Calcul parallèle avancé

ENSIMAG 3A - IF

11 février 2014

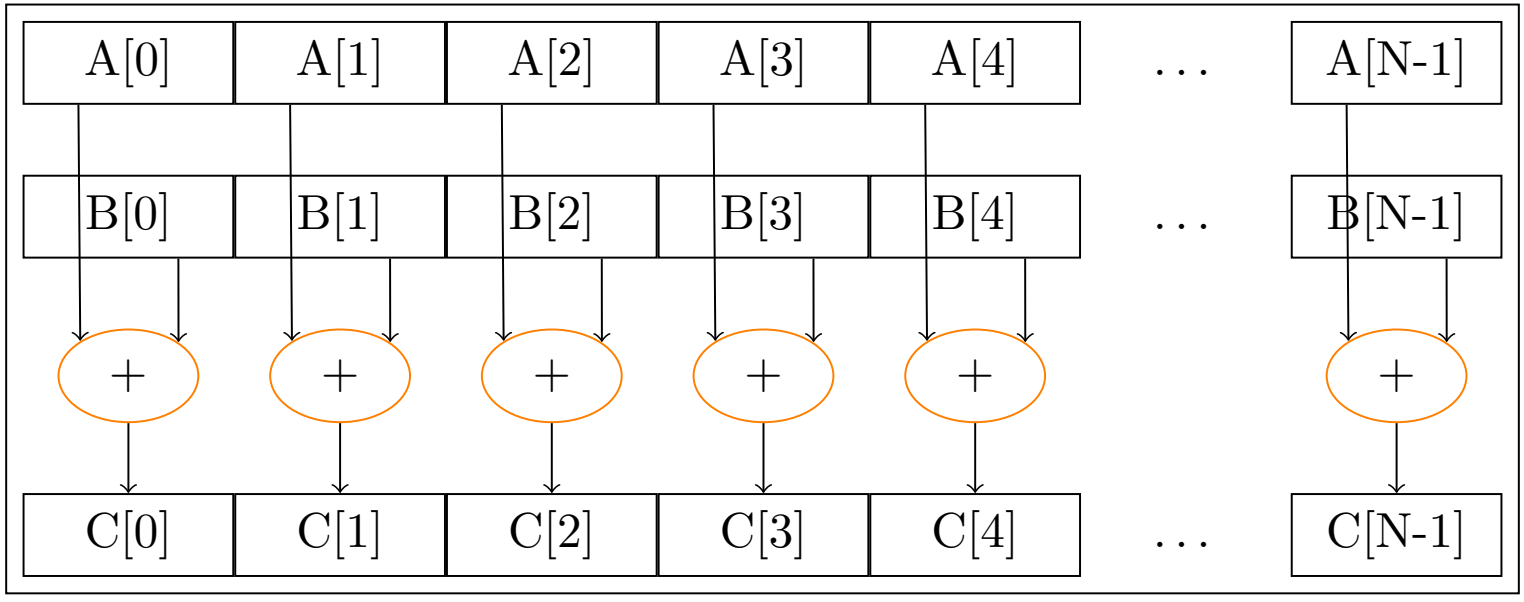
Chapitre 2 - CUDA et quelques opérations de bases

Objectifs

La programmation sur les accélérateurs s'organisent autour de trois notions

- Organisation des threads
- Interface de programmation pour l'exécution parallèle
- La relation entre l'indexage des threads et les données

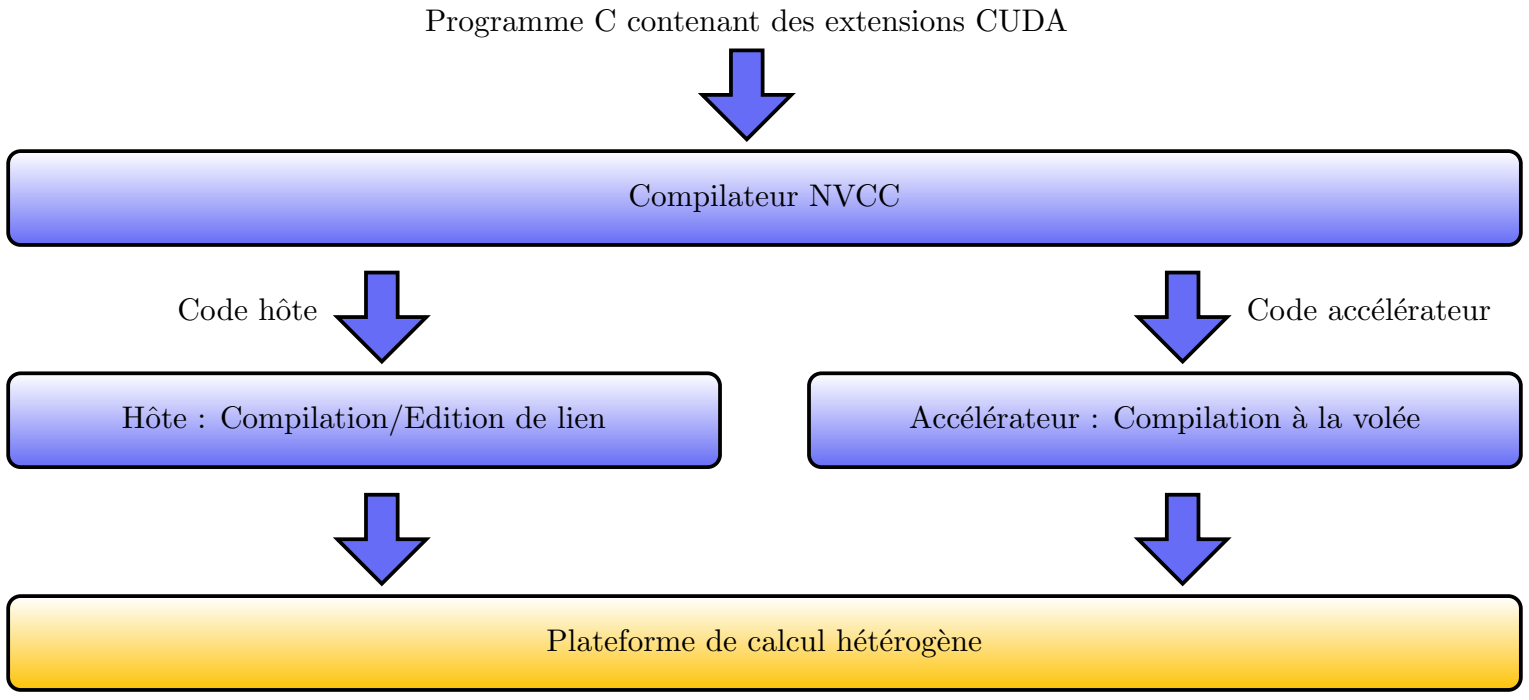
Parallélisme de données



Modèle d'exécution pour accélérateurs

- Programmation toujours hybride sur un système hétérogène
- Langage de programmation est le C
- Partie séquentielle exécutée sur l'hôte : code C
- Partie parallèle exécutée sur l'accélérateur : code C noyau

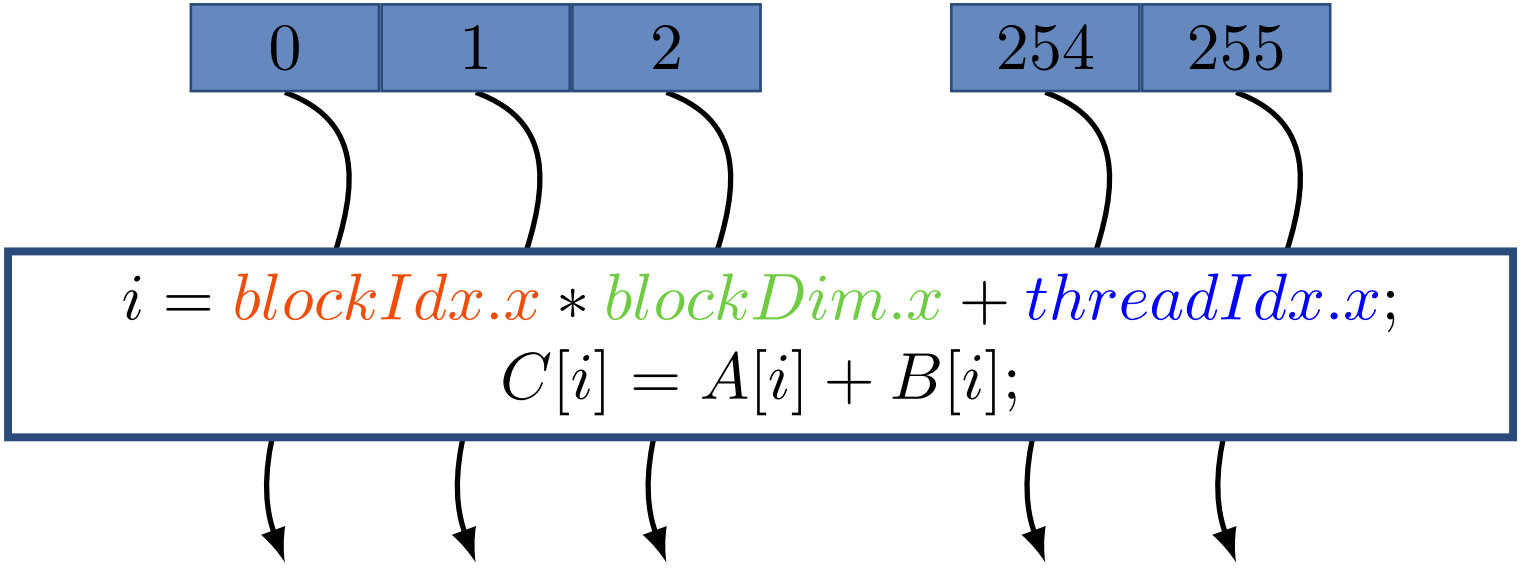
Compilation pour CUDA



- Compilation du code accélérateur

```
export PATH=$PATH:/opt/cuda/bin
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/opt/cuda/lib64
# Compilation du code
nvcc -c prog1.cu
# Compilation de l'executable
gcc -o test -L/opt/cuda/lib64 -lcudart prog1.o
```

Threads parallèles



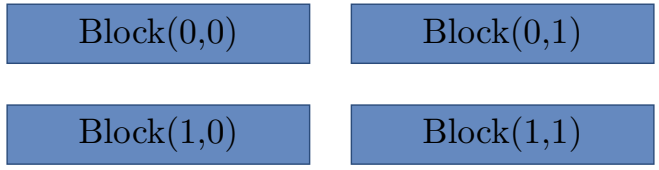
Un noyau d'accélérateur est exécuté par une une grille de threads

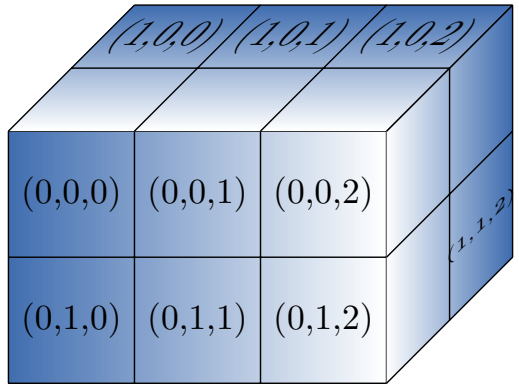
- Tous les threads d'une grille exécutent le même noyau CUDA.
- Chaque thread a son propre index qui est utilisé pour calculer l'adresse mémoire des données et déterminer les flux

Coopération

- Chaque tableau de thread est divisé en plusieurs blocs
- A l'intérieur d'un bloc, les threads utilisent la mémoire partagée, les opérations atomiques et une synchronisation par barrière.
- Les threads entre blocs distincts n'interagissent pas/

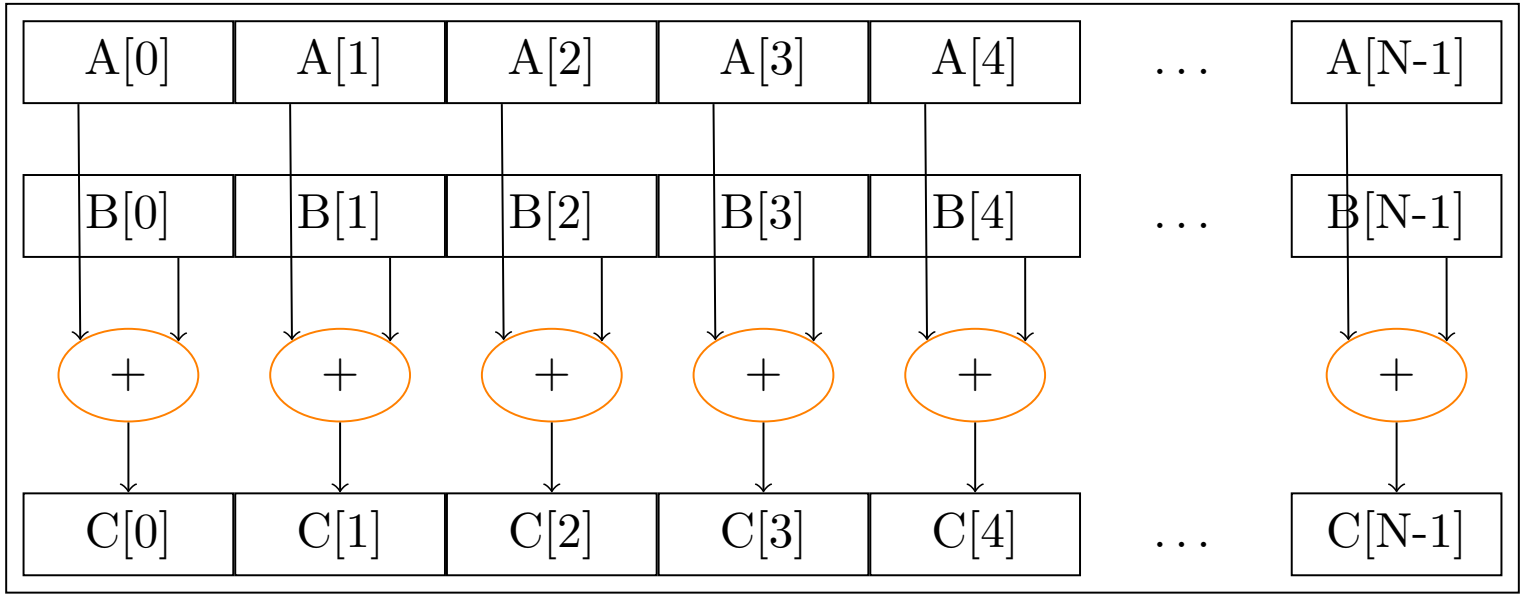
Index





- Chaque thread utilise des indices pour identifier les données sur lesquelles il travaille.
- `blockIdx` peut-être 1D, 2D ou 3D.
- `threadIdx` peut-être 1D, 2D ou 3D.
- Indexage permet de simplifier l'accès mémoire pour les données multidimensionnelle.

Exemple basique



Code séquentiel

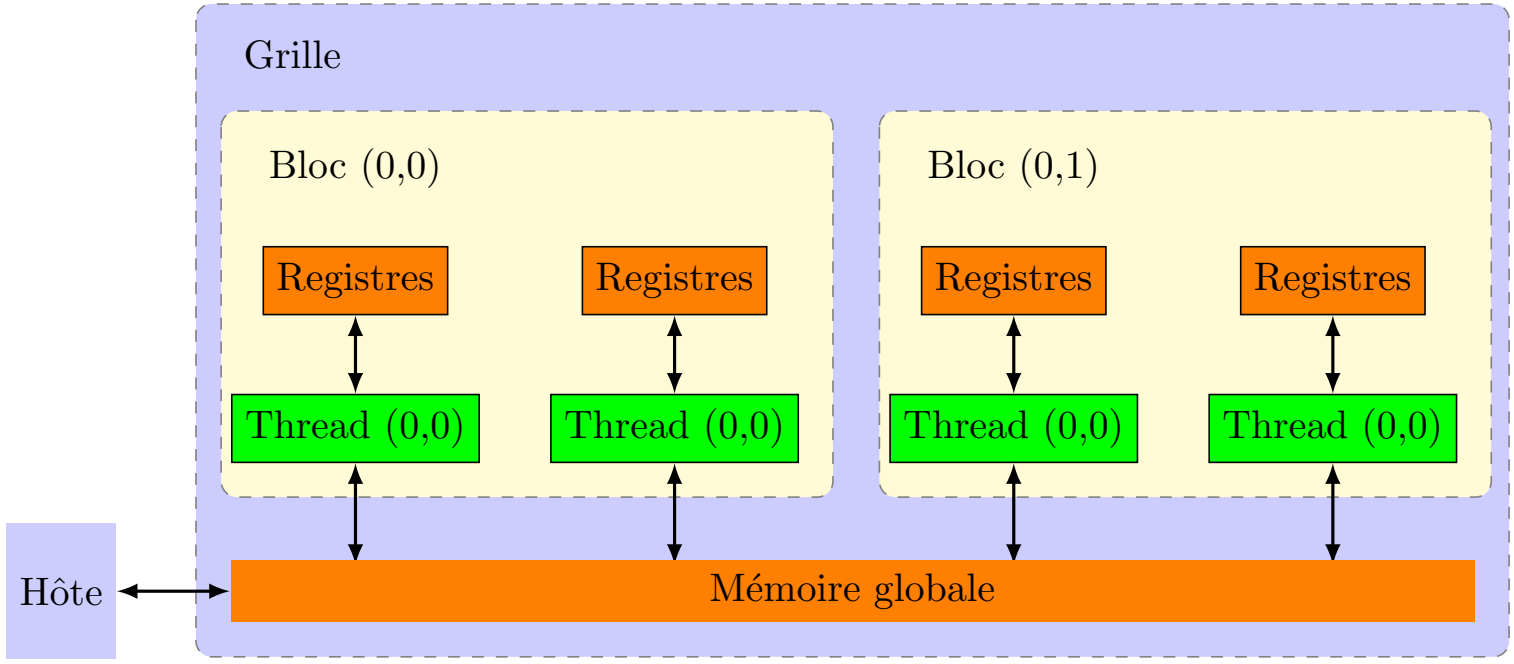
```
// Calcul effectif de la somme C = A+B                                    1
void vecAdd(float* h_A, float* h_B, float* h_C, int n)                    2
{                                                                         3
      int i;                                                              4
      for (i = 0; i < n; i++) h_C[i] = h_A[i] + h_B[i];                   5
}                                                                         6
int main()                                                                7
{                                                                         8
      // Allocation memoire de h_A, h_B, et h_C                           9
      // I/O pour lire les elements de h_A et h_B                         10
      vecAdd(h_A, h_B, h_C, N);                                           11
}                                                                         12
```

Code hétérogène

```
#include <cuda.h>
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
      int size = n* sizeof(float);
      float* d_A, d_B, d_C;
      // 1. Allocation de la memoire sur l'accelerateur pour A, B, et C
      // Copie de A et B sur l'accelerateur
      // 2. Execution du code noyau
      // L'accelerateur effectue reellement le travail
      // 3. Copie de C depuis la memoire de l'accelerateur
      // Libere la memoire sur l'accelerateur
}
```

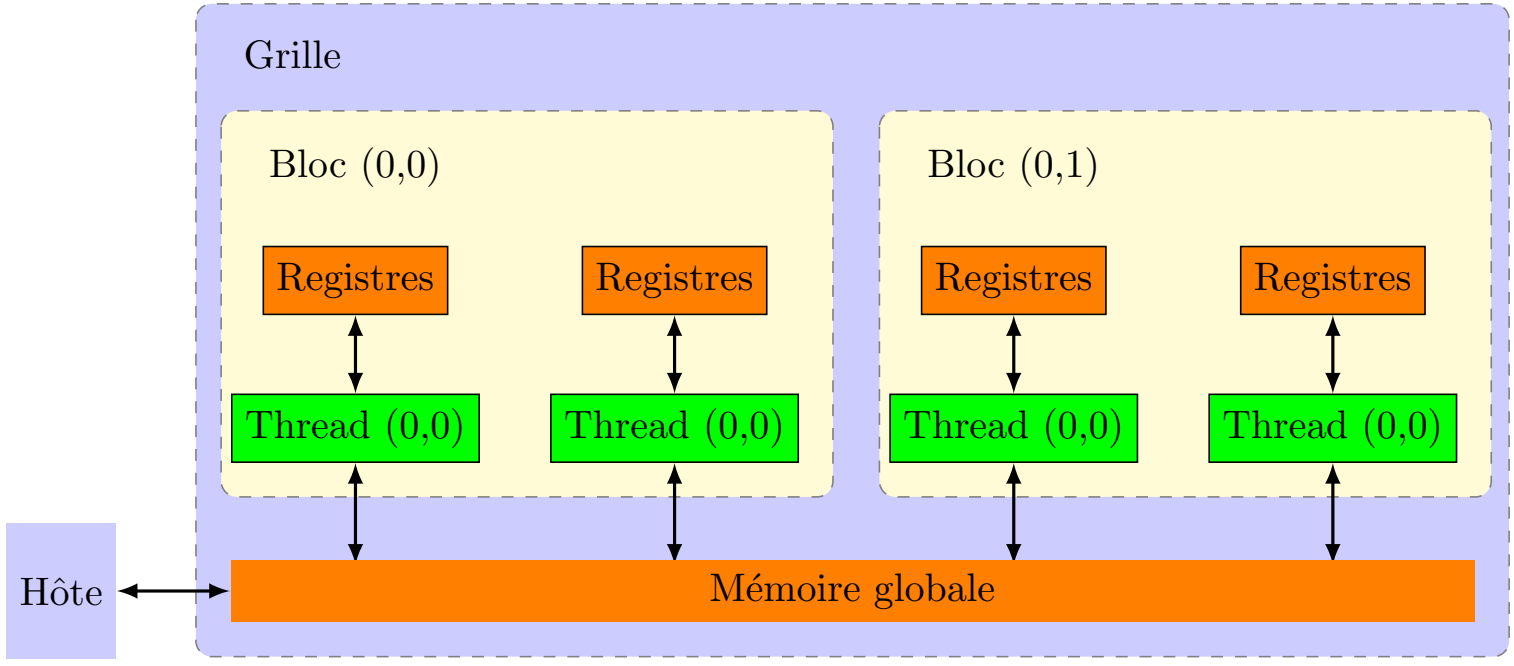
Gestion de la mémoire

- L'écriture et la lecture est privé à chaque thread sur les registres
- L'écriture et la lecture est partagée par tous les threads en mémoire globale
- L'hôte transfer les données pour chaque mémoire globale.



Allocation mémoire

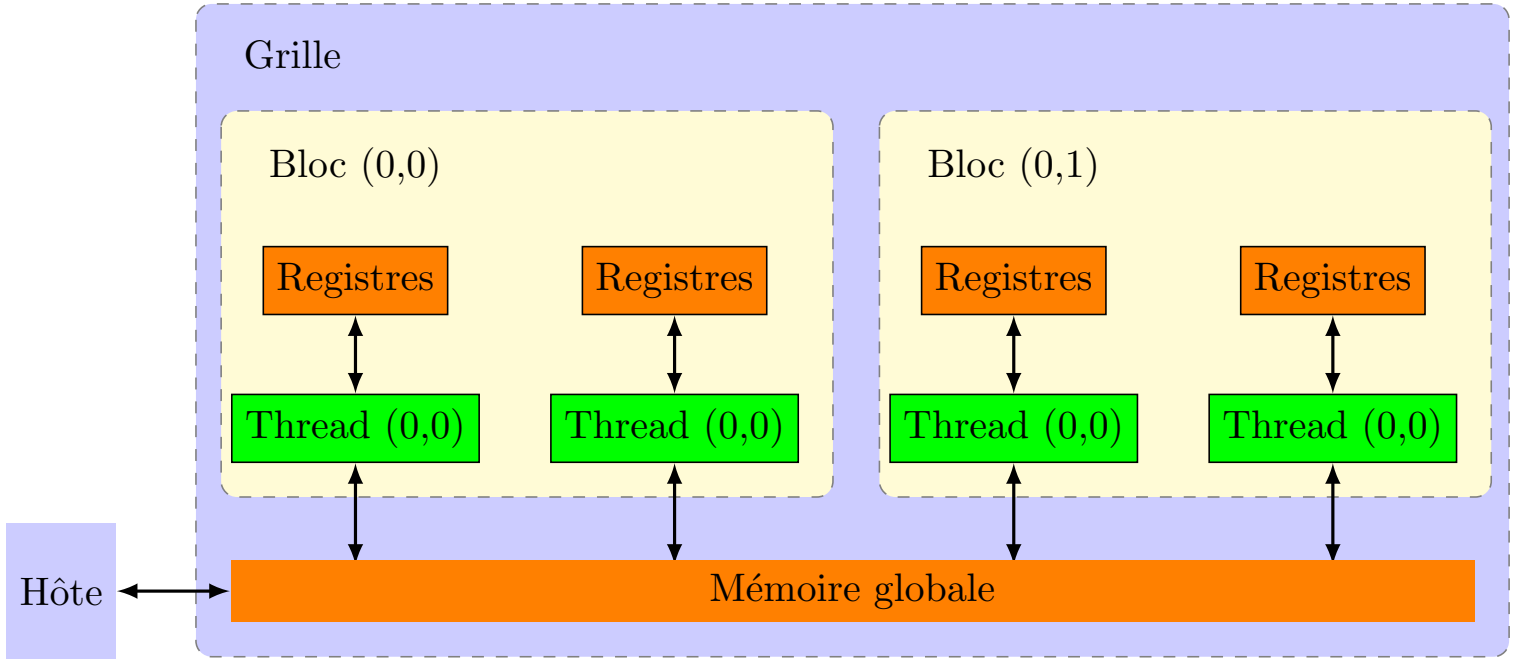
- `cudaMalloc()`
  - Alloue la mémoire sur l'accélérateur
  - Prends 2 paramètres : l'adresse du pointer de l'objet à allouer et la taille de l'objet en octets.
- `cudaFree()`
  - Libère la mémoire globale de l'objet.
  - Prends 1 paramètre :le pointeur de l'objet.



Transfert de données

`cudaMemcopy()`

- Transfère les données vers la mémoire globale
- Prends 4 paramètres :
  - Le pointeur de la destination
  - Le pointeur de la source
  - Le nombre d'octets
  - La direction du transfert
- Le transfer de données est asynchrone



Code de l'hôte

```
void vecAdd(float* h_A, float* h_B, float* h_C, int n)                    1
{                                                                         2
      int size = n * sizeof(float); float* d_A, d_B, d_C;                 3
      cudaMalloc((void **) &d_A, size);                                   4
      cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);                 5
      cudaMalloc((void **) &d_B, size);                                   6
      cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);                 7
      cudaMalloc((void **) &d_C, size);                                   8
      // Invocation du noyau                                              9
      cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);                 10
      cudaFree(d_A); cudaFree(d_B); cudaFree (d_C);                       11
}                                                                         12
```

Programmation défensive

les accélérateurs ne contiennent pas d'outils qui assure la consistance des données. Il est nécessaire de vérifier la manière dont se déroules les opérations

```
cudaError_t err = cudaMalloc((void **) &d_A, size);                    1
if (err != cudaSuccess) {                                              2
      printf("%s in %s at line %d\n", cudaGetErrorString(err), __FILE__, 3
            __LINE__);
      exit(EXIT_FAILURE);                                              4
}                                                                      5
```

Code du noyau

```
// Calcul de la somme C = A+B
// Chaque thread execute une addition
__global__ void vecAddKernel(float* A, float* B, float* C, int n)
{
      int i = threadIdx.x + blockDim.x * blockIdx.x;
      if(i<n) C[i] = A[i] + B[i];
}

int vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
      // Allocation des tableaux d_A, d_B, d_C
      // Calcul du nombre de blocs de 256 threads chacun
      int nblock=ceil(n/256.0);
      vecAddKernnel<<<ceil(n/256.0),256>>>(d_A, d_B, d_C, n); }
}
```

Modifier la géométrie de la grille

```
int vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
      // Allocation des tableaux d_A, d_B, d_C
      // Calcul du nombre de blocs de 256 threads chacun
      dim3 DimGrid((n-1)/256 + 1,1,1);
      // S'assurer que le nombre de thread est suffisant pour traiter l'
          ensemble
      // des donnees
      dim3 DimBlock(256 ,1,1);
      vecAddKernnel<<<DimGrid,DimBlock>>>(d_A, d_B, d_C, n); }
}
```

Déclaration de fonctions

	Exécuté sur	Appelable seulement de
`__device__ float DeviceFunc()`	Accé	Accé
`__global__ void KernelFunc()`	Accé	Hôte
`__host__ float HostFunc()`	Hôte	Hôte

- `__global__` définit une fonction noyau
- Une fonction de noyau retourne obligatoirement **`void`**
- `__device__` et `__host__` peuvent être utilisés ensemble.

Transformation d'un indexage 2D en indexage 1D

```
__global__ void PictureKernel(float* d_Pin, float* d_Pout, int n,int m)
{
      // Calcule le numero de la ligne de d_Pin et de d_Pout a manipuler
      int Row = blockIdx.y*blockDim.y + threadIdx.y;
      // Calcule le numero de la colonne de d_Pin et de d_Pout a
          manipuler
      int Col = blockIdx.x*blockDim.x + threadIdx.x;
      // Chaque thread realise une operation
      if in range if ((Row < m) && (Col < n))
      {
            d_Pout[Row*n+Col] = 2*d_Pin[Row*n+Col];
      }
}
```

Ce découpage permet de manipuler des grilles non régulières par rapport au nombre de threads.

Multiplication matrice-matrice

```
void MatrixMulOnHost(float* M, float* N, float* P, int Width) {
      for (int i = 0; i < Width; ++i)
            for (int j = 0; j < Width; ++j)
            {
                  double sum = 0;
                  for (int k = 0; k < Width; ++k)
                  {
                        double a = M[i * Width + k];
                        double b = N[k * Width + j];
                        sum += a * b;
                  }
                  P[i * Width + j] = sum;
            }
}
```

Sur un noyau

$P_{0,0}$	$P_{0,1}$	$P_{0,2}$	$P_{0,3}$
$P_{1,0}$	$P_{1,1}$	$P_{1,2}$	$P_{1,3}$
$P_{2,0}$	$P_{2,1}$	$P_{2,2}$	$P_{2,3}$
$P_{3,0}$	$P_{3,1}$	$P_{3,2}$	$P_{3,3}$

- Chaque bloc 2D de thread calcule nx^2 sous matrices de la matrice finale : chaque bloc a nx^2 threads.
- $nx = 4$ et $nx_{loc} = 2$: chaque bloc a $2 \times 2 = 4$ threads.
- $nx/nx_{loc} = 2$: Utilise $2 \times 2 = 4$ blocs.

Sur un noyau

$P_{0,0}$	$P_{0,1}$	$P_{0,2}$	$P_{0,3}$	$P_{0,4}$	$P_{0,5}$	$P_{0,6}$	$P_{0,7}$
$P_{1,0}$	$P_{1,1}$	$P_{1,2}$	$P_{1,3}$	$P_{1,4}$	$P_{1,5}$	$P_{1,6}$	$P_{1,7}$
$P_{2,0}$	$P_{2,1}$	$P_{2,2}$	$P_{2,3}$	$P_{2,4}$	$P_{2,5}$	$P_{2,6}$	$P_{2,7}$
$P_{3,0}$	$P_{3,1}$	$P_{3,2}$	$P_{3,3}$	$P_{3,4}$	$P_{3,5}$	$P_{3,6}$	$P_{3,7}$
$P_{4,0}$	$P_{4,1}$	$P_{4,2}$	$P_{4,3}$	$P_{4,4}$	$P_{4,5}$	$P_{4,6}$	$P_{4,7}$
$P_{5,0}$	$P_{5,1}$	$P_{5,2}$	$P_{5,3}$	$P_{5,4}$	$P_{5,5}$	$P_{5,6}$	$P_{5,7}$
$P_{6,0}$	$P_{6,1}$	$P_{6,2}$	$P_{6,3}$	$P_{6,4}$	$P_{6,5}$	$P_{6,6}$	$P_{6,7}$
$P_{7,0}$	$P_{7,1}$	$P_{7,2}$	$P_{7,3}$	$P_{7,4}$	$P_{7,5}$	$P_{7,6}$	$P_{7,7}$

- $nx = 8$ et $nx_{loc} = 2$: chaque bloc a $2 \times 2 = 4$ threads.
- $nx/nx_{loc} = 4$: Utilise $4 \times 4 = 16$ blocs.

Sur un noyau

$P_{0,0}$	$P_{0,1}$	$P_{0,2}$	$P_{0,3}$	$P_{0,4}$	$P_{0,5}$	$P_{0,6}$	$P_{0,7}$
$P_{1,0}$	$P_{1,1}$	$P_{1,2}$	$P_{1,3}$	$P_{1,4}$	$P_{1,5}$	$P_{1,6}$	$P_{1,7}$
$P_{2,0}$	$P_{2,1}$	$P_{2,2}$	$P_{2,3}$	$P_{2,4}$	$P_{2,5}$	$P_{2,6}$	$P_{2,7}$
$P_{3,0}$	$P_{3,1}$	$P_{3,2}$	$P_{3,3}$	$P_{3,4}$	$P_{3,5}$	$P_{3,6}$	$P_{3,7}$
$P_{4,0}$	$P_{4,1}$	$P_{4,2}$	$P_{4,3}$	$P_{4,4}$	$P_{4,5}$	$P_{4,6}$	$P_{4,7}$
$P_{5,0}$	$P_{5,1}$	$P_{5,2}$	$P_{5,3}$	$P_{5,4}$	$P_{5,5}$	$P_{5,6}$	$P_{5,7}$
$P_{6,0}$	$P_{6,1}$	$P_{6,2}$	$P_{6,3}$	$P_{6,4}$	$P_{6,5}$	$P_{6,6}$	$P_{6,7}$
$P_{7,0}$	$P_{7,1}$	$P_{7,2}$	$P_{7,3}$	$P_{7,4}$	$P_{7,5}$	$P_{7,6}$	$P_{7,7}$

- $nx = 8$ et $nx_{loc} = 4$: chaque bloc a $4 \times 4 = 16$ threads.
- $nx/nx_{loc} = 2$: Utilise $2 \times 2 = 4$ blocs.

Code correspondant

```
// TILE_WIDTH est une constante #define
dim3 dimGrid(Width/TILE_WIDTH, Width/TILE_WIDTH, 1);
dim3 dimBlock(TILE_WIDTH, TILE_WIDTH, 1);
// Execute le calcul sur l'accelerateur !
MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, Width);


__global__ void MatrixMulKernel(float* M, float* N, float* P, int Width)
{
      int Row = blockIdx.y*blockDim.y+threadIdx.y;
      int Col = blockIdx.x*blockDim.x+threadIdx.x;
      if ((Row < Width) && (Col < Width))
      {
            float Pvalue = 0;
            for (int k = 0; k < Width; ++k)
                  Pvalue += M[Row*Width+k] * N[k*Width+Col];
            P[Row*Width+Col] = Pvalue;
      }
}
```

Notion

A l'intérieur d'un bloc, les threads s'exécutent par groupe de 32.

- Si le bloc n'est pas divisible par 32, alors certains processeurs ne feront rien
- Dans les blocs multidimensionnels, les threads sont ordonnés par dimension, et ensuite séparé en warp de taille 32.
- Tous les threads d'un warp exécutent la même tâche au même moment.
- Différents warp sont exécutés indépendamment, sauf en cas de synchronisation.

Divergence

- Si des threads du même warp on besoin de réaliser des chose différentes

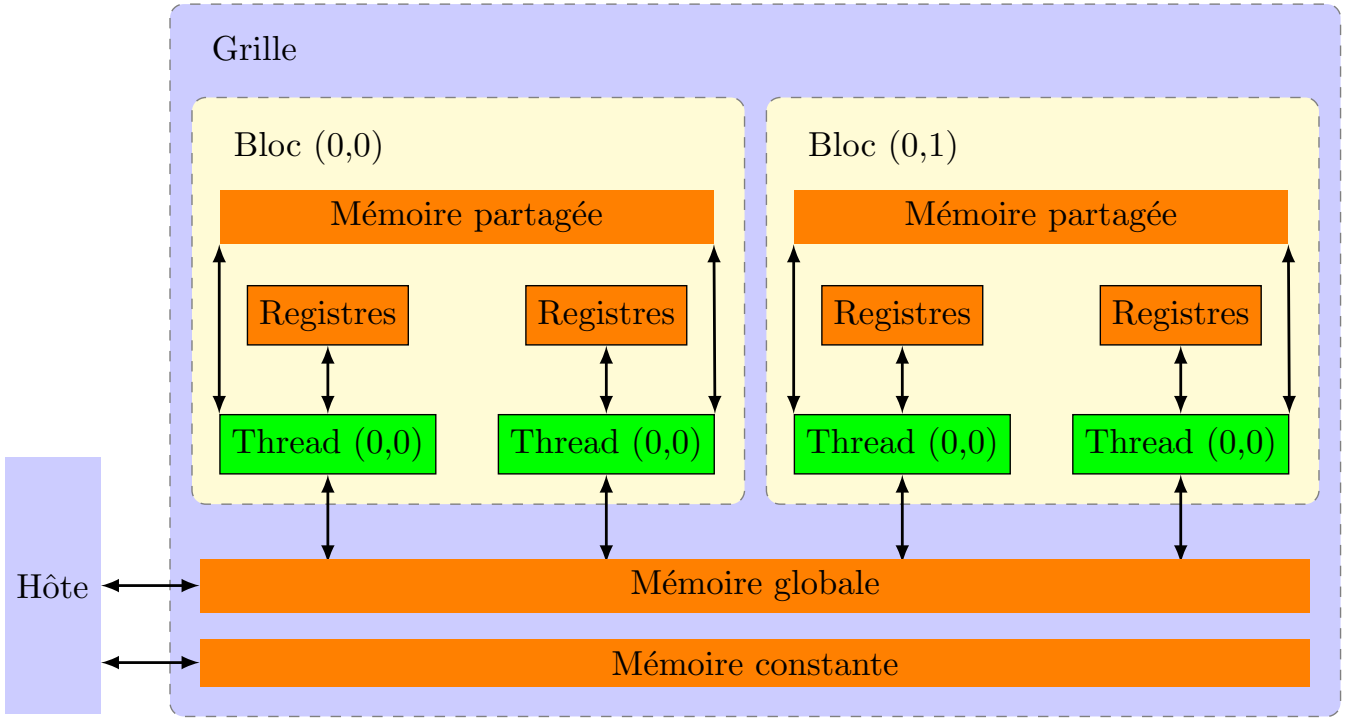
```
if (x<0.0)                                                    1
      z = x-2.0;                                              2
else                                                          3
      z = sqrt(x);                                            4
```

- Dans un cas simple, tous les threads calculeront tous les prédicats logiques

```
p = (x<0.0);                                                  1
if (p) z = x-2.0; // single instruction                       2
if (!p) z = sqrt(x);                                          3
```

- Dans la mesure du possible, éviter au maximum les divergences.

Les différents niveaux



- Registres : accès très haut débit en lecture et écriture (1 cycle) par les threads.
- Mémoire partagée : accès haut débit en lecture et écriture (5 cycles) par les blocs
- Mémoire globale : accès bas débit en lecture et écriture (500 cycles) par la grille
- Mémoire constante : accès haut débit en lecture seule (5 cycles)

Les variables

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

- `__device__` est optionnel lorsqu'il est utilisé avec `__constant__` ou `__shared__`
- Le choix de l'emplacement de déclaration des variables dépends de la nécessité pour l'hôte d'accéder à son contenu.