



GP-GPU

## CUDA basics

Stéphane Vialle



université  
PARIS-SACLAY

ÉCOLE DOCTORALE  
Sciences et technologies  
de l'information  
et de la communication (STIC)



Stephane.Vialle@centralesupelec.fr  
<http://www.metz.supelec.fr/~vialle>

1

## CUDA basics

- 1. Principle of execution of a CUDA program**
2. Variable definitions and CPU/GPU data transfers
3. Definition of a grid of blocks (of threads)
4. Definition of a 1<sup>st</sup> CUDA kernel
5. Execution of a CUDA kernel
6. Compiling a CUDA application

2

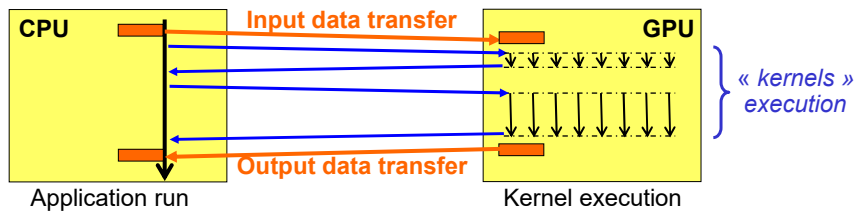
# Launching GPU calculations from the CPU

## Steps for running a CUDA application:

- CPU: launch of a C++ pgm (*main* function)
- CPU: initialization of variables, light calculations
- CPU: data transfer from CPU memory to GPU memory
- CPU: launch of remote calculations on the GPU  
→ GPU: execution of massively parallel "CUDA kernels"
- CPU: transfer of results from GPU memory to CPU memory

*The GPU kernel codes are transferred from the CPU.*

**Warning :** CPU/GPU transfers are time consuming !

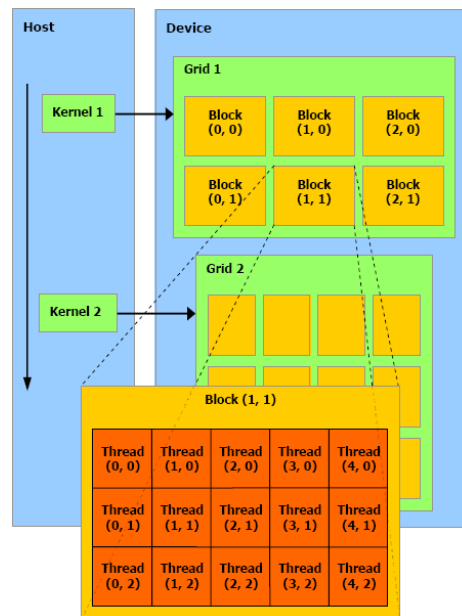


3

# Execution of thread block grids

The CPU program requests the execution of a grid of blocks of CUDA threads:

- identical threads
- threads organized in blocks, each block running on a single Stream Multiprocessor
- blocks organized within a grid, which distributes its blocks on all Stream Multiprocessors
- it exists 1D, 2D and 3D grids and blocks:
  - 1D grid of 2D blocks
  - 2D grid of 1D blocks
  - 2D grid of 2D blocks
  - ...



4

POLYTECH PARIS-SACLAY

## Principle of execution of a CUDA program

# Execution of thread block grids

**The CPU program requests the execution of a grid of blocks of CUDA threads:**

- the block scheduler distributes the blocks on the different Streams Multiprocessors (SMs)
- different GPUs will execute the same grid of blocks at different rates

5

POLYTECH PARIS-SACLAY

## Principle of execution of a CUDA program

# Thread block execution, *warp by warp*

Block scheduler, SIMT model

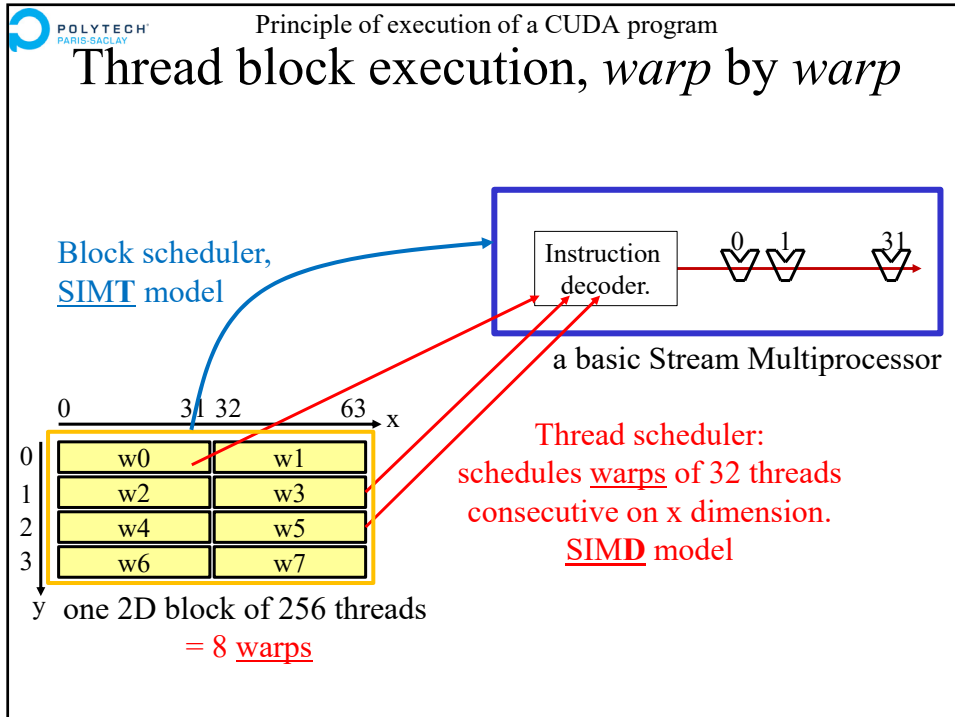
0 63 x

0 1 2 3 y

one 2D block of 256 threads

one Stream Multiprocessor

6



7

POLYTECH PARIS-SACLAY

## Principle of execution of a CUDA program

### Constraints on block size

- an instruction decoder drives 32 "hardware threads" (32 ALUs)
- the thread scheduler enables warps of 32 thread
- if possible, warps are made of threads consecutive in x (privileged dimension)

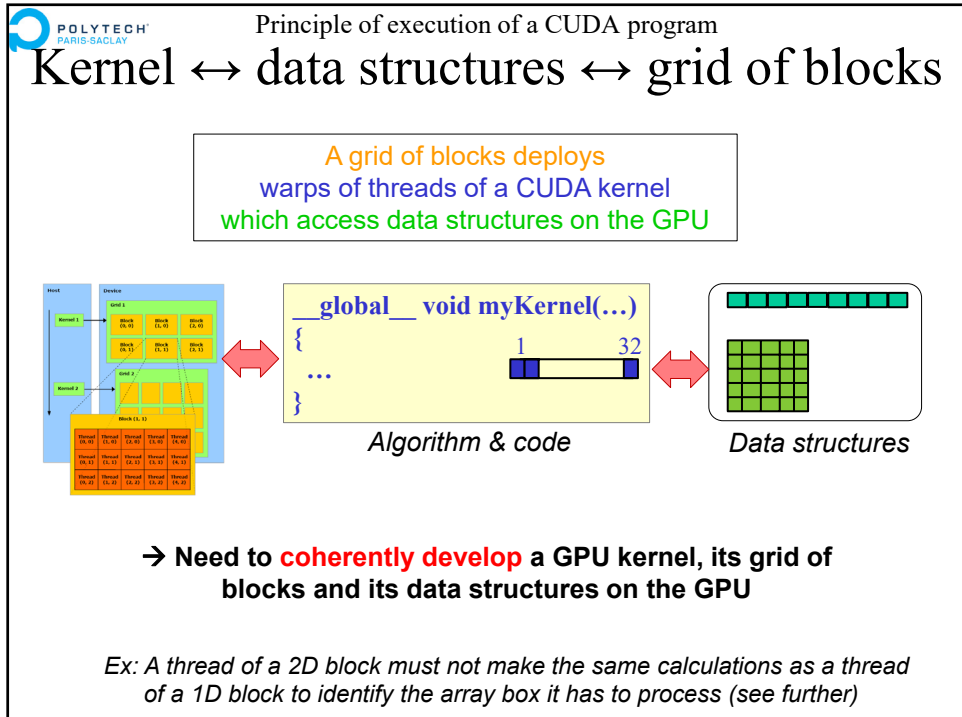
→ **Create blocks of at least 32 threads**  
otherwise part of the ALU will always be unused

→ **Create blocks with a thread count multiple of 32**  
otherwise the last warp will be incomplete and ALUs will be unused.

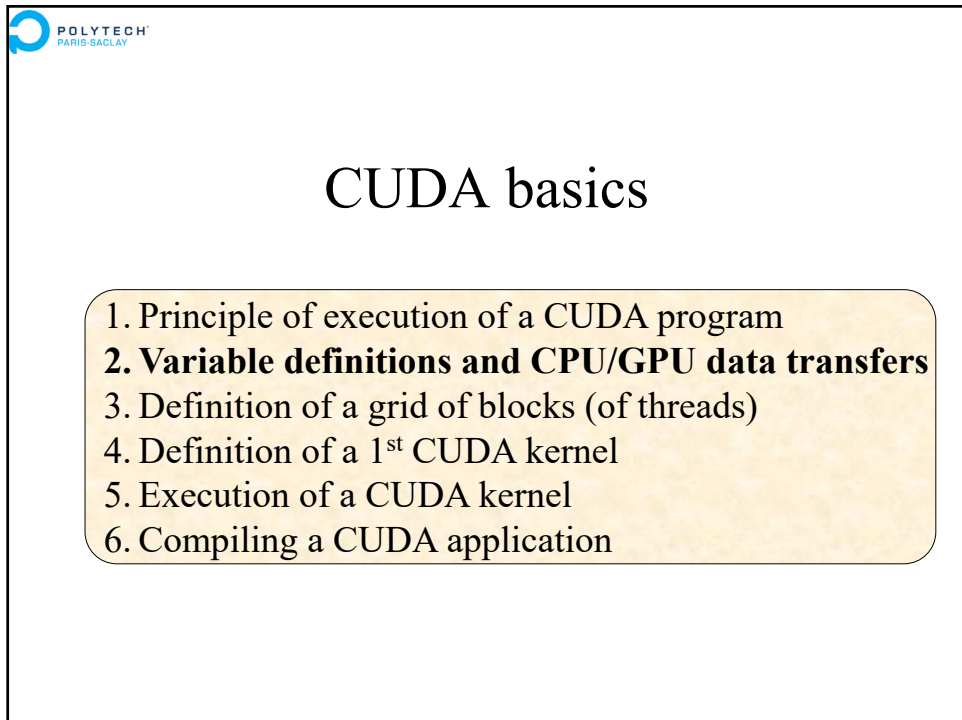
→ **Create blocks with an x-dimension multiple of 32**  
otherwise the "coalescence" will be poor (see further)

*But sometimes it still works very well with a block x-dimension multiple of 16...!*

8



9



10

## CUDA « qualifiers »

How CUDA "qualifiers" work:

	<u>__device__</u>	<u>__constant__</u>	<u>__shared__</u>
Variables	Global memory of the GPU	Constant memory of the GPU	Shared memory of a multiprocessor
	Lifetime of the application	Lifetime of the application	Lifetime of the block of threads
	R/W by GPU and CPU codes	W by CPU code R by GPU code	R/W by GPU code: cache memory of the Global GPU memory
Functions	<u>__device__</u>	<u>__host__</u> (default)	<u>__global__</u>
	Call from GPU run on GPU	Call from CPU run on CPU	Call on CPU run on GPU

11

## Data transfers CPU↔GPU

Variables allocated on the GPU at compile time ("symbol"):

```
float TabCPU[N]; // Array on CPU
__device__ float TabGPU[N]; // Array on GPU (symbol)
```

Transfer of a variable on the CPU to a "symbol" on the GPU:

```
// Copy all TabCPU array into TabGPU array
cudaMemcpyToSymbol(TabGPU, &TabCPU[0],
    sizeof(float)*N, 0,
    cudaMemcpyHostToDevice);

// Copy 2nd half of TabCPU array into 2nd half TabGPU array
cudaMemcpyToSymbol(TabGPU, &TabCPU[N/2],
    sizeof(float)*N/2, sizeof(float)*N/2,
    cudaMemcpyHostToDevice);
```

Transfer of a "symbol" on the GPU to a variable on the CPU:

```
// Copy all TabGPU array into TabCPU array
cudaMemcpyFromSymbol(&TabCPU[0], TabGPU,
    sizeof(float)*N, 0,
    cudaMemcpyDeviceToHost);
```

12

## Variable definitions and CPU/GPU data transfers Specificities of static variables

### Static variables on GPU

- Fully defined at compilation time
- Directly accessible from the GPU code  
→ no need to pass their addresses as kernel parameters
- BUT: any GPU code that uses these variables must be in the same file as the variable definition !**  
(you quickly get to a single big file that contains all the GPU code!).
- In fact, these variables/symbols can be shared between files. (declared external in .h files), but only by activating the mode of separate compilation



Compilation steps:

```
gcc -c ... X.c
nvcc -dc ... Y.cu
nvcc -dlink ... Y.o -o __gpu_rdc.o -lcudadevrt
gcc -o pgm ... Y.o __gpu_rdc.o X.o ...
```

13

## Variable definitions and CPU/GPU data transfers Data transfers CPU↔GPU

### Variables allocated on GPU at runtime:

```
float *TabCPU;           // Dynamic array on CPU
float *TabGPU;          // Dynamic array on GPU
cudaError_t cudaStat;  // Result of op on dynamic CUDA vars.

// Allocation of the dynamic arrays from the CPU
TabCPU = (float *) malloc(N*sizeof(float));
cudaStat = cudaMalloc((void **) &TabGPU, N*sizeof(float));
```

### Copying dynamic variables from CPU to GPU:

```
// Copy TabCPU dynamic array into TabGPU dynamic array
cudaStat = cudaMemcpy(TabGPU, TabCPU, sizeof(float)*N,
                      cudaMemcpyHostToDevice);
```

### Copying dynamic variables from GPU to CPU:

```
// Copy TabGPU dynamic array into TabCPU dynamic array
cudaStat = cudaMemcpy(&TabCPU[N/2], &TabGPU[N/2],
                      sizeof(float)*N/2, cudaMemcpyDeviceToHost);
```

### Release of dynamic allocations:

```
free(TabCPU);
cudaStat = cudaFree(TabGPU);
```

14

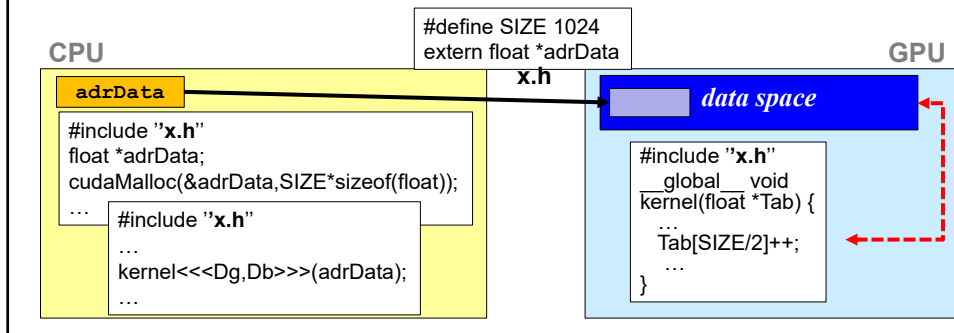
## Specificities of dynamic variables

### Dynamic variables on the GPU

Most are allocated and released by the CPU:

- their pointers are stored on the CPU: the CPU maps the GPU memory !
- their pointers must be passed as parameters when calling GPU kernels

These variables can be easily shared between several files  
(variables declared "extern" in .h files)



15

## CUDA basics

1. Principle of execution of a CUDA program
2. Variable definitions and CPU/GPU data transfers
- 3. Definition of a grid of blocks (of threads)**
4. Definition of a 1<sup>st</sup> CUDA kernel
5. Execution of a CUDA kernel
6. Compiling a CUDA application

16



## Limits on grid and block sizes

### CUDA datatype for grid and block dimensions:

- **dim3** is a structured datatype of 3 int (**\_x**, **\_y** and **\_z**)
- For grid and block descriptors (**Dg** and **Db**)

### Limits of 3D block size:

- $Db.x \leq 1024, Db.y \leq 1024, Db.z \leq 64$
- Total nb of threads / bloc  $\leq 1024$
- $Db.z = 1 \rightarrow 2D$   
 $Db.z = Db.y = 1 \rightarrow 1D$

### Limits of 3D grid size:

- $Dg.x \leq 2^{31} - 1$   
 $Dg.y \leq 65535, Dg.z \leq 65535$
- $Dg.z = 1 \rightarrow 2D$   
 $Dg.z = Dg.y = 1 \rightarrow 1D$

```
// GPU thread management
dim3 Dg, Db;
// Block of 128x4 threads
Db.x = 128;
Db.y = 4;
Db.z = 1;
// Grid of 512 blocks
Dg.x = 512;
Dg.y = 1;
Dg.z = 1;
// → 262144 threads!
```

Launching a kernel from its block grid (see further):

```
// Classical call from a CPU routine
MyKernel<<< Dg, Db >>>(parameters);
```

17

## Block grid without overflow

### Basic example of a 1D grid, with 1D blocks:

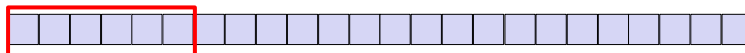
Tab[N] 

For each box « i »:  
 $Tab[i] = Tab[i] * 2.0$

Strategy:

- 1 GPU thread will process 1 box
- 1D grid of 1D blocks of threads will process the array

### 1 - Definition of the blocks (1D)



```
Db.x = BLOCK_SIZE_X; // = 32; 64; 128; 256; 512; 1024
Db.y = BLOCK_SIZE_Y; // = 1
Db.z = BLOCK_SIZE_Z; // = 1
```

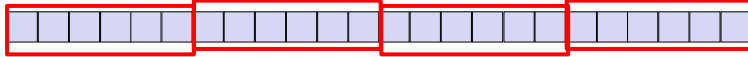
18

## Block grid without overflow


### Basic example of a 1D grid, with 1D blocks:

2 - Definition of the grid, when:  $(N \% Db.x) = 0$

We "pave" the data with blocks.



```
Dg.x = N/Db.x; // Dg.x = N/BLOCK_SIZE_X
Dg.y = 1;
Dg.z = 1;
```

What if  $(N \% Db.x) \neq 0$  ? 

19

## Block grid **with** overflow

### Basic example of a 1D grid, with 1D blocks:

2 - Definition of the grid, when:  $(N \% Db.x) \neq 0$  :



We "pave" the data with whole blocks, even if it overflows!

```
if (N%BLOCK_SIZE_X == 0)
    Dg.x = N/BLOCK_SIZE_X;
else
    Dg.x = N/BLOCK_SIZE_X + 1;
Dg.y = 1;
Dg.z = 1;
```

Warning : we're going to create too many threads

→ we'll have to take this into account in the code...

20

## Block grid **with** overflow

### Basic example of a 1D grid, with 1D blocks:

2 - Definition of the grid, when:  $(N \% Db.x) \neq 0$  :



We "pave" the data with whole blocks, even if it overflows!

```
Dg.x = N/BLOCK_SIZE_X + (N%BLOCK_SIZE_X ? 1 : 0) ;
Dg.y = 1 ;
Dg.z = 1 ;
```

Warning : we're going to create too many threads  
→ we'll have to take this into account in the code...

21

## Block grid **with** overflow

### Basic example of a 1D grid, with 1D blocks:

2 - Definition of the grid, when:  $(N \% Db.x) \neq 0$  :



We "pave" the data with whole blocks, even if it overflows!

```
Dg.x = (N-1)/BLOCK_SIZE_X + 1 ;
Dg.y = 1 ;
Dg.z = 1 ;
```

Warning : we're going to create too many threads  
→ we'll have to take this into account in the code...

22

## Create many small threads

### Masking of GPU memory access times :

- a GPU switches from one thread warp to another very quickly
  - a GPU masks the latency of its memory accesses by multi-threading
- Do not hesitate to create large numbers of small GPU threads

Ex.: to process an array of  $N$  elements:

- Threads dealing with ONE element each
- and a Grid of blocks of  $N$  threads in total



or:

- Threads dealing with  $n$  elements each
- and a Grid of blocks of  $N/n$  threads in total



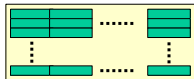
23

## Granularity of the grid and blocks

### How many threads/block and blocks/grid to create ?

The **thread scheduler of each block** likes to have many warps ready to be activated, in order to hide the memory access times.

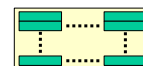
→ "big blocks of threads"



The **block scheduler** likes to have several blocks ready to be loaded at the same time in each SM (*resident blocks*), in order to hide memory access times.

But a SM can only load a new block when it still has enough registers and other resources.

→ "lots of not too big blocks"



So, is it better to make a grid with few big blocks or many small blocks?

24

# Granularity of the grid and blocks

How many threads/block and blocks/grid to create ?

Several possible strategies:

- Calculate the size of the blocks leading to the maximum occupation of the GPU resources...
- Make medium size blocks (128/256 threads) to enable both schedulers to optimize execution
- Experimenting with various block sizes: 32/64/.../512/1024 threads

→ Optimal block grid granularity can be tedious to identify (depends on the CUDA code and GPU model)

But it remains an important source of performance improvement

→ Labs

25

# Granularity of the grid and blocks

Source : <https://en.wikipedia.org/wiki/CUDA>

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		
Maximum number of resident grids per device (concurrent kernel execution)	t.b.d.			16		4		32				16		128		32		16	128
Maximum dimensionality of grid of thread blocks	2			3															
Maximum x-dimension of a grid of thread blocks	65535			2 <sup>31</sup> - 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				32				16			
Maximum number of resident warps per multiprocessor	24	32		48	64											32			
Maximum number of resident threads per multiprocessor	768	1024		1536	2048												1024		
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													

Some characteristics are very stable, others less so...

26

# CUDA basics

1. Principle of execution of a CUDA program
2. Variable definitions and CPU/GPU data transfers
3. Definition of a grid of blocks (of threads)
- 4. Definition of a 1<sup>st</sup> CUDA kernel**
5. Execution of a CUDA kernel
6. Compiling a CUDA application

27

Définition et exécution d'un 1<sup>er</sup> *kernel*

## « Qualifiers » de CUDA

Fonctionnement des « qualifiers » de CUDA :

	<u>__device__</u>	<u>__constant__</u>	<u>__shared__</u>
Variables	Mémoire globale GPU	Mémoire constante GPU	Mémoire partagée d'un multiprocesseur
	Durée de vie de l'application	Durée de vie de l'application	Durée de vie du <i>block de threads</i>
	Accessible par les codes GPU et CPU	Ecrit par code CPU, lu par code GPU	Accessible par le code GPU, sert à <i>cache</i> la mémoire globale GPU
Fonctions	<u>__device__</u>	<u>__host__</u> (default)	<u>__global__</u>
	Appel sur GPU Exec sur GPU	Appel sur CPU Exec sur CPU	Appel sur CPU Exec sur GPU

→ Les « qualifiers » différencient les parties de code GPU et CPU.

28

# 1<sup>er</sup> Kernel (traitant 1 donnée par thread)

## Kernel utilisant la mémoire globale et des registres

Une barre de threads par bloc, et une barre de blocs par grille (un choix).

Un thread traite *une seule* donnée.

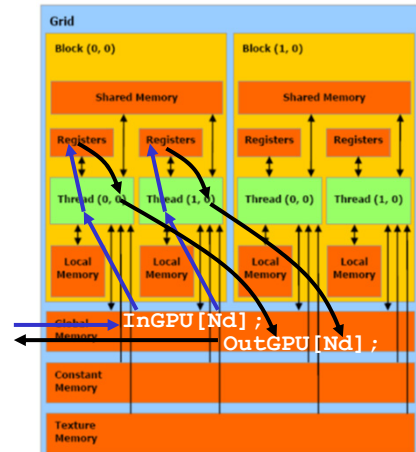
Hyp :  $Nd = k \cdot \text{BLOCK\_SIZE\_X}$

$D_b = \{\text{BLOCK\_SIZE\_X}, 1, 1\}$   
 $D_g = \{Nd/\text{BLOCK\_SIZE\_X}, 1, 1\}$

```

__global__ void k1(void)
{
    int idx;          // Registers:
    float data;      // // 16 Kreg per
    float res;       // // multipro.

    // Compute data idx of the thread
    idx = threadIdx.x +
        blockIdx.x * BLOCK_SIZE_X;
    // Read data from the global mem
    data = InGPU[idx];
    // Compute result
    res = (data + 1.0f) * data ...;
    // Write result in the global mem
    OutGPU[idx] = res;
}
    
```



29

# 1<sup>er</sup> Kernel (traitant 1 donnée par thread)

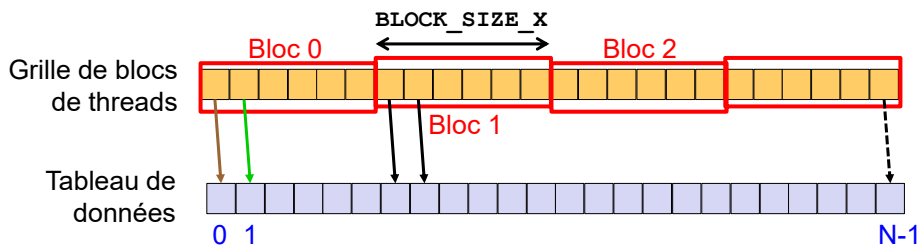
## Calcul de l'indice de la donnée traitée par chaque thread

```

// Compute data idx of the thread
idx = blockIdx.x * BLOCK_SIZE_X
    + threadIdx.x;
// Read data from the global mem
data = InGPU[idx];
    
```

2 variables implicites et propres à chaque thread :

**dim3 threadIdx**  
**dim3 blockIdx**



$$\text{idx} = \text{blockIdx.x} * \text{BLOCK\_SIZE\_X} + \text{threadIdx.x}$$

Indexage permettant des accès coalescents

30

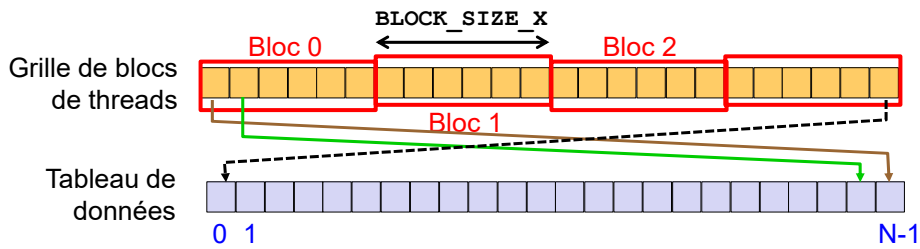
# 1<sup>er</sup> Kernel (traitant 1 donnée par thread)

## Calcul de l'indice de la donnée traitée par chaque thread

```
// Compute data idx of the thread
idx = (N-1) - (threadIdx.x +
              blockDim.x*BLOCK_SIZE_X);
// Read data from the global mem
data = InGPU[idx];
```

2 variables implicites et propres à chaque thread :

```
dim3 threadIdx
dim3 blockDim
```



$$idx = (N-1) - (blockIdx.x * BLOCK\_SIZE\_X + threadIdx.x)$$

Mais ce serait un indexage *moins coalescent*... !!

31

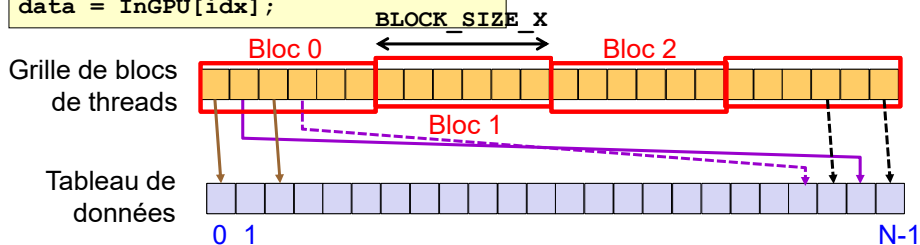
# 1<sup>er</sup> Kernel (traitant 1 donnée par thread)

## Calcul de l'indice de la donnée traitée par chaque thread

```
// Compute data idx of the thread
idx = threadIdx.x +
      blockDim.x*BLOCK_SIZE_X;
if (idx % 2 == 1)
  idx = (N-1) - idx;
// Read data from the global mem
data = InGPU[idx];
```

2 variables implicites et propres à chaque thread :

```
dim3 threadIdx
dim3 blockDim
```



$$idx = (N-1) - (blockIdx.x * BLOCK\_SIZE\_X + threadIdx.x)$$

Mais ce serait un indexage *NON coalescent*... !!

32



# 1<sup>er</sup> *Kernel* (traitant 1 donnée par *thread*)

## Kernel utilisant la mémoire globale et des registres

Une barre de threads par bloc, et une barre de blocs par grille (un choix).  
Un thread traite **une seule** donnée.

Hyp :  $N_d \neq k \cdot \text{BLOCK\_SIZE\_X}$

```

global__ void k1(void)
{
    int idx;          // Registers:
    float data;      // 16Kreg per
    float res;       // multipro.

    // Compute data idx of the thread
    idx = threadIdx.x +
          blockIdx.x * BLOCK_SIZE_X;

    // If the elt indexed exists:
    if (idx < Nd) {
        // Read data from the global mem
        data = InGPU[idx];
        // Compute result
        res = (data + 1.0f) * data ...;
        // Write result in the global mem
        OutGPU[idx] = res;
    }
}
    
```

```

Db = {BLOCK_SIZE_X, 1, 1}
if (Nd % BLOCK_SIZE_X == 0)
    Dg = {Nd / BLOCK_SIZE_X, 1, 1}
else
    Dg = {Nd / BLOCK_SIZE_X + 1, 1, 1}
    
```

### Pavage classique

#### Protection classique :

Les threads « en trop »  
ne font rien...

33

## CUDA basics

1. Principle of execution of a CUDA program
2. Variable definitions and CPU/GPU data transfers
3. Definition of a grid of blocks (of threads)
4. Definition of a 1<sup>st</sup> CUDA kernel
- 5. Execution of a CUDA kernel**
6. Compiling a CUDA application

34

## Execution of a CUDA kernel

Exécution depuis le CPU d'une grille de threads sur le GPU :

```
// Usually only 2 arguments are specified:  
Kernel<<< Dg, Db >>>(parameter, ..., ... ..);
```

↳ Descripteur d'un *bloc 3D* de *threads*

↳ Descripteur d'une *grille 3D* de *blocs 3D* de *threads*

Version complète

```
// Complete syntax  
Kernel<<< Dg, Db, Ns, S >>>(parameter, ..., ... ..);
```

Voir plus loin

↳ Numéro du *stream* d'interaction CPU/GPU (le 0 par défaut)... utile pour du recouvrement transferts/calculs en utilisant plusieurs *streams*

↳ Allocation dynamique de *shared memory* au lancement de chaque bloc sur un SM... (0 octets par défaut)

35

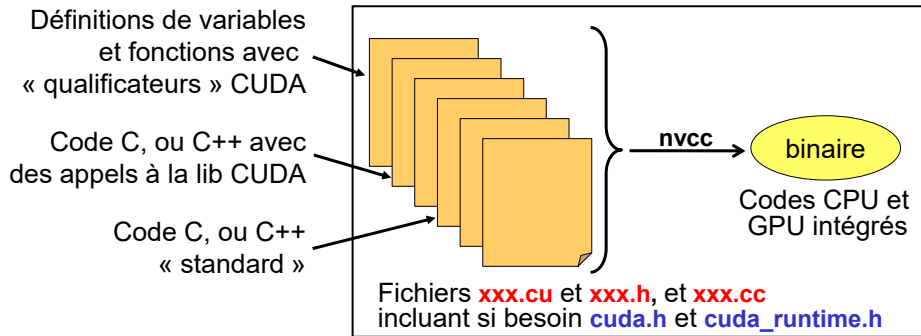
## CUDA basics

1. Principle of execution of a CUDA program
2. Variable definitions and CPU/GPU data transfers
3. Definition of a grid of blocks (of threads)
4. Definition of a 1<sup>st</sup> CUDA kernel
5. Execution of a CUDA kernel
- 6. Compiling a CUDA application**

36

# Compilation 100% CUDA

Compilation d'applications CUDA – entièrement développées en CUDA :



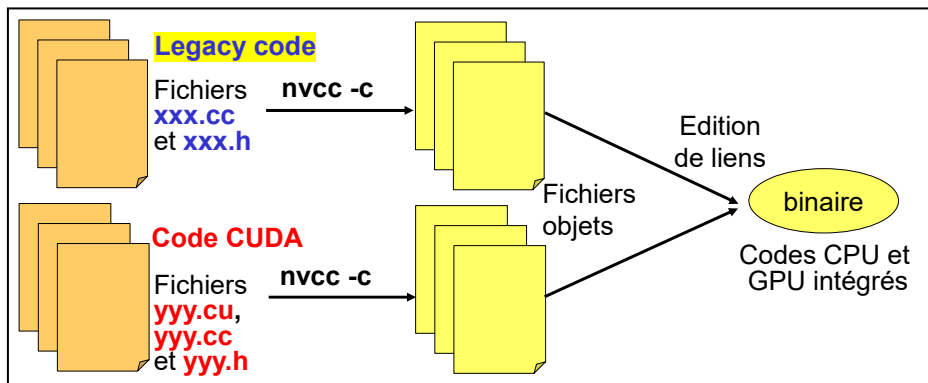
Pour les codes C/C++ simples :

- Il est possible de simplement tout recompiler en `nvcc` dans des fichiers `xxx.cu` (et `xxx.cc` incluant `cuda.h` et `cuda_runtime.h`)
- Mais les optimisations sérielles peuvent en souffrir...

37

# Compilation mixte CUDA & C++

Compilation d'applications mixte CUDA & C++ avec 1 compilateur :

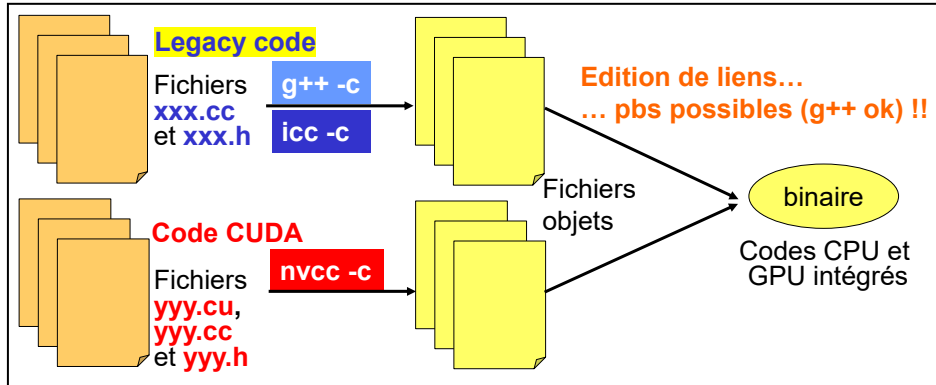


- Code C++ métier : fichiers `.cc` compilés en `nvcc`
- Code CUDA : fichiers `.cu` compilés en `nvcc`
- Edition de liens : des pbs peuvent encore apparaître avec des templates ...

38

# Compilation mixte CUDA & C++

Compilation d'applications mixte CUDA & C++ avec 2 compilateurs :



- Code C++ métier : fichiers `.cc` compilés en `gcc/g++` ou en `icc` ou ...
- Code CUDA : fichiers `.cu` compilés en `nvcc`
- Edition de liens : des pbs de nommage peuvent apparaitre (mais pas en `gcc`)

39

## CUDA basics

End

40