# UF I5AISE51
# Dimensionnement et évaluation des architectures

## Introduction à la programmation massivement parallèle

—

## Partie 2
## Allocation des données et exécution d'un kernel

—

P.-E. Hladik

INSA Toulouse

4 janvier 2021

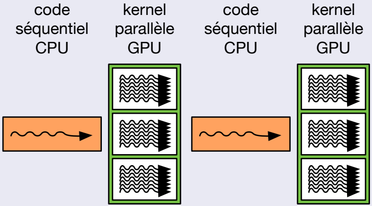
# Plan

# Structure d'un programme C en CUDA

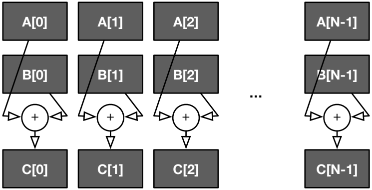
## *host* et *device*

- *host* = CPU
- *device* = GPU

## Principe d'exécution : fonction et kernel

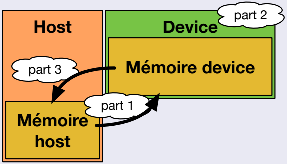


# Exemple avec une addition vectorielle




```c
// Compute vector sum C = A + B
void vecAdd (float *h_A , float *h_B , float * h_C , int n)
{
  int i;
  for (i = 0; i<n; i++) h_C [i] = h_A [i] + h_B [i];
}

int main()
{
  // Memory allocation for for h_A , h_B , and h_C
  // I/O to read  h_A and h_B , N elements
  ...
  vecAdd (h_A , h_B , h_C , N);
}
```

# Structure basique d'un code CUDA




```
#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;
    // Part 1
    // Allocate device memory for A, B, and C
    // copy A and B to device memory

    // Part 2
    // Kernel launch code - the device performs the actual vector addition

    // Part 3
    // copy C from the device memory
    // Free device vectors
}
```

# Allocation mémoire : cudaMalloc

```
#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;
    // Allocate device memory for A, B, and C
    cudaMalloc((void **) &d_A, size);
    cudaMalloc((void **) &d_B, size);
    cudaMalloc((void **) &d_C, size);

    // copy A and B to device memory

    // Part 2
    // Kernel launch code - the device performs the actual vector addition

    // Part 3
    // copy C from the device memory
    // Free device vectors
}
```

# Désallocation mémoire : cudaFree

```
#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;
    // Allocate device memory for A, B, and C
    cudaMalloc((void **) &d_A, size);
    cudaMalloc((void **) &d_B, size);
    cudaMalloc((void **) &d_C, size);

    // copy A and B to device memory

    // Part 2
    // Kernel launch code - the device performs the actual vector addition

    // Part 3
    // copy C from the device memory
    // Free device vectors
    cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
}
```

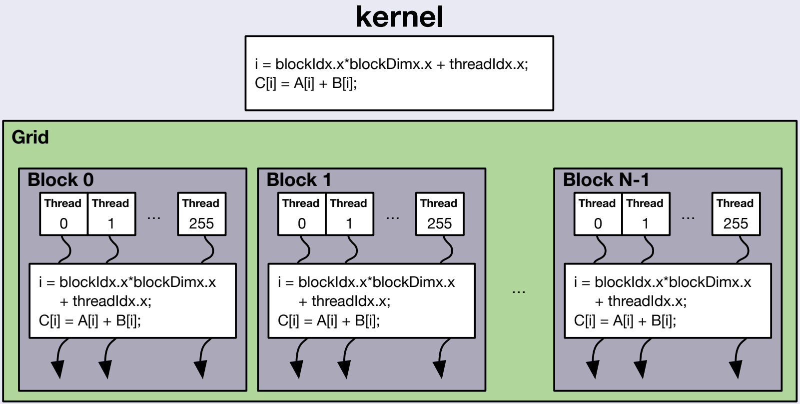
# Copie mémoire : cudaMemcpy

```
#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;
    // Allocate device memory for A, B, and C
    cudaMalloc((void **) &d_A, size);
    cudaMalloc((void **) &d_B, size);
    cudaMalloc((void **) &d_C, size);

    // copy A and B to device memory
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Part 2
    // Kernel launch code - the device performs the actual vector addition

    // Part 3
    // copy C from the device memory
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
    // Free device vectors
    cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
}
```

# Si on voulait bien faire les choses : gestion des erreurs

```
cudaError_t  err = cudaMalloc((void **) &d_A, size);

if (err != cudaSuccess)  {
   printf("%s in %s at line %d\n",   cudaGetErrorString(err), __FILE__,
   __LINE__);
   exit(EXIT_FAILURE);
}
```

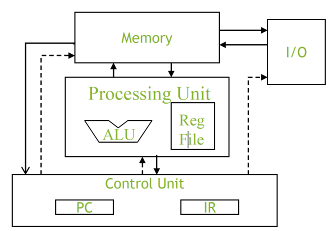
# Kernel, grid, block et thread




- Les threads d'un block coopèrent via une **shared memory**, **atomic operations** et **barrier synchronization**
- les threads dans des blocs différents ne coopèrent pas

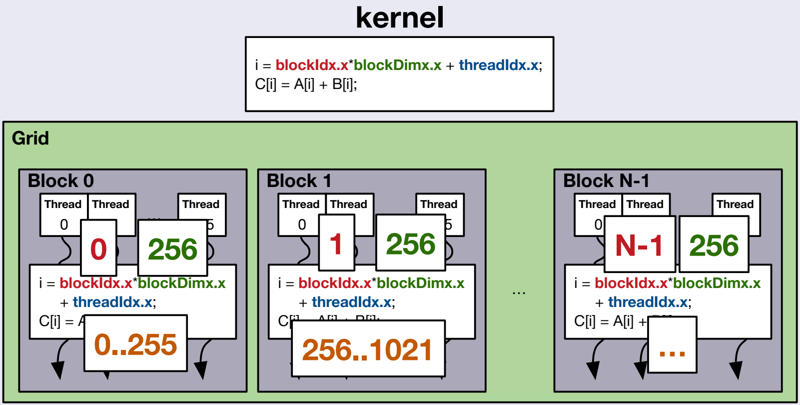
# Execution

- Un thread peut être vu comme un processeur de type Von-Neumann
- tous les threads d'un bloc exécutent la même instruction (Single Program Multiple Data)



# Thread index

- Chaque thread a un index qui peut être utilisé pour manipuler les adresses mémoires et faire des choix d'exécution



# Définition d'un kernel en CUDA

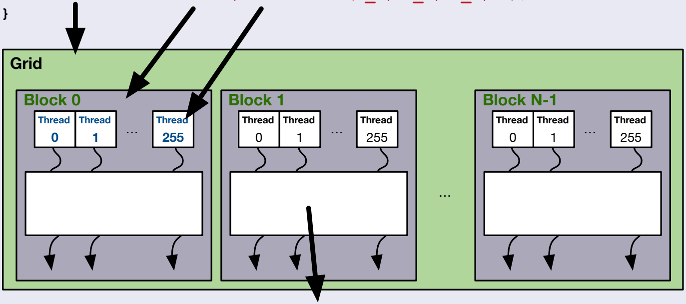
```
// Compute vector sum C = A + B
// Each thread performs one pair-wise addition

__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];
}
```

# Execution d'un kernel

```
// Compute vector sum C = A + B
// Each thread performs one pair-wise 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];
}

void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
  // d_A, d_B, d_C allocations and copies omitted
  // Run ceil(n/256.0) blocks of 256 threads each
  int DimGrid = ceil(n/256.0);
  int DimBlock = 256;
  vecAddKernel<<<DimGrid,DimBlock>>>(d_A, d_B, d_C, n);
}
```

# Thread index

```
__host__
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
  int DimGrid = ceil(n/256.0);
  int DimBlock = 256;
  vecAddKernel<<<DimGrid,DimBlock>>>(d_A, d_B, d_C, n);
}
```




```
__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];
}
```

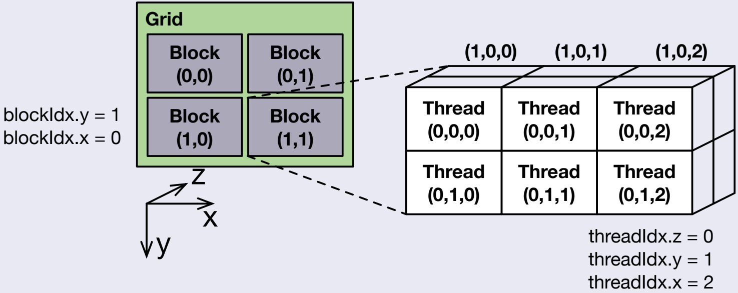
# Mots-clef en CUDA-C pour déclarer les fonctions

| | exécuté sur | appelé par |
|---|---|---|
| `__device__ float DeviceFunc()` | device | device |
| `__global__ void KernelFunc()` | device | host |
| `__host__ float HostFunc()` | host | host |

- `__global__` définit une fonction kernel
- Une fonction kernel doit retourner un `void` (mais peut prendre ce que l'on veut en argument)
- `__device__` et `__host__` peuvent être utilisés conjointement pour avoir deux versions objet de la même fonction.
- `__host__` est optionnel s'il est utilisé seul

# blockIdx et threadIdx

- les index de block et de thread sont des variables à trois dimensions
- cela simplifie les adresses mémoires quand on travaille avec des données multidimentionelles (image, volumes, etc.)



# Execution d'un kernel (multidim)

```
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
  // d_A, d_B, d_C allocations and copies omitted
  // Run ceil(n/256.0) blocks of 256 threads each
  dim3 DimGrid((n-1)/256 + 1, 1, 1);
  dim3 DimBlock(256, 1, 1);
  vecAddKernel<<<DimGrid,DimBlock>>>(d_A, d_B, d_C, n);
}
```