



Institut
Mines-Télécom

GPU for Deep Learning :

Introduction to GPU architecture

A decorative horizontal bar at the bottom of the slide, composed of three segments: a blue segment on the left, a black segment in the middle, and a brown segment on the right.

Elisabeth Brunet



Plan

- Introduction
- GPU Architecture
- CUDA Architecture
- CUDA Programming

GPU ARCHITECTURE

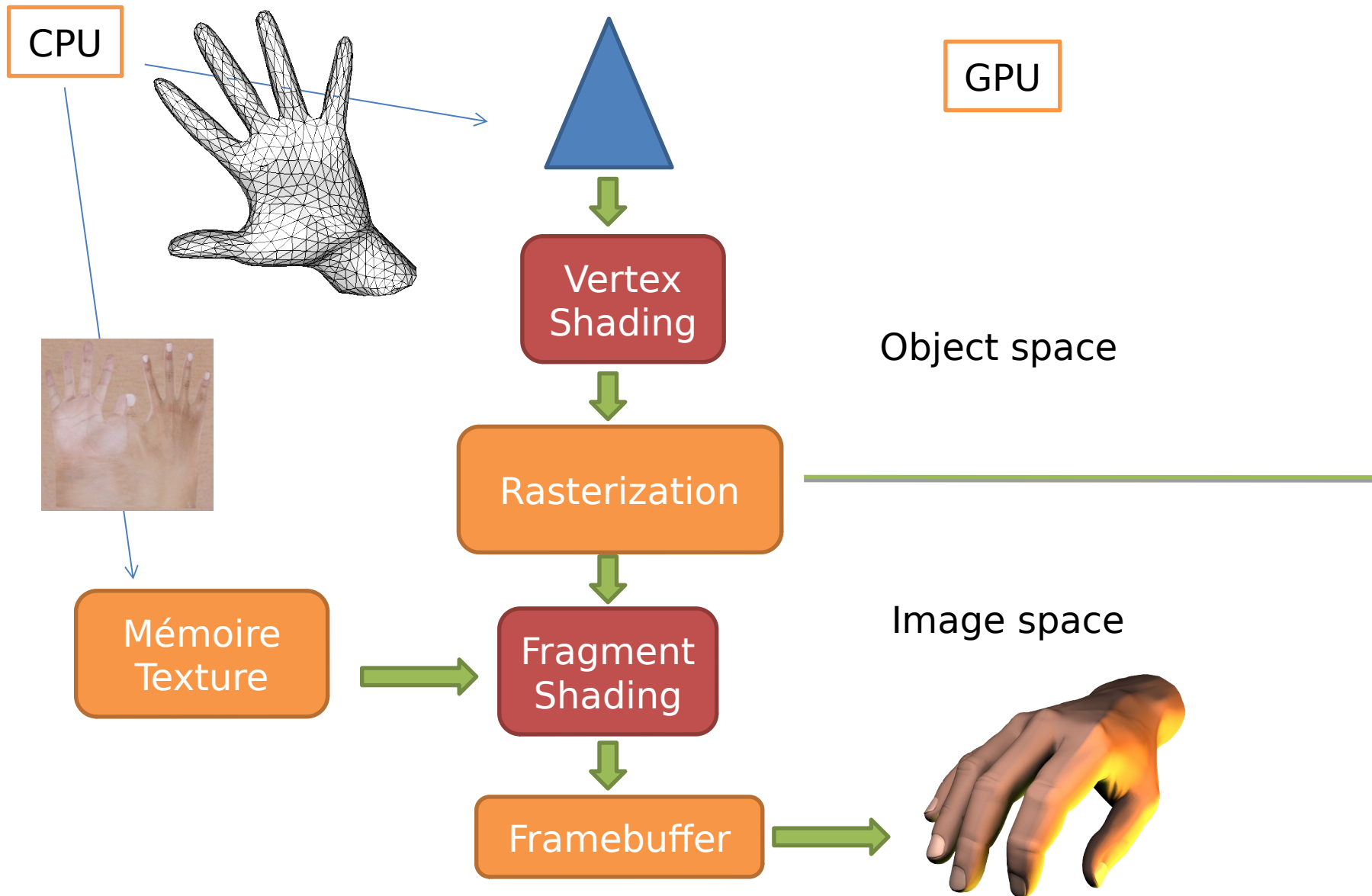


GPU

- Graphics Processor Unit
- Co-processor located on a pci-express slot
- Architecture many-core with its own memory space

- Initially, static graphics pipeline
 - Designed for 3D computation required by image synthesis
 - Driven by the video game market

Graphics Pipeline



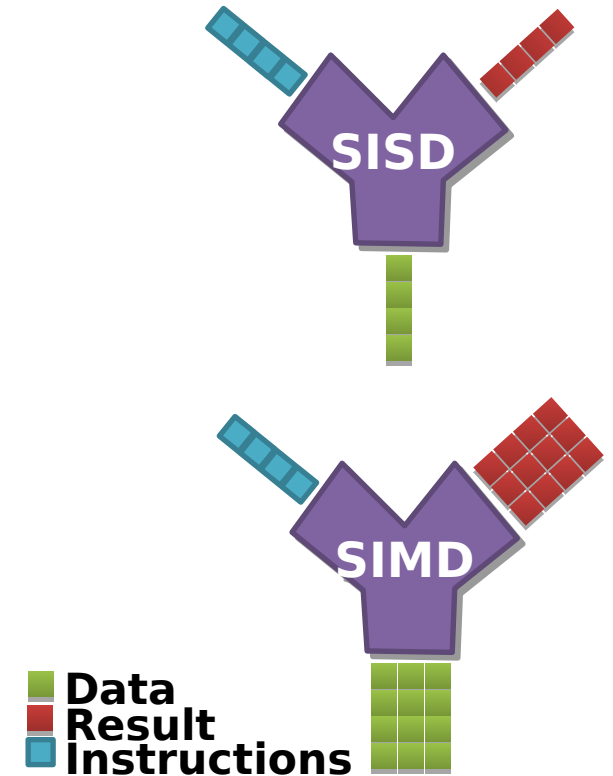
GPU

- Graphics Processor Unit
- Co-processor located on a pci-express slot
- Architecture many-core with its own memory space

- Initially, static graphics pipeline
 - Designed for 3D computation required by image synthesis
 - Driven by the video game market
- Early 2000s, GPGPU (General Processing GPU)
 - Opening of the architecture
 - More general computation types
 - A lot of application domains : image processing, numerical simulations, linear algebra, deep learning, etc.

GPU

- Architecture **SIMD**
 - **Single Instruction / Multiple Data**
 - Data parallelism
 - Ideal for intensive massively parallel computation
- Hundreds of cores
- Cores with limited capabilities
 - No dynamic memory allocation
 - No heap > no recursion
- Memory hierarchy
 - NUMA effects



CPU vs GPU

- CPU

- Architecture to minimize latency

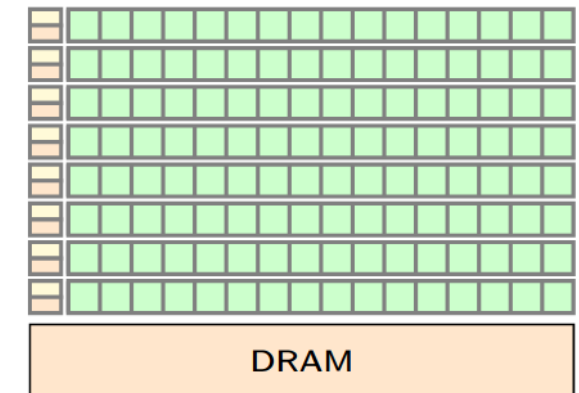
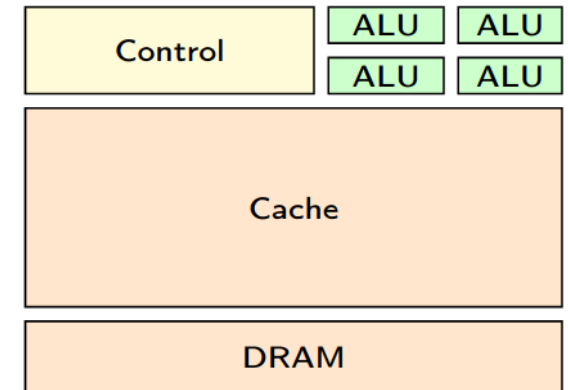
- Demanding operations, e.g. keyboard events
 - Relying on caches
 - Dedicated circuits for out-of-cache operations
 - e.g. pre-fetch, out-of-order execution

- GPU

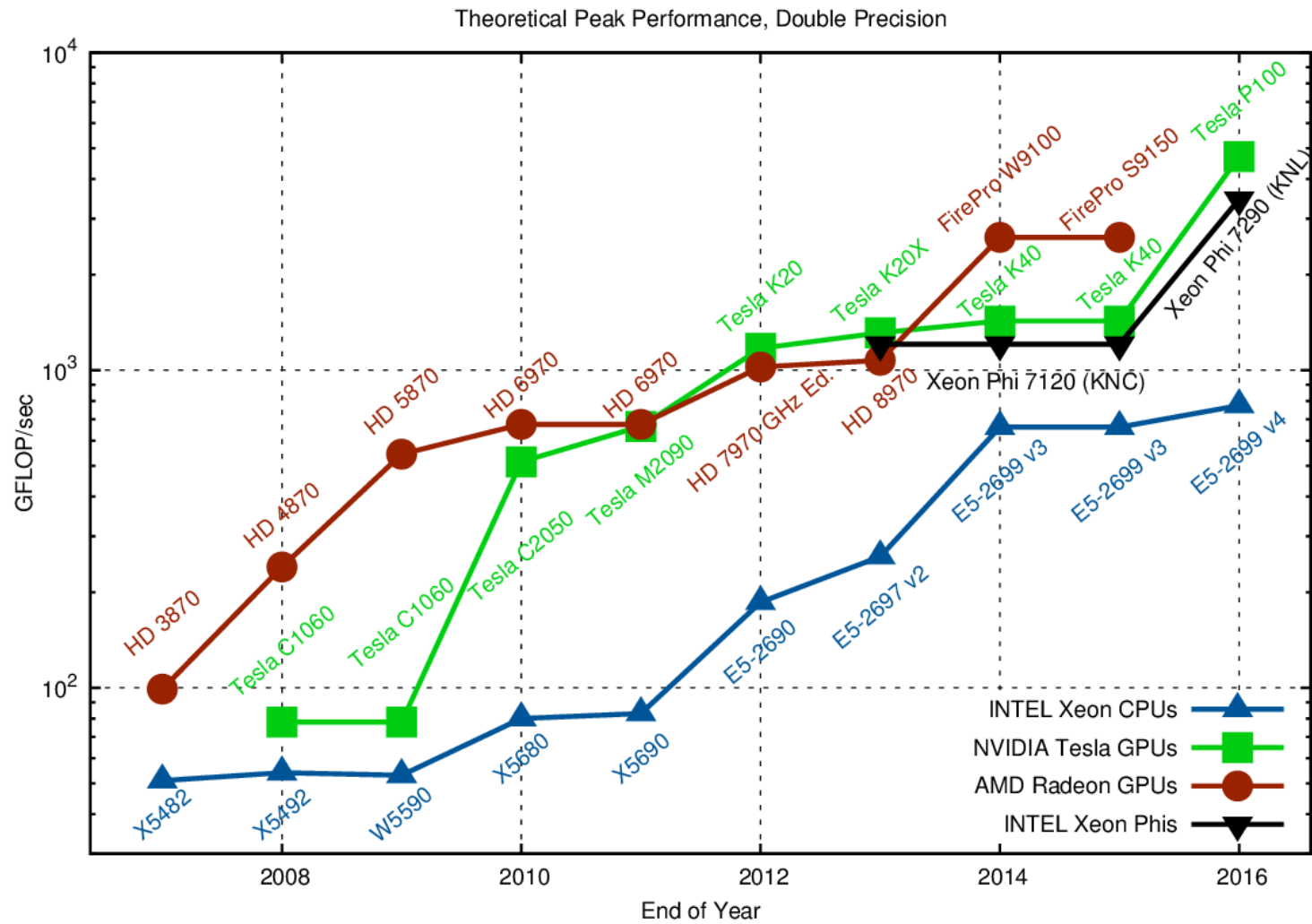
- High latency and high bandwidth processors

- No need for a large cache
 - Transistors dedicated to data processing rather than cache management
 - Chip of the same size but with much more ALU

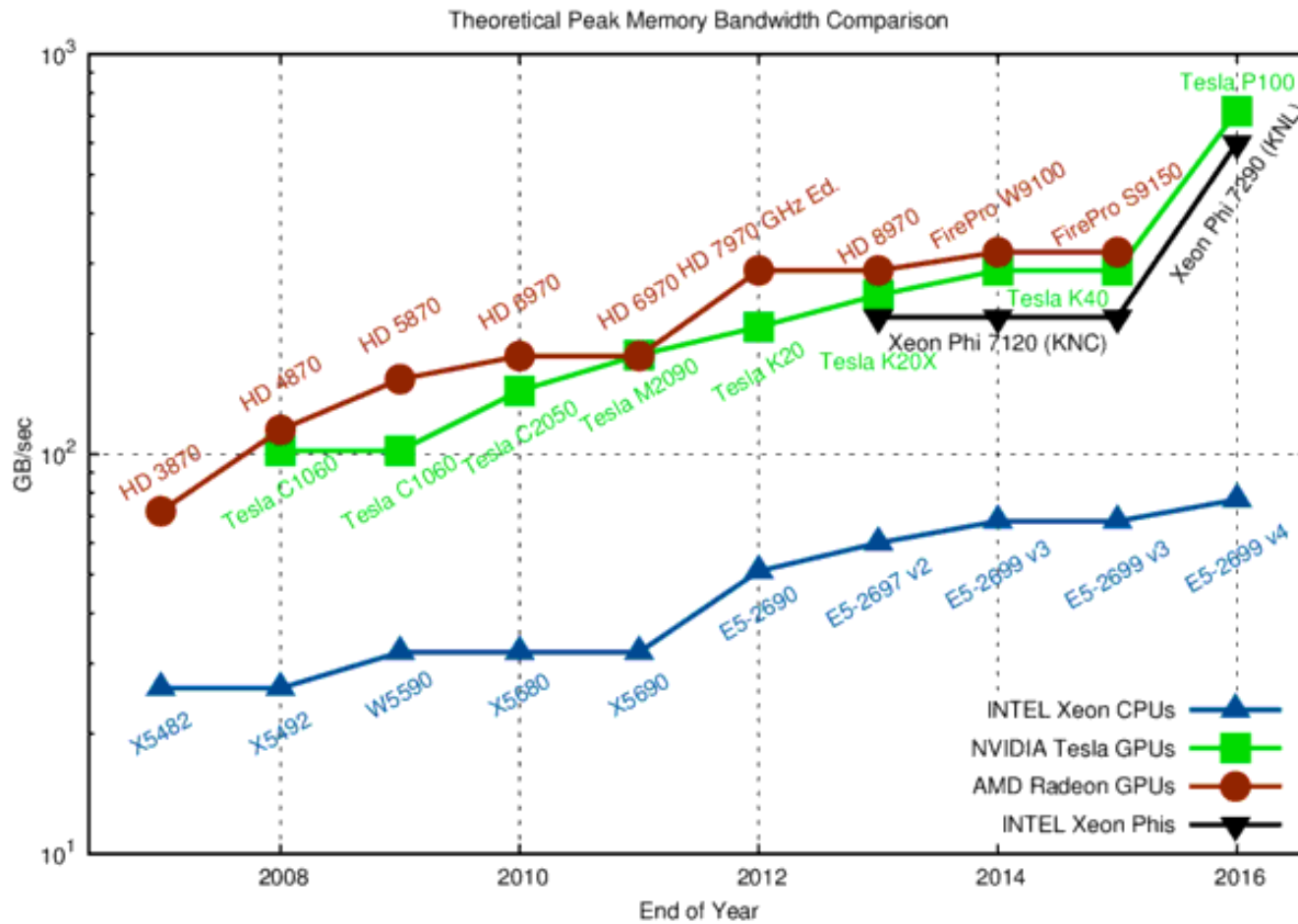
- Latency
 - Delay between the initialization of an operation and when its effects are detectable
 - A car has a lower latency than a bus.
- Bandwidth
 - Amount of work achieved over a given period of time
 - A bus has a higher bandwidth than a car.



CPU vs GPU



CPU vs GPU



Concurrent GPUs

- GeForce/Quadro/Tesla NVIDIA cards
 - Micro-architectures : Fermi, Kepler, Maxwell, Pascal, Volta, Turing, Ampere
 - Calculation-oriented programming : CUDA
- AMD Radeon cards
 - Including ATI's Stream Computing architectures
 - OpenCL : standardization of GPU programming
- Graphic programming : OpenGL, Vulkan, Direct3D, DirectX

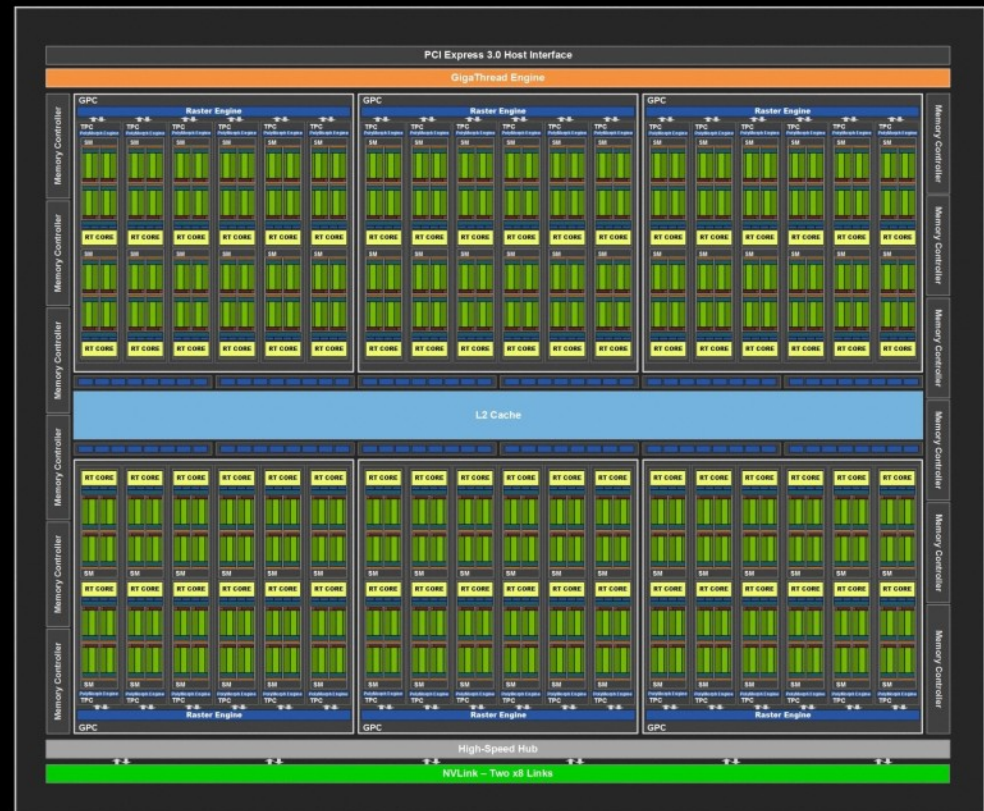
Turing Architecture

INTRODUCING TURING

TU102 – FULL CONFIG

18.6 BILLION TRANSISTORS

SM	72
CUDA CORES	4608
TENSOR CORES	576
RT CORES	72
GEOMETRY UNITS	36
TEXTURE UNITS	288
ROP UNITS	96
MEMORY	384-bit 7 GHz GDDR6
NVLINK CHANNELS	2



THIS PRESENTATION IS EMBARGOED UNTIL SEPT EMBER 14, 2018



TELECOM
SudParis





Tensor cores

- Specialized cores
 - 4x4 matrix cores
 - Ultra fast for operations on very small matrixes
 - 1 matrix multiply-accumulate operation per 1 GPU clock
 - Particularly adapted to the demands of deep learning
 - <https://devblogs.nvidia.com/programming-tensor-cores-cuda-9/>

CUDA ARCHITECTURE



Description

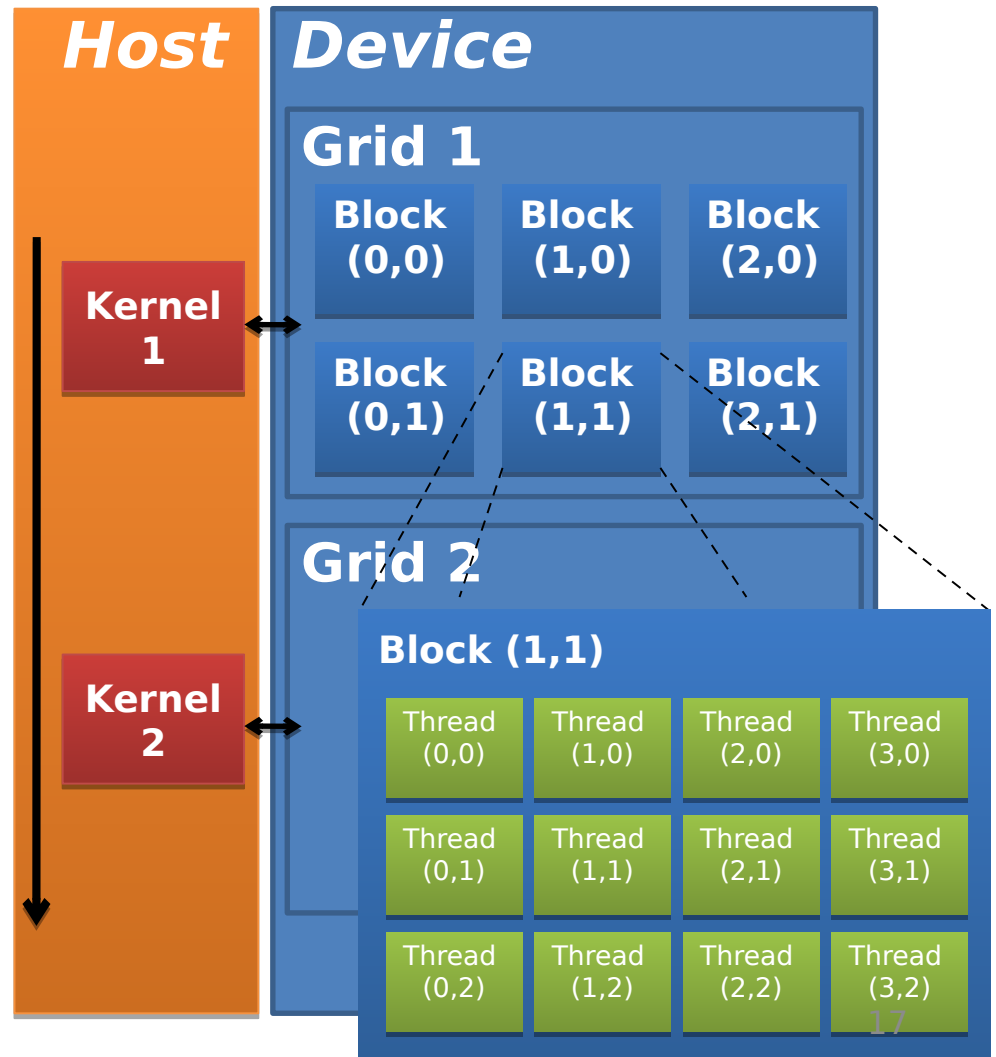
- *Compute Unified Device Architecture*
- Hardware and software architecture of NVidia GPUs
- Programmable in C, C++, Fortran, Python
- Exploits directly the unified architecture (G80 and +)

Principle

- From a host program running on a CPU,
 - Launching a computing **kernel** on the GPU **device** that
 - Executes the same calculation
 - Thanks to many very light threads
 - On different data loaded in the GPU memory
- Since Fermi, several kernels can be launched in parallel
- Since Kepler, launching kernels from a kernel

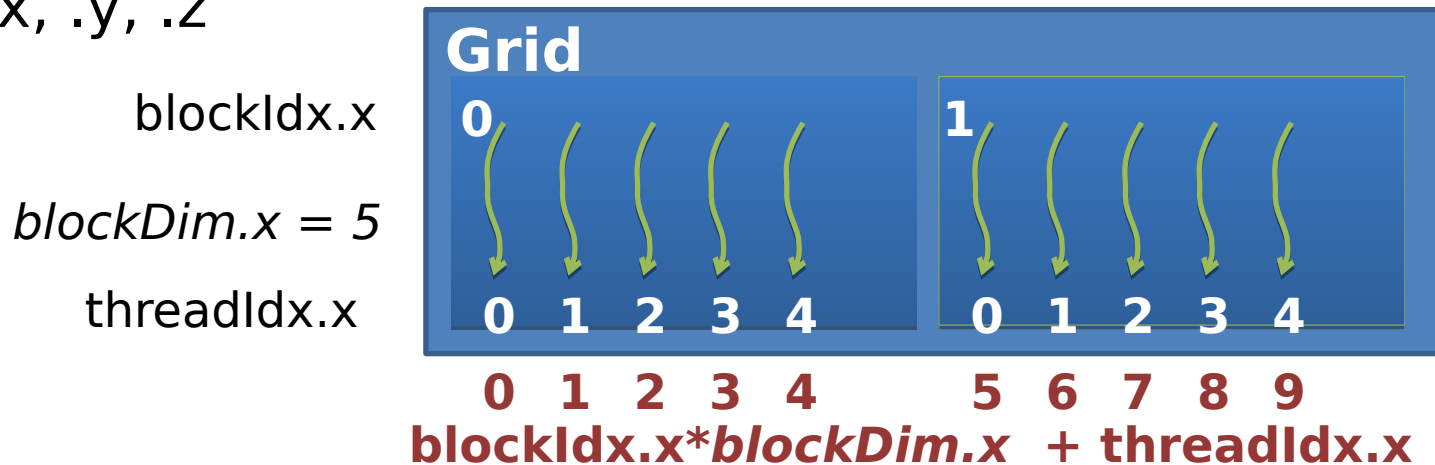
Programming model

- Kernel executed by a grid of thread blocks
 - 3D Grid
 - 3D blocks
- In a block, the threads
 - Cooperate via shared memory
 - Are scheduled by warp
 - *Warp = 32 threads*
 - Threads of a warp are synchronous
- No inter-block cooperation

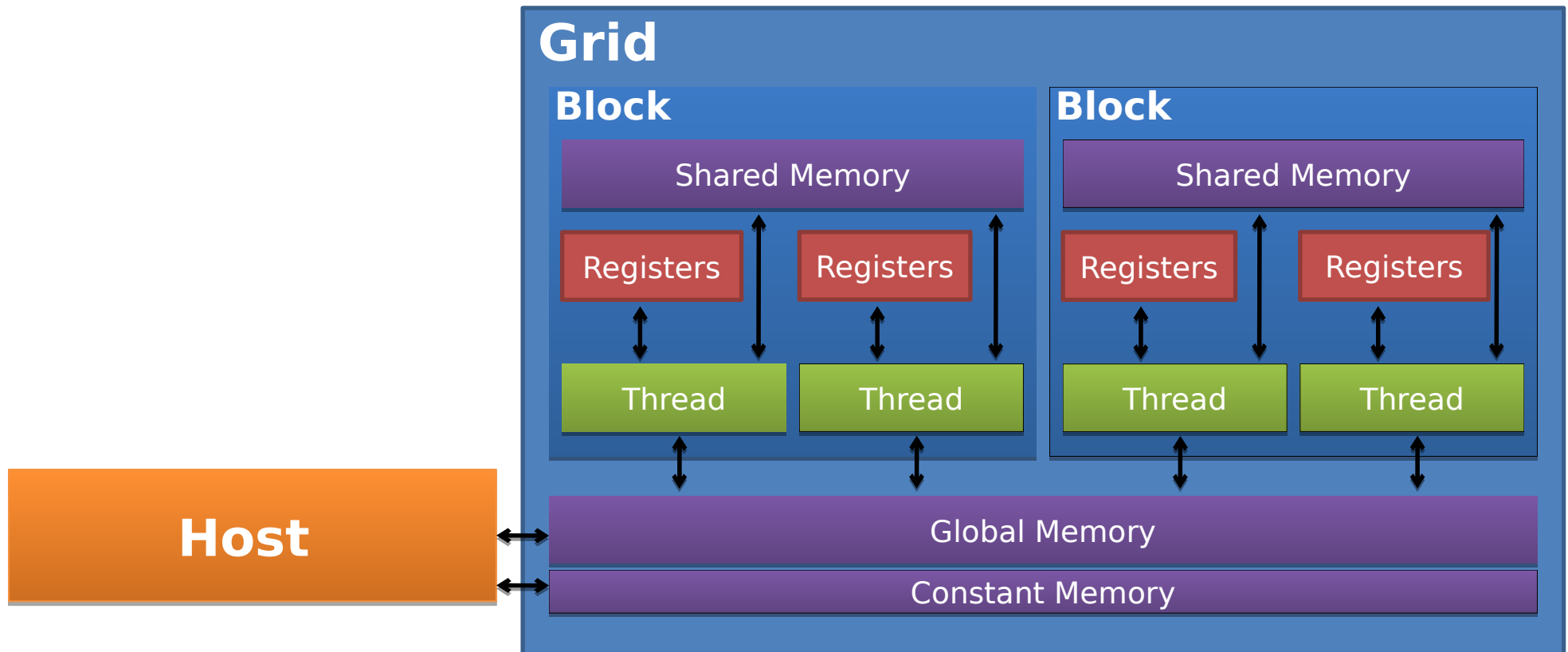


Multi-dimensional identifiers

- Each thread accesses a different part of the data
 - Complies with the data structure
- Thread indexing information
 - `threadIdx.x, .y, .z` : thread index within the block
 - `blockIdx.x, .y, .z` : block index within the grid
- Information about the grid at runtime
 - `blockDim.x, .y, .z`
 - `gridDim.x, .y, .z`



Memory



CUDA C PROGRAMMING

GPU Information

- Computing capacities at runtime in order to tune application
- Number of GPUs available : **cudaGetDeviceCount**
- Characteristics of a GPU : **cudaGetDeviceProperties**
 - Model, memory size, etc.

- Choosing a GPU :
cudaChooseDevice

```
cudaDeviceProp prop ;
int count;

cudaGetDeviceCount(&count) ;

for(int i = 0; i < count ; i++){
    cudaGetDeviceProperties (&prop , i) ;
    printf («[GPU%d] Taille de la mémoire globale%d\n »,
           i, prop.TotalGlobalMem ) ;
}
```

CUDA Kernel

- `__global__ void my_kernel(parameters) {...}`
- A kernel is a C function with some features :
 - Identified using the keyword **__global__**
 - Invoked by the CPU and runs on the GPU
 - Only accesses GPU memory
 - Void return
 - No variable number of arguments
 - No recursion
 - No static variable
 - Kernel arguments are passed by copy
 - Flow instructions (if, while, for, switch, do)
 - Branch serialization within warp → performance loss

Kernel Invocation

- `my_kernel <<<dim3 Grid,dim3 Block>>>(parameters)`
- Maximum number of threads per block :
1024 to be distributed over the 3 dimensions
- Maximum grid size : $2^{31}-1 \times 65535 \times 65535$
- Information related to the specification of the GPU used
- Predefined variables set by invocation
 - `dim3 gridDim` : grid dimensions
 - `dim3 blockDim` : block dimensions
 - `dim3 blockIdx` : block index in the grid
 - `dim3 threadIdx` : thread index in the block

Example : Array incrementation

CPU code

```
void incr_cpu(float *a, float b, int N){
    for (int idx = 0; idx<N; idx++)
        a[idx] = a[idx] + b;
}

void main(){
    .....
    incr_cpu(a, b, N);
}
```

GPU code

```
__global__ void incr_gpu(float *a, float b, int N){
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    if (idx < N)
        a[idx] = a[idx] + b;
}

void main(){
    ...
    dim3 dimBlock (blocksize);
    dim3 dimGrid( ceil( N / (float)blocksize) );
    incr_gpu<<<dimGrid, dimBlock>>>(a, b, N);
}
```


Functions

Qualifier	Effect
__global__	Computation kernel Function invoked since the host and executed on the device
__device__	Function invoked and executed on the device
__host__	(optionnel) Function invoked and executed on the host

- **__host__** and **__device__** can be combined
 - Overloaded operators
 - The compiler generates 2 codes: a CPU version and a GPU version.

Data management (1/3)

- CPU and GPU have physically separate memory spaces
 - Different GPU memories seen in an upcoming course
- Data must be in GPU global memory to be processed
- From host,
 - allocation/free and copy of data
- From device,
 - Static declaration with keyword `__device__`

Data management (2/3)

- Allocation : **cudaMalloc**(void ** pointer, size_t nbytes)
- Desallocation : **cudaFree**(void* p)
- Cleaning : **cudaMemset**(void * p, int val, size_t nbytes)

```
// Allocation of an array of n integers  
int n = 1024;  
int nbytes = n*sizeof(int);  
int *d_tab = NULL;  
cudaMalloc( (void**)&d_a, nbytes );  
cudaMemset( d_a, 0, nbytes);  
  
...  
cudaFree(d_a);
```

Data management (3/3)

- Copy of the data from host :
cudaMemcpy(void *dst, void *src,
 size_t nbytes,
 enum cudaMemcpyKind direction);

with enum **cudaMemcpyKind**

= {*cudaMemcpyHostToDevice*,
cudaMemcpyDeviceToHost,
cudaMemcpyDeviceToDevice}

- Copies after previous CUDA calls are completed
- Blocks the master thread for copy time

Example : Array incrementation

CPU code

```
void incr_cpu(float *a, float b, int N){
  for (int idx = 0; idx<N; idx++)
    a[idx] = a[idx] + b;
}

void main(){
  ...
  float*a=malloc(N*sizeof(float)) ;
  // initialisation de a
  incr_cpu(a, b, N);
}
```

GPU code

```
__global__ void incr_gpu(float *a, float b, int N){
  int idx = blockIdx.x*blockDim.x + threadIdx.x;
  if (idx < N)  a[idx] = a[idx] + b;
}

void main(){
  ...
  float *a=malloc(N*sizeof(float)) ;
  // initialisation de a
  float *d_a = NULL ;
  cudaMalloc( (void**)&d_a, N*sizeof(float) );
  cudaMemcpy(d_a, a,N*sizeof(float),
              cudaMemcpyHostToDevice) ;
  ...
  dim3 dimBlock (blocksize);
  dim3 dimGrid( ceil( N / (float)blocksize) );
  incr_gpu<<<dimGrid, dimBlock>>>(d_a, b, N);
  cudaMemcpy(a, d_a, N*sizeof(float),
              cudaMemcpyDeviceToHost) ;
}
```

Synchronization from host

- Kernels are asynchronous
 - Kernel calls return immediately
 - Kernels run after all previous ones have run
- `cudaMemcpy()` is synchronous
 - Call returns after the copy is made
 - Copying starts after all previous CUDA calls have been executed
- **`cudaThreadSynchronize()`**
 - Blocks until all previous CUDA calls have fully executed



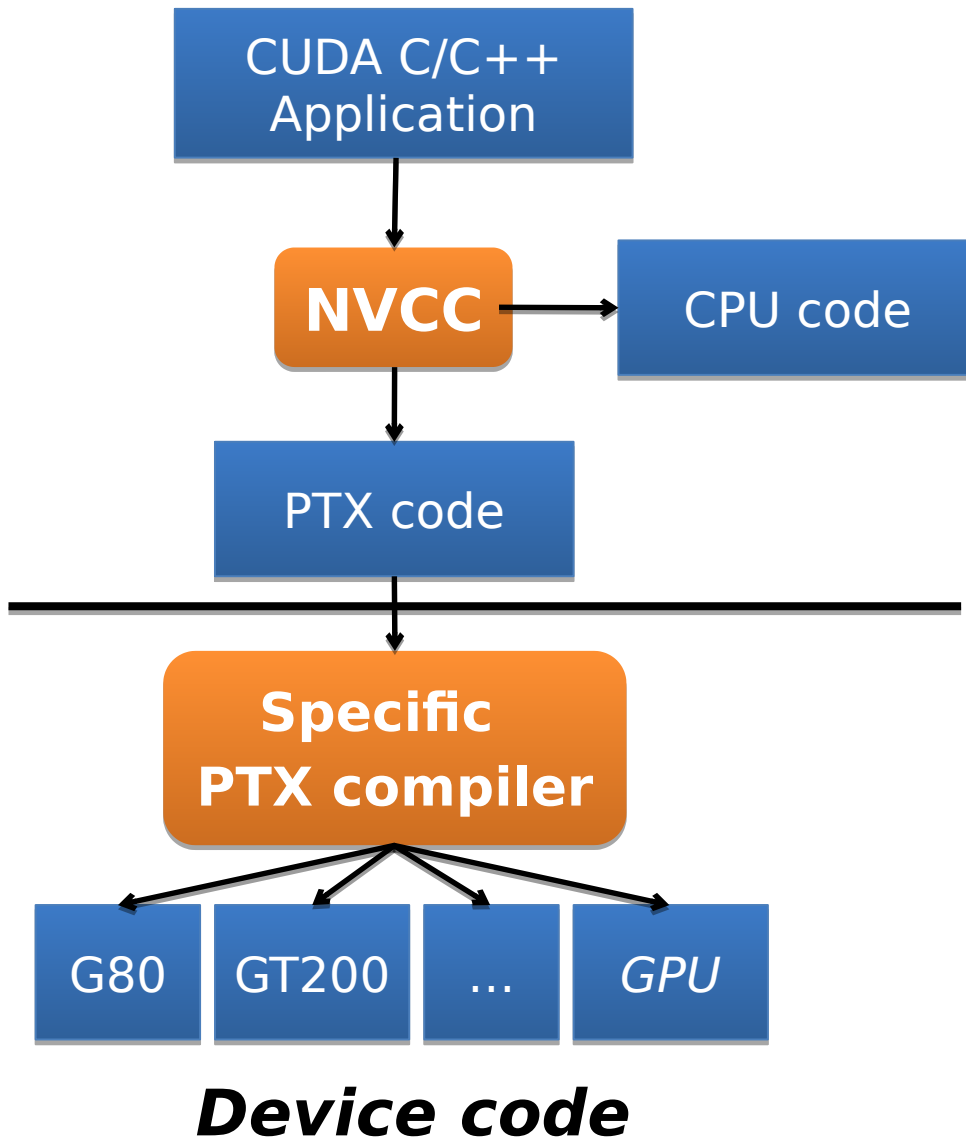
Synchronization on GPU

- `void __syncthreads();`
 - Synchronizes all threads of a block

Race condition

- Atomic operations
 - atomicAdd()
 - atomicSub()
 - atomicMin()
 - atomicMax()
 - atomicInc()
 - atomicDec()
 - atomicExch()
 - atomicCAS()
 - atomicAnd()
 - atomicOr()
 - atomicXor()
- Tools for mutual exclusion (mutex) not natively available
 - Implementation based on shared variables and atomicCAS (= comparison of a variable to a given value)
 - Problem of execution within warps

Compilation



- Meta-compiler `nvcc`
 - CPU and GPU codes
- Binary containing CUDA code requires
 - CUDA core library (`cuda`)
 - CUDA runtime library (`cuda`)
sif needed

Runtime Error Management

- All CUDA calls, except kernels, return an error code
 - **cudaError_t** type
- `cudaError_t cudaGetLastError(void)`
 - Returns the error code of the last call made to CUDA
 - Useful for asynchronous calls
- `char* cudaGetErrorString(cudaError_t code)`
 - Returns a string describing the error
 - `printf ("%s\n", cudaGetErrorString(cudaGetLastError ()));`

Time measurement

- API CUDA event

```
cudaEvent_t start, stop ;  
float milliseconds = 0.0;  
cudaEventCreate(&start) ; cudaEventCreate(&stop) ;  
...  
cudaEventRecord(start) ;  
saxpy <<<(N+255) /256 , 256>>>(N, 2.0 f , d_x ,d_y) ;  
cudaEventRecord(stop) ;  
cudaEventSynchronize(stop) ; // Guarantees that the event has been executed  
cudaEventElapsedTime(&milliseconds, start, stop)
```

- If another timer is used (e.g. `clock_gettime`)
 - `cudaDeviceSynchronize` to wait for the end of the kernel



Let's go to practise now !