



Institut  
Mines-Télécom

# ***Introduction to GPU architecture***



Elisabeth Brunet



# Plan

- Introduction
- GPU Architecture
- CUDA Architecture
- CUDA Programming

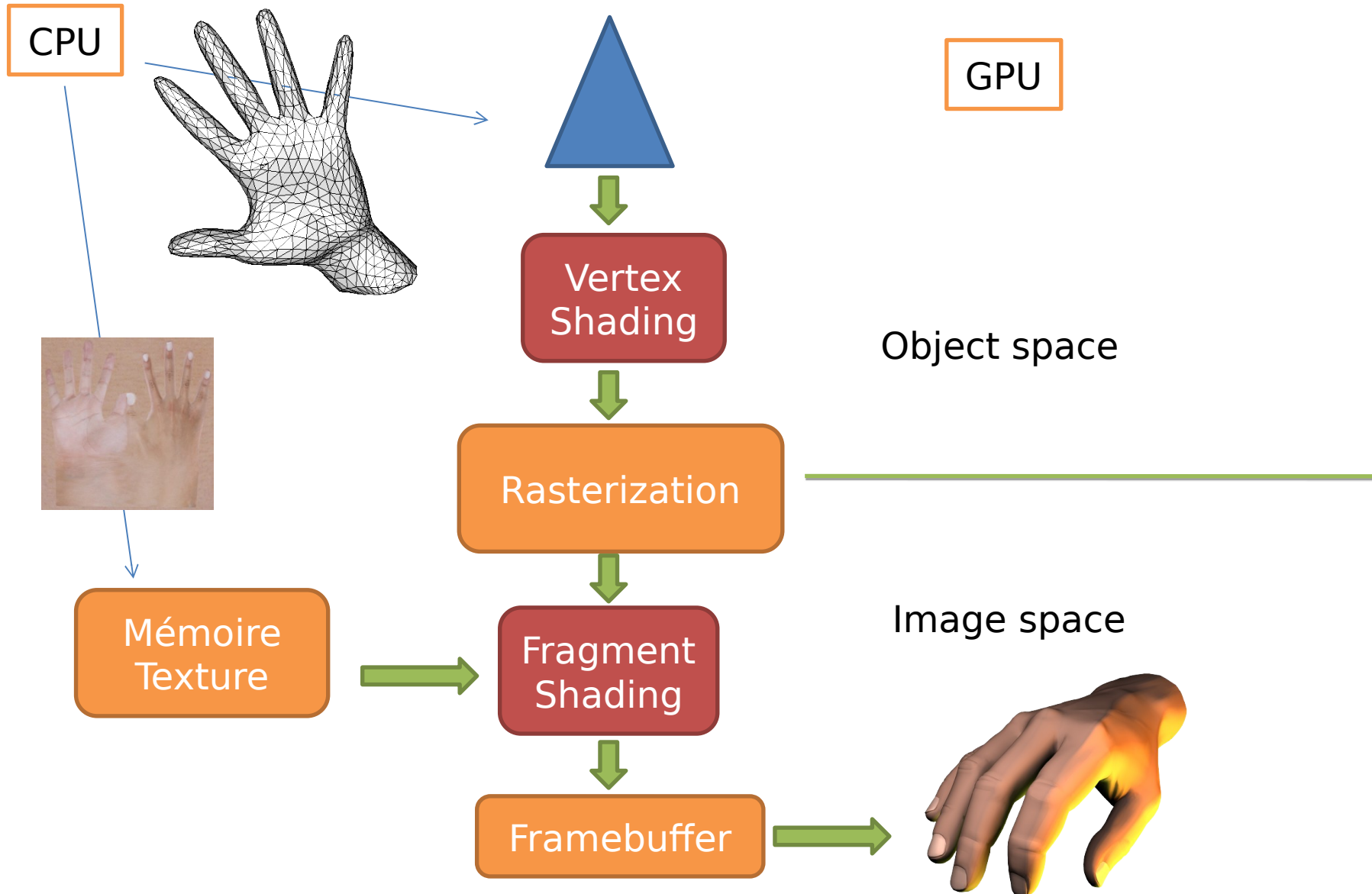
# **GPU ARCHITECTURE**



# 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



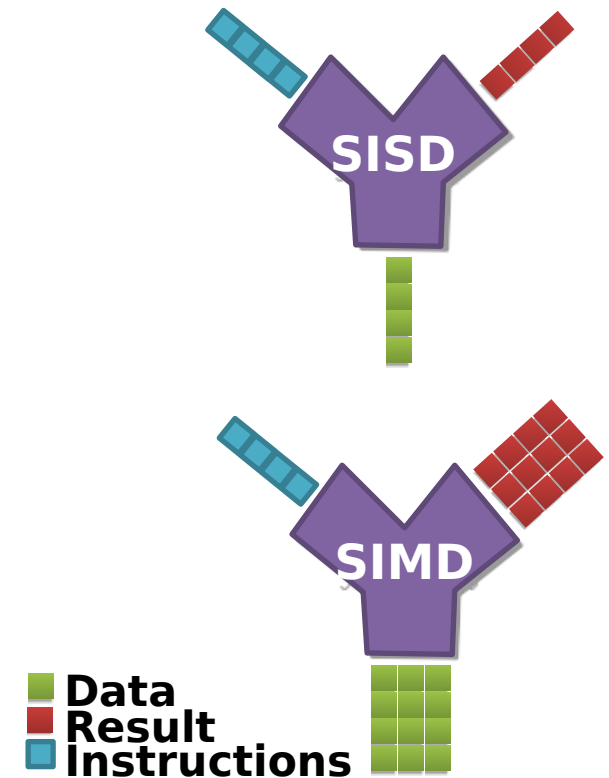


# GPU

- 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

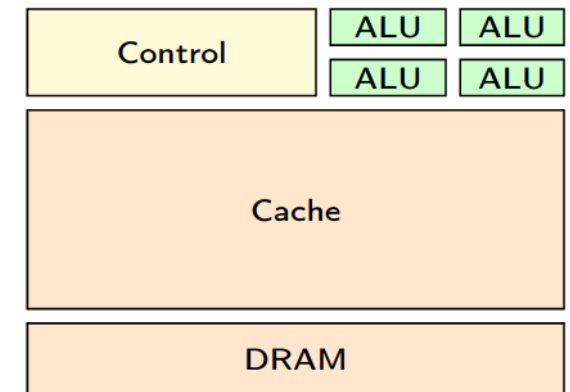


# CPU vs GPU

- 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.

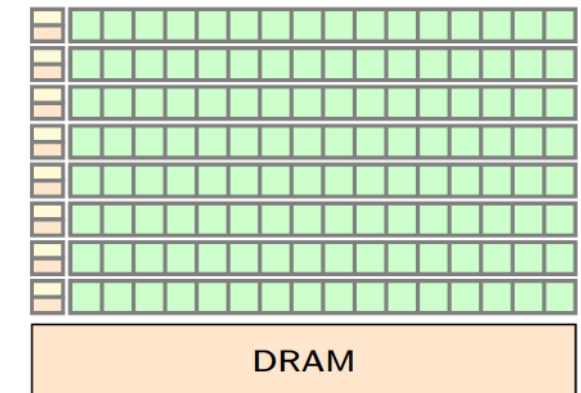
- CPU

- 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



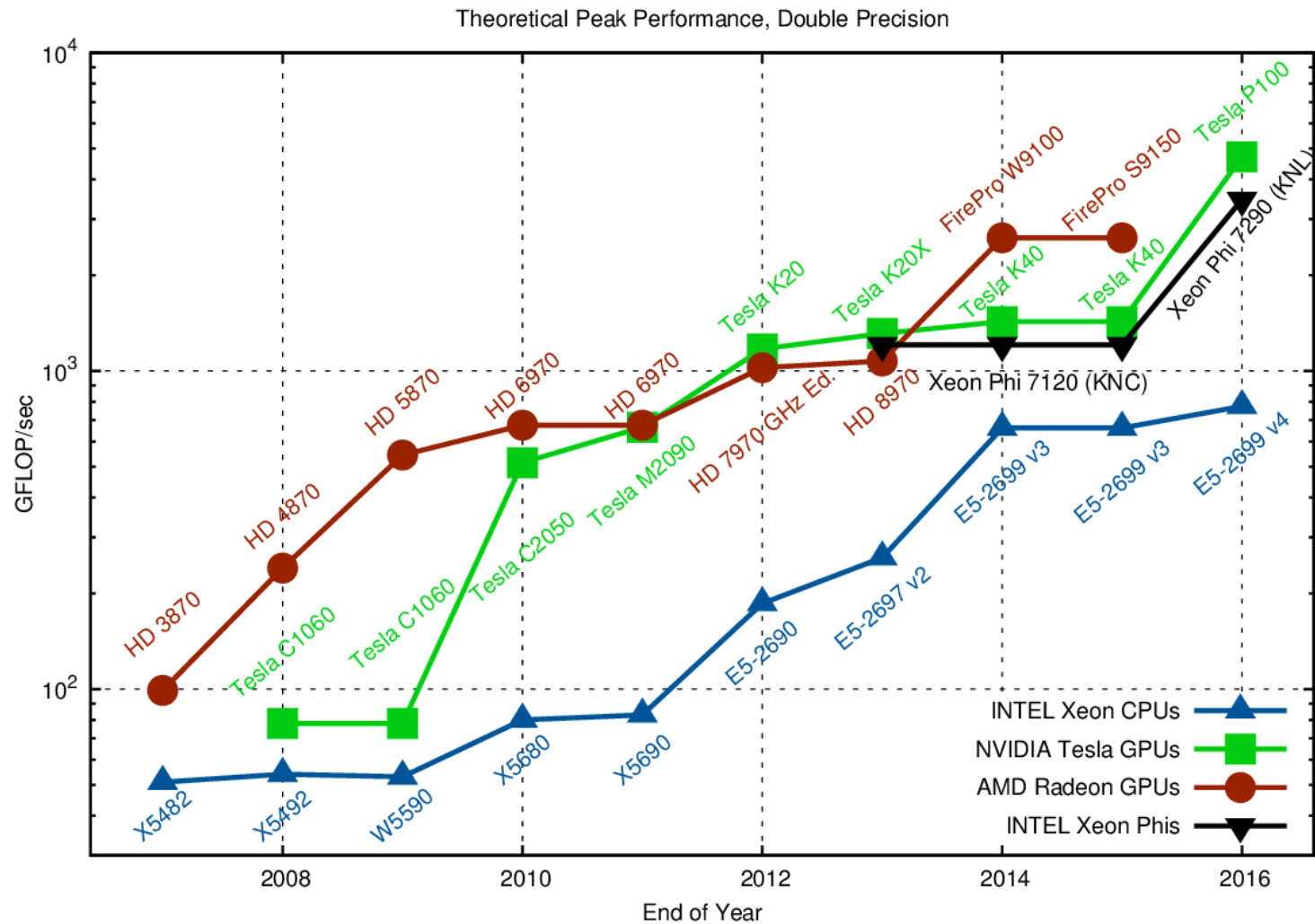
- GPU

- 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

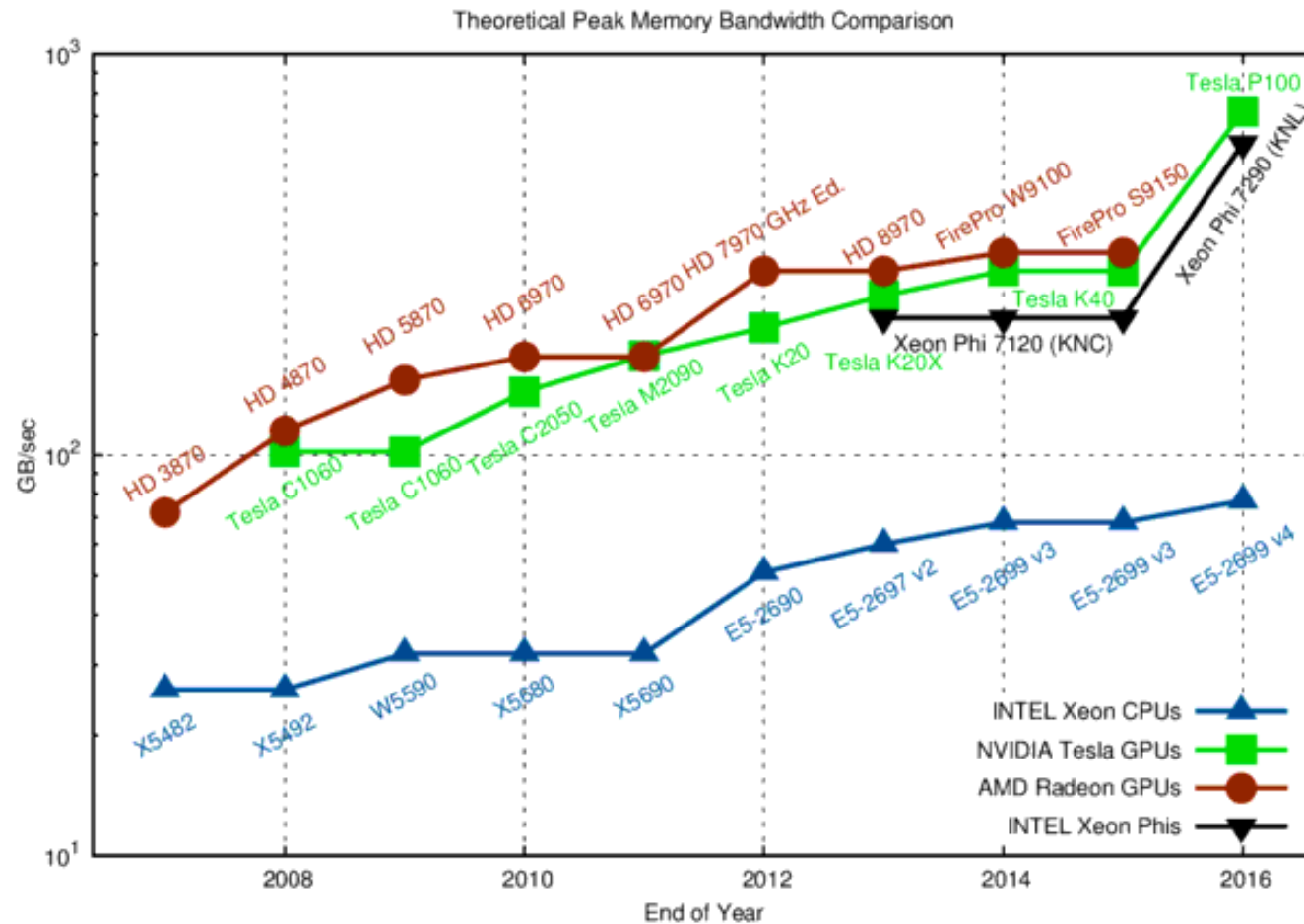




# CPU vs GPU



# 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 (2018)

## 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

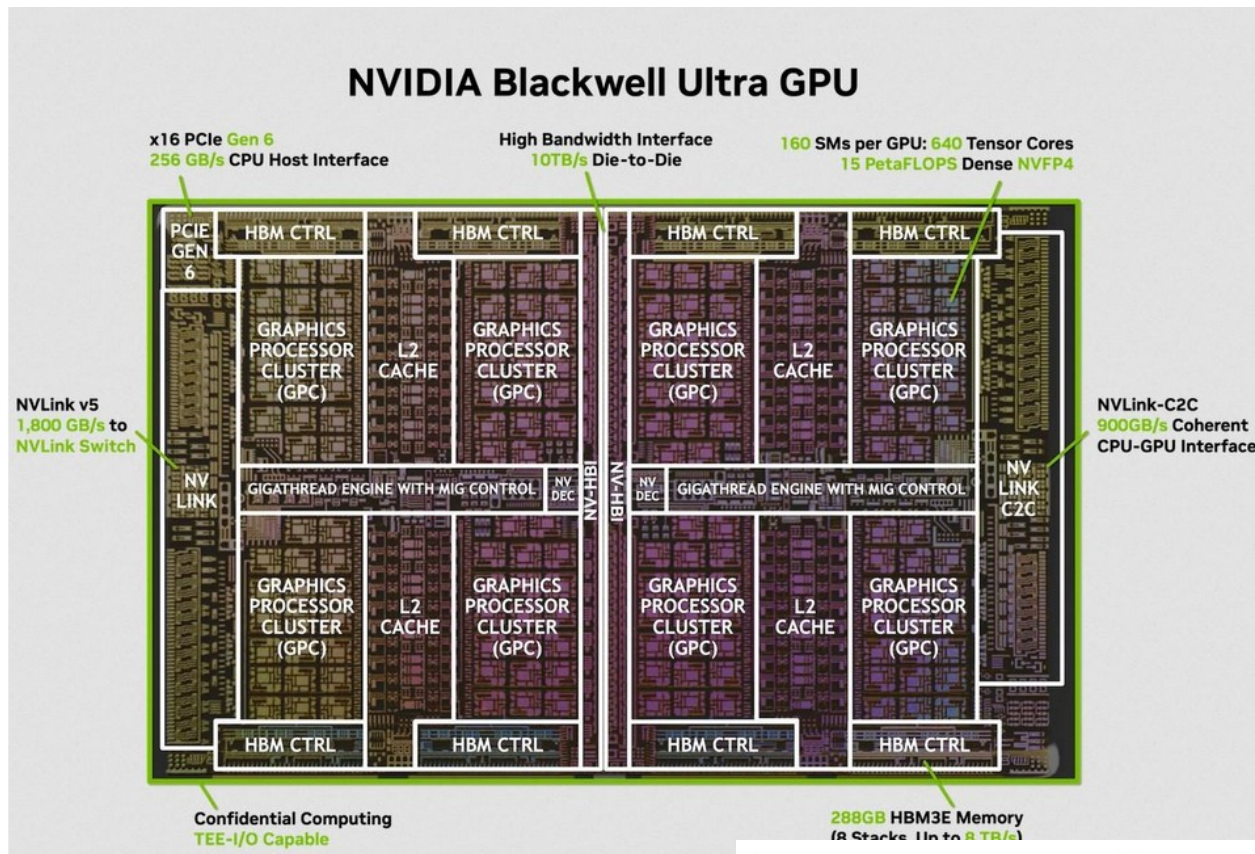


THIS PRESENTATION IS EMBARGOED UNTIL SEPTEMBER 14, 2018





# Blackwell Architecture (2024)



Die		GB100	GB102	GB200
Variant(s)		Unknown	Unknown	Unknown
Release date		Dec 2024	Nov 2024	Unknown
Cores	CUDA Cores	18,432		
	TMUs	576		
	ROPs	24		
	RT Cores	Unknown	Unknown	Unknown
	Tensor Cores	576		



# 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 +)



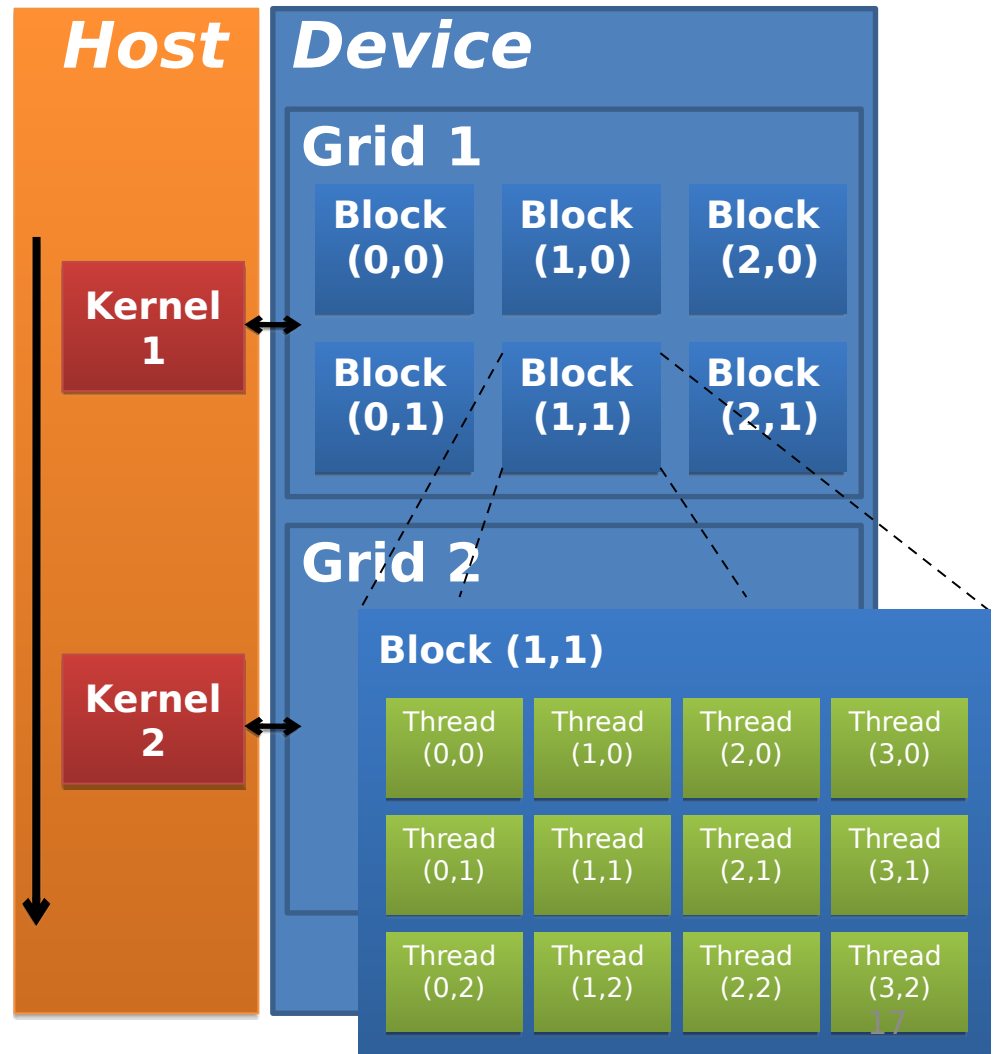


# Principle

- 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

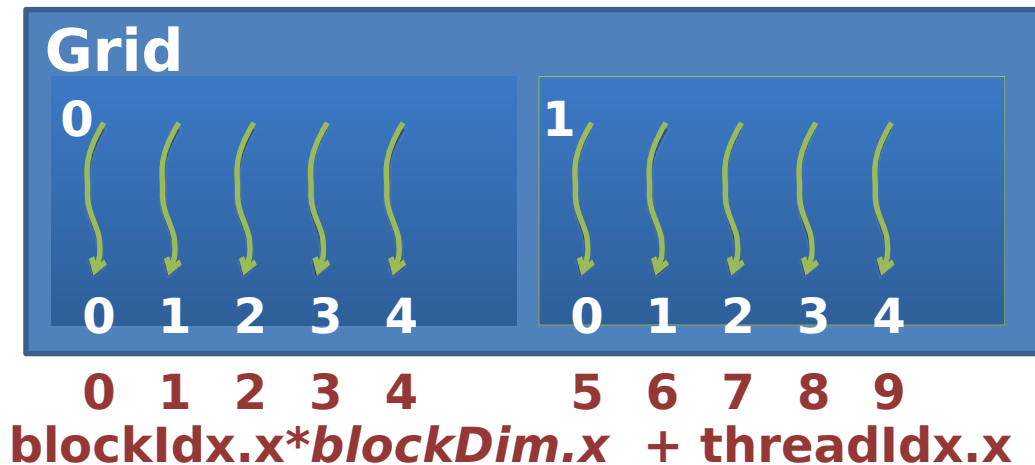
- Kernel executed by a grid of thread blocks
  - 3D Grid
  - 3D blocks
- In a block, the threads
  - Cooperate via shared memory
  - Are scheduled by warp
    - *Warp = 32 threads*
  - Threads of a warp are synchronous
- No inter-block cooperation



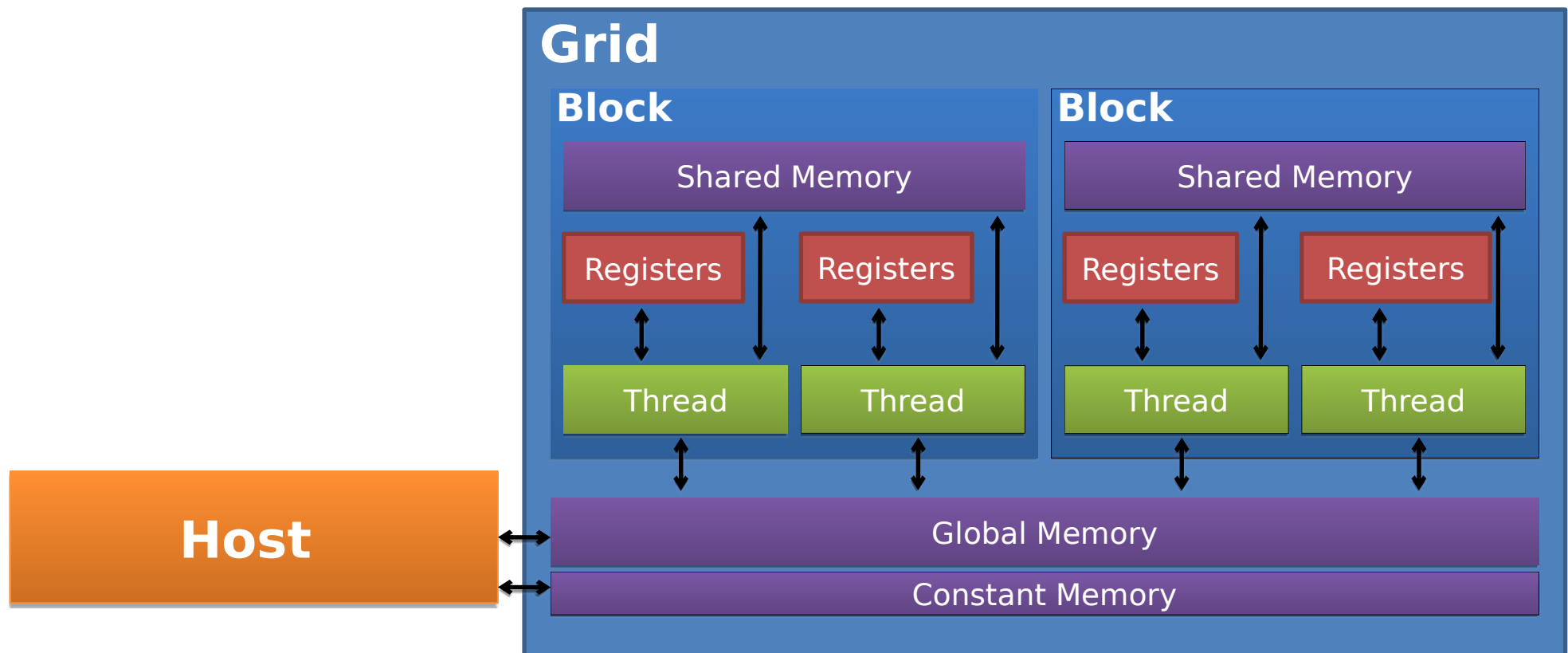
# Multi-dimensional identifiers

- 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`

`blockIdx.x`  
  
`blockDim.x = 5`  
  
`threadIdx.x`

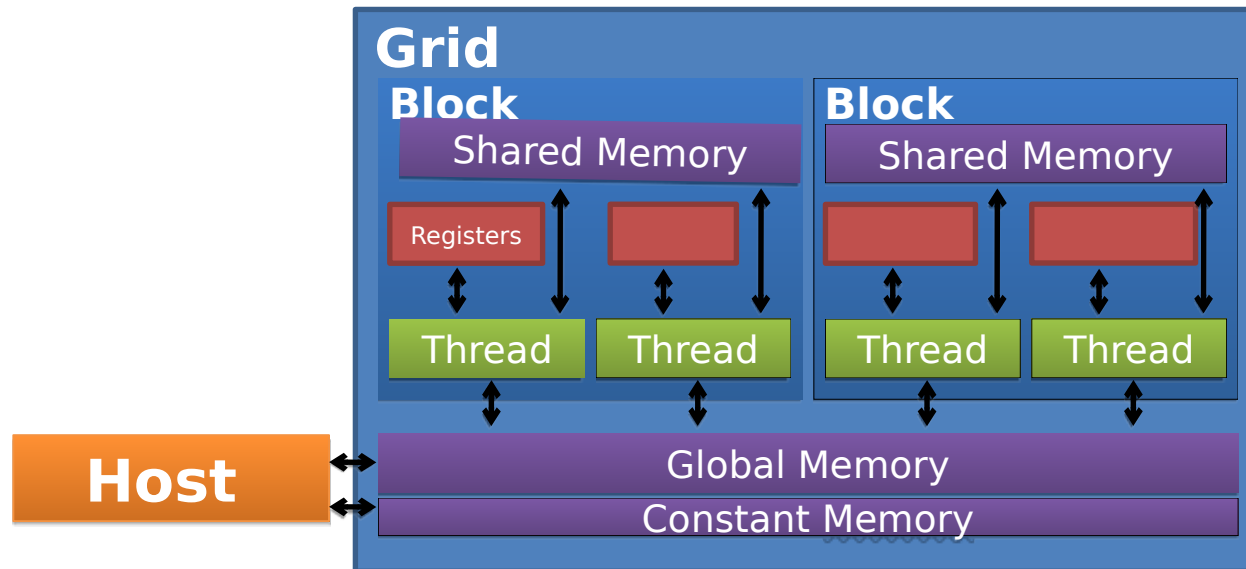


# Memory



# GPU Memory Architecture

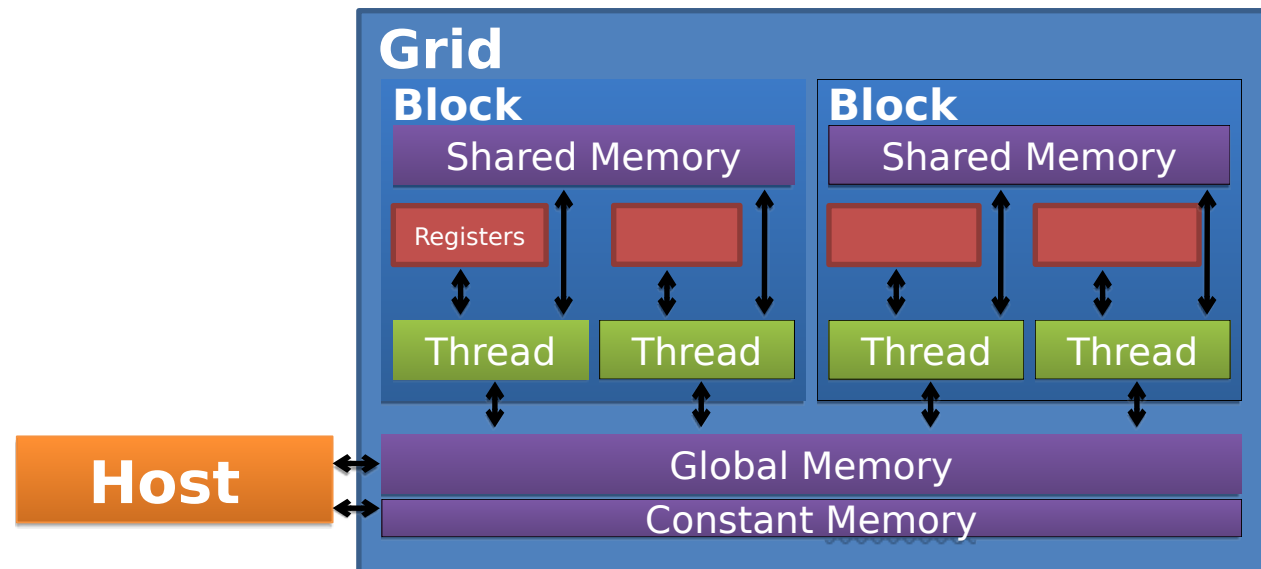
CPU and GPU memory spaces physically separated



- Explicit transfers between the two spaces
- Two entry points on the GPU
  - Global and constant memories

# GPU memory hierarchy

- On GPU, 4 levels of memory [+ texture memory]
  - Global memory [\_\_device\_\_]
  - Constant memory [\_\_device\_\_] \_\_constant\_\_
  - Shared memory [\_\_device\_\_] \_\_shared\_\_
  - Registers





# A) Global Memory

- Large, high latency, no cache
- Data
  - Accessible by all the threads of the grid
  - Lifespan : as required by the application
- From host,
  - Allocation/Free + copies in both ways
- Static declaration from the GPU with keyword `__device__`

# Global memory management

- Allocation : **cudaMalloc**(void \*\* pointer, size\_t nbytes)
- Desallocation : **cudaFree**(void\* p)
- Cleaning : **cudaMemset**(void \* p, int val, size\_t nbytes)
- Copy of the data from host :  
**cudaMemcpy**(void \*dst, void \*src,  
size\_t nbytes,  
enum cudaMemcpyKind direction);

with enum **cudaMemcpyKind**

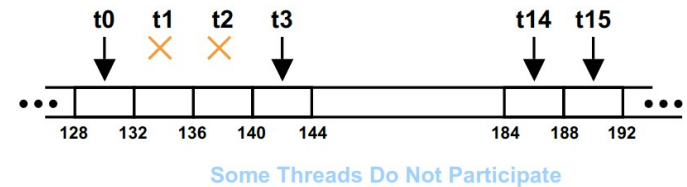
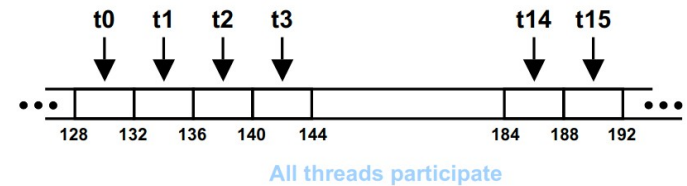
= {*cudaMemcpyHostToDevice*,  
*cudaMemcpyDeviceToHost*,  
*cudaMemcpyDeviceToDevice*}



# Global Memory coalescing

- Multiple memory accesses into a single transaction

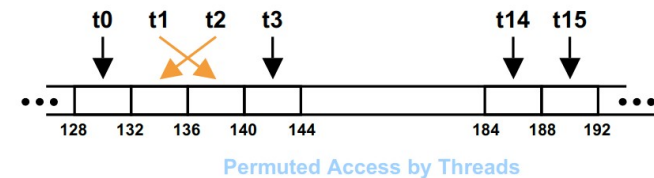
Coalesced Access:  
Reading floats



- Uncoalesced load,  
ie serialized memory access,  
when memory accesses

- are not sequential
- are sparse
- are misaligned

Uncoalesced Access:  
Reading floats





## B) Constant Memory

- For data that will not change over a kernel execution
- Read-only, pretty small memory, slow, cached
  - The first read from constant memory costs one memory read from global memory ; after, costs one read from the constant cache
  - Cache for each multiprocessor very small
    - Optimized when warp of threads read same location
- Data accessible by all the threads of the grid

# Constant memory management

- Declaration : `__constant__ float buffer [size];`
- Copy of the data from the host :

`cudaError_t cudaMemcpytoSymbol`

`(const char * symbol,  
const void * src, size_t count ,  
size_t offset=0,  
enum cudaMemcpyKind )`

with enum **cudaMemcpyKind**

`= {cudaMemcpyHostToDevice,  
cudaMemcpyDeviceToHost,  
cudaMemcpyDeviceToDevice}`

## C) Shared Memory

- Keyword `__shared__`
  - Separate space with very low latency
- Data
  - Accessible by all threads of the same block
  - Lifetime: kernel run
- Static allocation
  - From the GPU device
  - Static size given at compile time (case a)  
or at the kernel launch (case b)

```
// case a
__global__ void myKernel(){
    __shared__ int shared[32];
    ...
}
```

```
// case b
__global__ void myKernel(){
    extern __shared__ int s[];
    ...
}
int main() {
    int size= numThreadsPerBlock* sizeof(int);
    myKernel<<< dimGrid, dimBlock, size>>>();}
```



# Shared memory management

- All operations on the device within a same kernel
- Static allocation from device : `__shared__ int tab[4] ;`
- Classic explicit initialization/modification in kernel  
`for (int i = 0 ; i < 4 ; i++) tab[i]=i ;`



## D) Registers

- Fast, only for one thread
- For local kernel variables
  - Allocation of scalar variables in registers
  - Allocation of arrays of more than 4 elements in the global memory
- No specific keyword

# **CUDA C PROGRAMMING**

# 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);
}
```



# 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

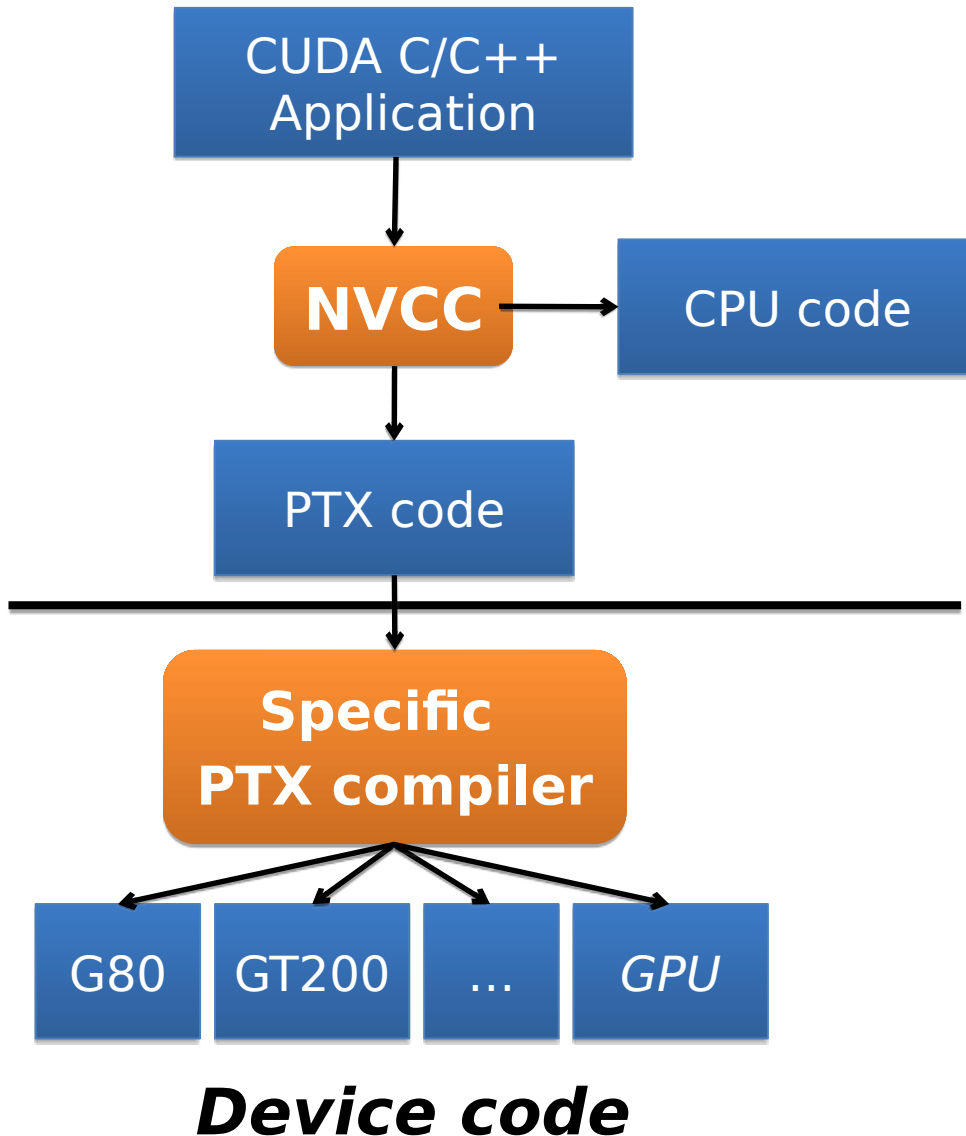
- Kernels are asynchronous
  - Kernel calls return immediately
  - Kernels run after all previous ones have run
- `cudaMemcpy()` is synchronous
  - Call returns after the copy is made
  - Copying starts after all previous CUDA calls have been executed
- **`cudaThreadSynchronize()`**
  - Blocks until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

# Synchronization on GPU

- `void __syncthreads();`
  - Synchronizes all threads of a block
- Atomic operations
  - `atomicAdd()`
  - `atomicSub()`
  - `atomicMin()`
  - `atomicMax()`
  - `atomicInc()`
  - `atomicDec()`
  - `atomicExch()`
  - `atomicCAS()`
  - `atomicAnd()`
  - `atomicOr()`
  - `atomicXor()`



# Compilation



- Meta-compiler nvcc
  - CPU and GPU codes
- Binary containing CUDA code requires
  - CUDA core library (cuda)
  - CUDA runtime library (cudart) 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 !