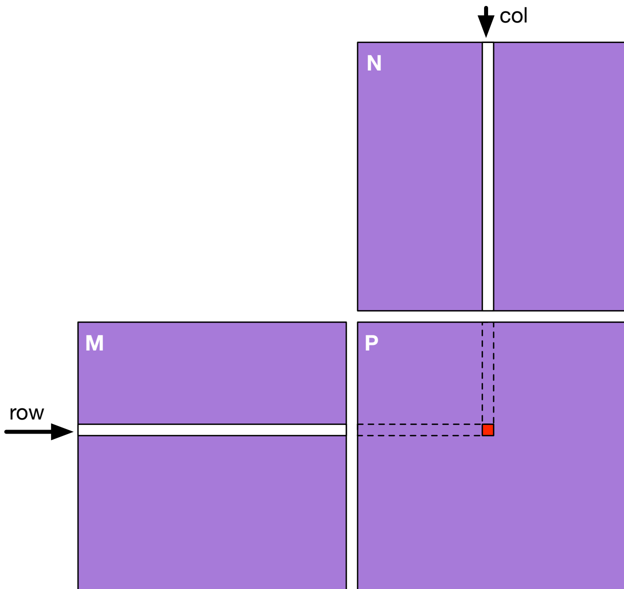
Exemple `__shared__ float ds_in[TILE_WIDTH][TILE_WIDTH];`

Focus *shared memory*

La *shared memory* est un type spécial de mémoire dont le contenu est explicitement défini et utilisé dans le code source d'un *kernel*.
Elle a les caractéristiques suivantes :

- Présente dans chaque SM
- Accès à une vitesse beaucoup plus élevée (en termes de latence et de débit) que la mémoire globale
- Portée limitées aux threads qui compose un block
- Durée de vie du block, le contenu disparait une fois que les threads terminent leur exécution
- Accès par instructions de chargement (load) et de stockage (store)
- Une forme de mémoire de type "scratchpad"

Exemple avec la multiplication matricielle : $P = M \times N$

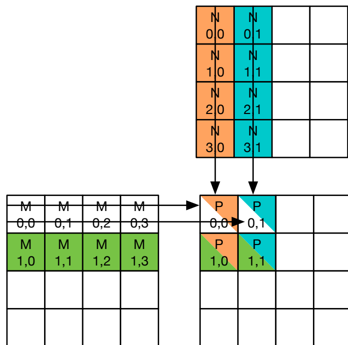


Exemple avec la multiplication matricielle : kernel

```
__global__
void MatrixMulKernel(float* M, float* N, float* P, int Width) {

  // Calculate the row index of the P element and M
  int Row = blockIdx.y*blockDim.y+threadIdx.y;

  // Calculate the column index of P and N
  int Col = blockIdx.x*blockDim.x+threadIdx.x;

  if ((Row < Width) && (Col < Width)) {
    float Pvalue = 0;
    // each thread computes one element of the block sub-matrix
    for (int k = 0; k < Width; ++k) {
      Pvalue += M[Row*Width+k]*N[k*Width+Col];
    }
    P[Row*Width+Col] = Pvalue;
  }

}
```

- Thread (i,j) compute $P_{i,j} = \sum_{k=0..Width-1} M_{i,k} N_{k,j}$

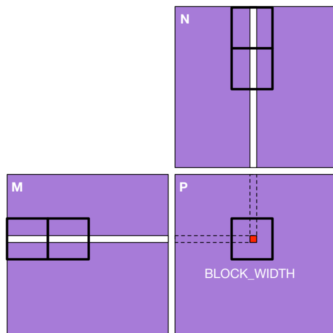
Accès mémoire



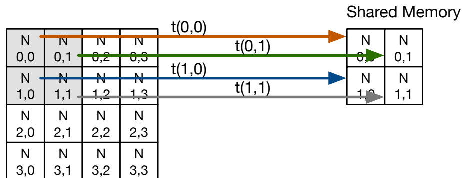
$\text{thread}_{0,0}$	$M_{0,0} * N_{0,0}$	$M_{0,1} * N_{1,0}$	$M_{0,2} * N_{2,0}$	$M_{0,3} * N_{3,0}$
$\text{thread}_{0,1}$	$M_{0,0} * N_{0,1}$	$M_{0,1} * N_{1,1}$	$M_{0,2} * N_{2,1}$	$M_{0,3} * N_{3,1}$
$\text{thread}_{1,0}$	$M_{1,0} * N_{0,0}$	$M_{1,1} * N_{1,0}$	$M_{1,2} * N_{2,0}$	$M_{1,3} * N_{3,0}$
$\text{thread}_{1,1}$	$M_{1,0} * N_{0,1}$	$M_{1,1} * N_{1,1}$	$M_{1,2} * N_{2,1}$	$M_{1,3} * N_{3,1}$

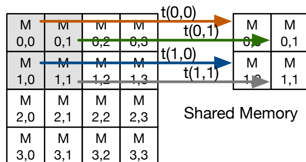
Multiplication matricielle par tuile
Tiled multiplication matrix

- Diviser l'exécution de chaque thread en plusieurs phases
- Les accès aux données par les threads d'un bloc se font sur une tuile de M et une tuile de N



Multiplication matricielle par tuile
Tiled multiplication matrix

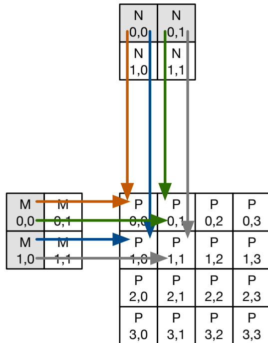




Multiplication matricielle par tuile
Tiled multiplication matrix

N 0,0	N 0,1	N 0,2	N 0,3
N 1,0	N 1,1	N 1,2	N 1,3
N 2,0	N 2,1	N 2,2	N 2,3
N 3,0	N 3,1	N 3,2	N 3,3

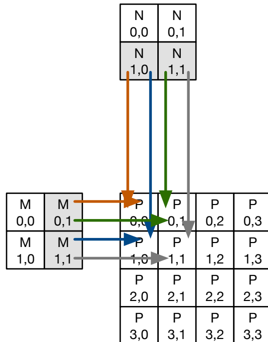
M 0,0	M 0,1	M 0,2	M 0,3
M 1,0	M 1,1	M 1,2	M 1,3
M 2,0	M 2,1	M 2,2	M 2,3
M 3,0	M 3,1	M 3,2	M 3,3



Multiplication matricielle par tuile
Tiled multiplication matrix

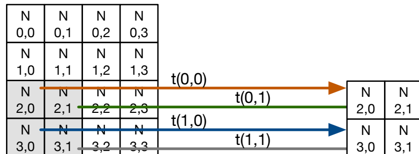
N 0,0	N 0,1	N 0,2	N 0,3
N 1,0	N 1,1	N 1,2	N 1,3
N 2,0	N 2,1	N 2,2	N 2,3
N 3,0	N 3,1	N 3,2	N 3,3

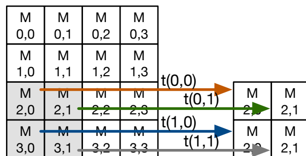
M 0,0	M 0,1	M 0,2	M 0,3
M 1,0	M 1,1	M 1,2	M 1,3
M 2,0	M 2,1	M 2,2	M 2,3
M 3,0	M 3,1	M 3,2	M 3,3



Multiplication matricielle par tuile

Tiled multiplication matrix



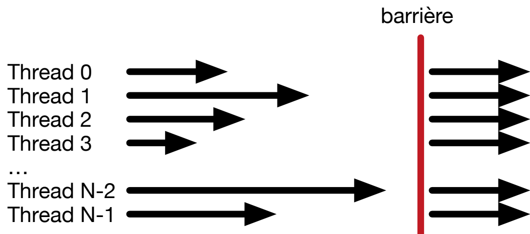


Code du kernel

```
__global__
void MatrixMulKernel(float* M, float* N, float* P, IntWidth)
{
  __shared__ float ds_M[TILE_WIDTH][TILE_WIDTH];
  __shared__ float ds_N[TILE_WIDTH][TILE_WIDTH];

  int bx = blockIdx.x; int by  = blockIdx.y;
  int tx = threadIdx.x; int ty = threadIdx.y;

  int Row = by * blockDim.y+ ty;
  int Col = bx* blockDim.x+ tx;
  float Pvalue= 0;

  // Loop over the M and N tiles required to compute the P element
  for (int p = 0; p < n/TILE_WIDTH; ++p) {
    // Collaborative loading of M and N tiles into shared memory
    ds_M[ty][tx] = M[Row*Width + p*TILE_WIDTH+tx];
    ds_N[ty][tx] = N[(t*TILE_WIDTH+ty)*Width + Col];

    // Usage
    for (int i = 0; i < TILE_WIDTH; ++i) Pvalue+= ds_M[ty][i] * ds_N[i][tx];

  }
  P[Row*Width+Col] = Pvalue;
}
```

Attention dangers : synchronisation

Que se passe-t-il si un thread n'a pas chargé la mémoire avant de commencer les calculs ?




```
// synchronize threads in this block
__syncthreads();
```

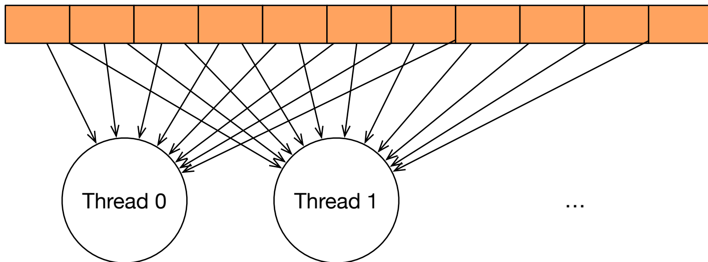
Cet appel garantit que chaque thread du *block* a terminé ses instructions avant le `__syncthreads()`

Exemple avec la multiplication matricielle : kernel

```
__global__
void MatrixMulKernel(float* M, float* N, float* P, IntWidth)
{
  __shared__ float ds_M[TILE_WIDTH][TILE_WIDTH];
  __shared__ float ds_N[TILE_WIDTH][TILE_WIDTH];

  int bx = blockIdx.x; int by  = blockIdx.y;
  int tx = threadIdx.x; int ty = threadIdx.y;

  int Row = by * blockDim.y+ ty;
  int Col = bx* blockDim.x+ tx;
  float Pvalue= 0;

  // Loop over the M and N tiles required to compute the P element
  for (int p = 0; p < n/TILE_WIDTH; ++p) {
    // Collaborative loading of M and N tiles into shared memory
    ds_M[ty][tx] = M[Row*Width + p*TILE_WIDTH+tx];
    ds_N[ty][tx] = N[(t*TILE_WIDTH+ty)*Width + Col];
    __syncthreads();

    // Usage
    for (int i = 0; i < TILE_WIDTH; ++i) Pvalue+= ds_M[ty][i] * ds_N[i][tx];
    __syncthreads();
  }
  P[Row*Width+Col] = Pvalue;
}
```

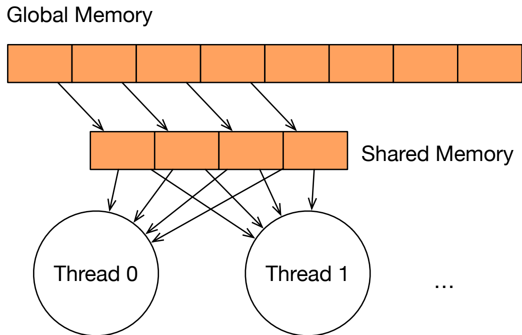
Technique de décomposition en tuiles

Réduire l'effet limitatif de la bande passante mémoire sur les performances des kernels parallèles



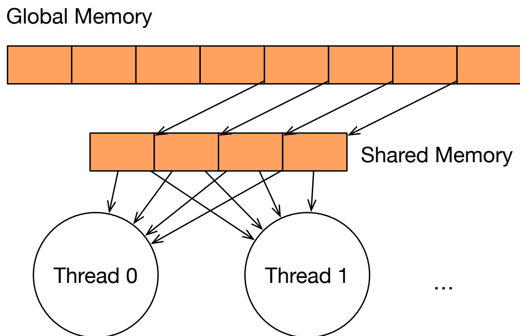
Technique de décomposition en tuiles

Réduire l'effet limitatif de la bande passante mémoire sur les performances des kernels parallèles



Technique de décomposition en tuiles

Réduire l'effet limitatif de la bande passante mémoire sur les performances des kernels parallèles



Le calcul avec des tuiles dans les grandes lignes

- Identifier une tuile de la mémoire globale qui est utilisée par de multiples threads
- Charger la tuile de la mémoire globale dans la mémoire sur le SM
- Utiliser la synchronisation des barrières pour s'assurer que tous les threads ont finis le chargement mémoire
- Avoid plusieurs threds pour accéder à leurs données à partir de la mémoire partagée
- Utiliser la synchronisation des barrières pour s'assurer que tous les threads ont terminé leur calcul
- Passer à la tuile suivante