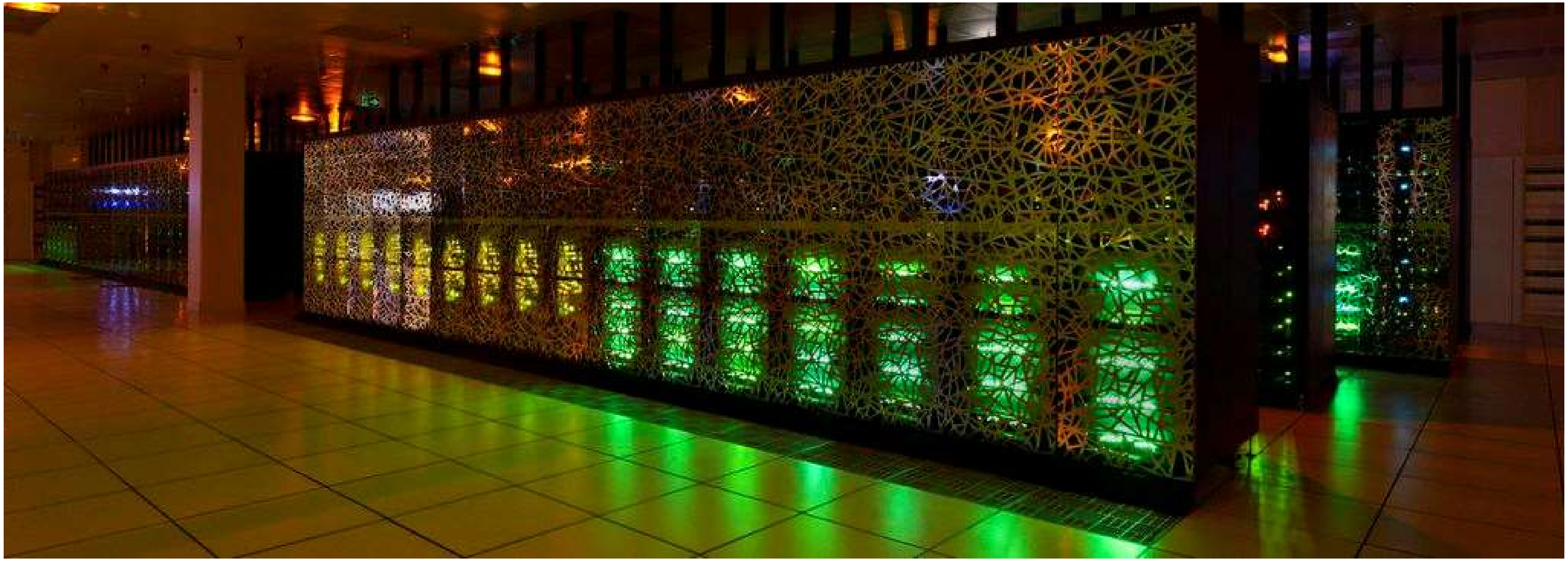
center for excellence in parallel programming

CUDA

Georges-Emmanuel Moulard
Paul Karlshöfer



Bull atos technologies

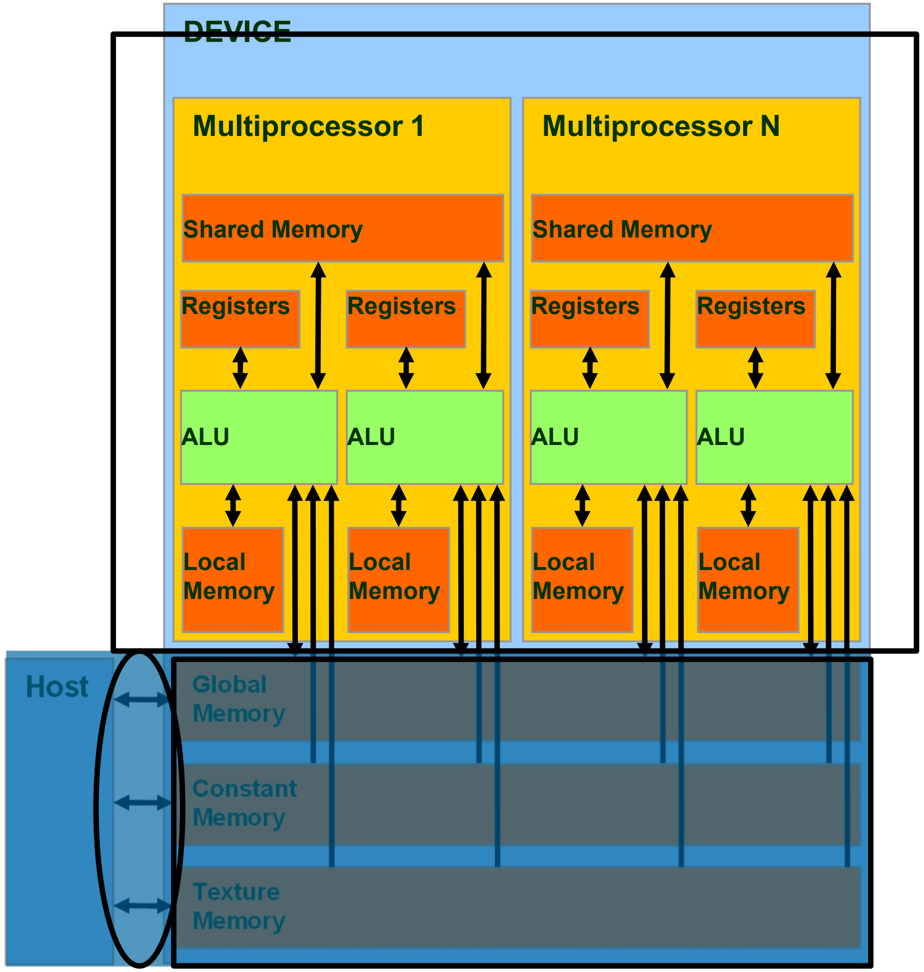
GPU Overview

16/09/2019

Bull
atos technologies

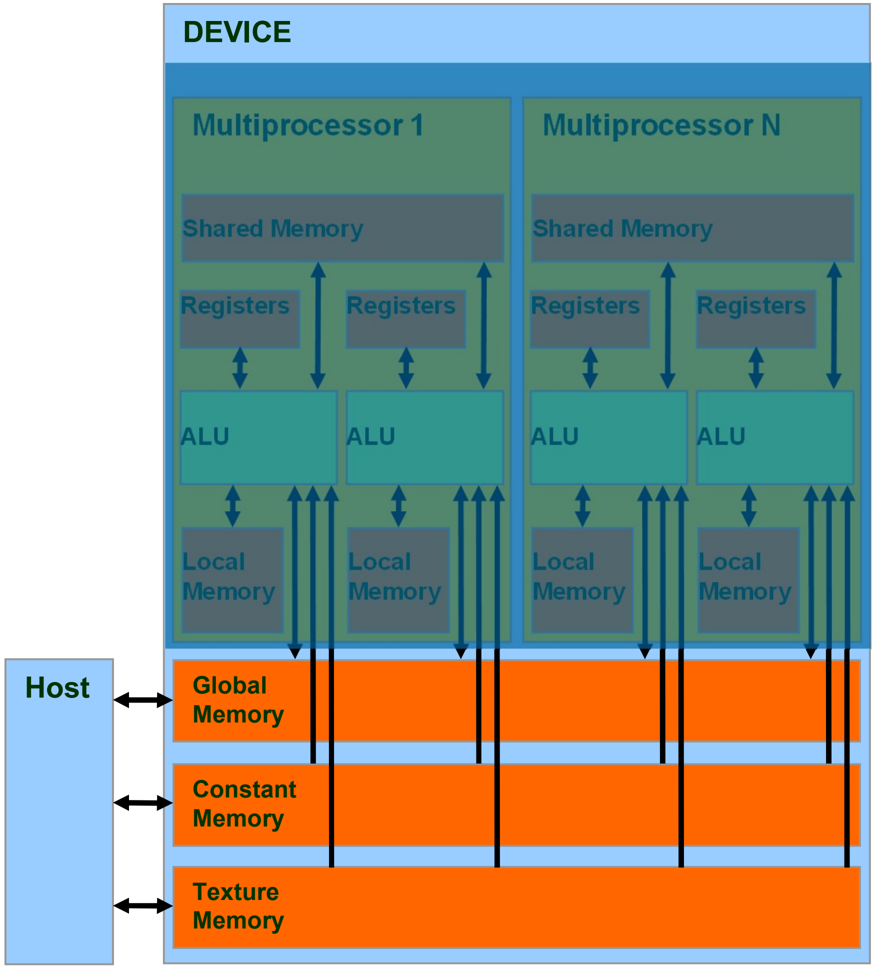
Hardware Overview

- **Device** contains
  - Multiprocessors
  - Memory
  - Host access interface
- **Several generations :**
  - 1.x TESLA, 2.x FERMI, 3.x KEPLER,
  - 6.x PASCAL (Tesla P100,P40,P4)
  - 7.x VOLTA (Tesla V100)
  - Ampere
- **Multiprocessors** contains
  - ALUs
  - Registers
  - Shared Memory
  - Access to Local Memory
  - Access to Global Memory



Memory Overview

- Global memory
  - Main memory to communicate (R/W data) between host and device
  - Contents visible to all threads
  - No cache (HW < 2.0)
  - Cache L1/L2 (HW >= 2.0)
- Texture and constant memories
  - Constants initialized by host
  - Contents visible to all threads
  - Cache available



Bull atos technologies

Device Query

```
Device 0: "Tesla K80"
  CUDA Driver Version / Runtime Version          6.5 / 6.5
  CUDA Capability Major/Minor version number:    3.7
  Total amount of global memory:                 11520 MBytes (12079136768 bytes)
  (13) Multiprocessors, (192) CUDA Cores/MP:     2496 CUDA Cores
  GPU Clock rate:                                824 MHz (0.82 GHz)
  Memory Clock rate:                             2505 Mhz
  Memory Bus Width:                              384-bit
  L2 Cache Size:                                 1572864 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536, 65536), 3D=(4096, 4096, 4096)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 2 copy engine(s)
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Enabled
  Device supports Unified Addressing (UVA):      Yes
  Device PCI Bus ID / PCI location ID:           4 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >
```

How to Compute on GPU

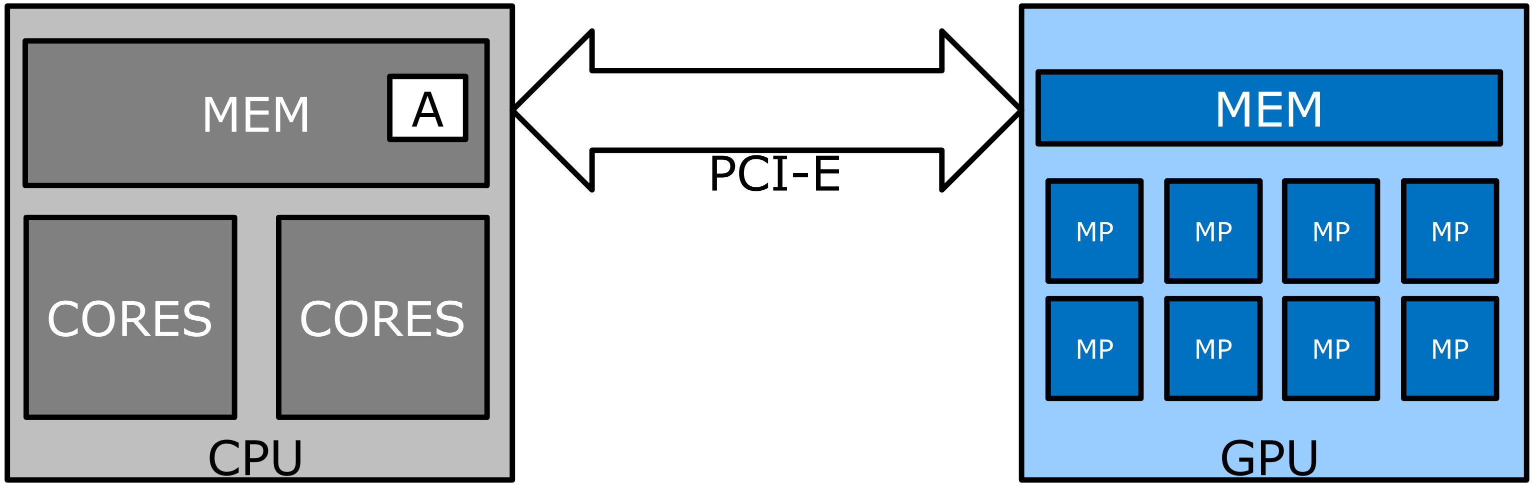
16/09/2019

Bull
atos technologies

How to Compute on GPU

- **5 steps to offload computation on the GPU:**

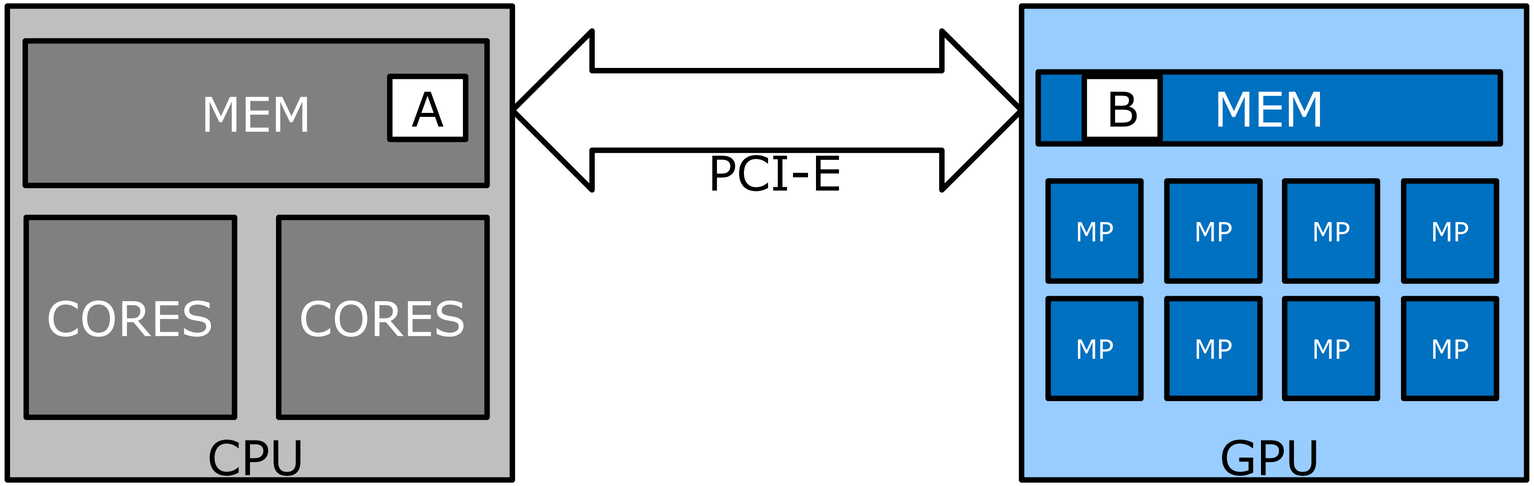
- (1) Memory Allocation : **cudaMalloc(&B, …)**
- (2) H2D transfer : **cudaMemcpy(B, A, …)**
- (3) Execute : **kernel<<< … >>>(…)**
- (4) D2H transfer : **cudaMemcpy(A, B, …)**
- (5) Free Memory : **cudaFree(B)**



How to Compute on GPU

- **5 steps to offload computation on the GPU:**

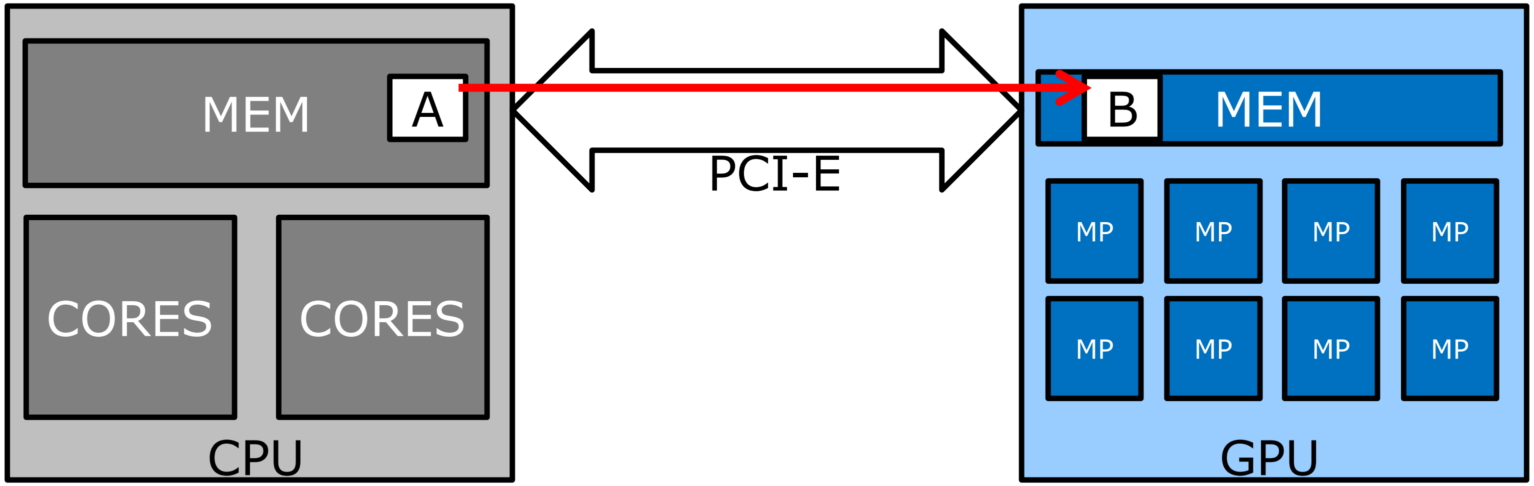
- (1) Memory Allocation : **cudaMalloc(&B, …)**
- (2) H2D transfer : **cudaMemcpy(B, A, …)**
- (3) Execute : **kernel<<< … >>>(…)**
- (4) D2H transfer : **cudaMemcpy(A, B, …)**
- (5) Free Memory : **cudaFree(B)**



How to Compute on GPU

- **5 steps to offload computation on the GPU:**

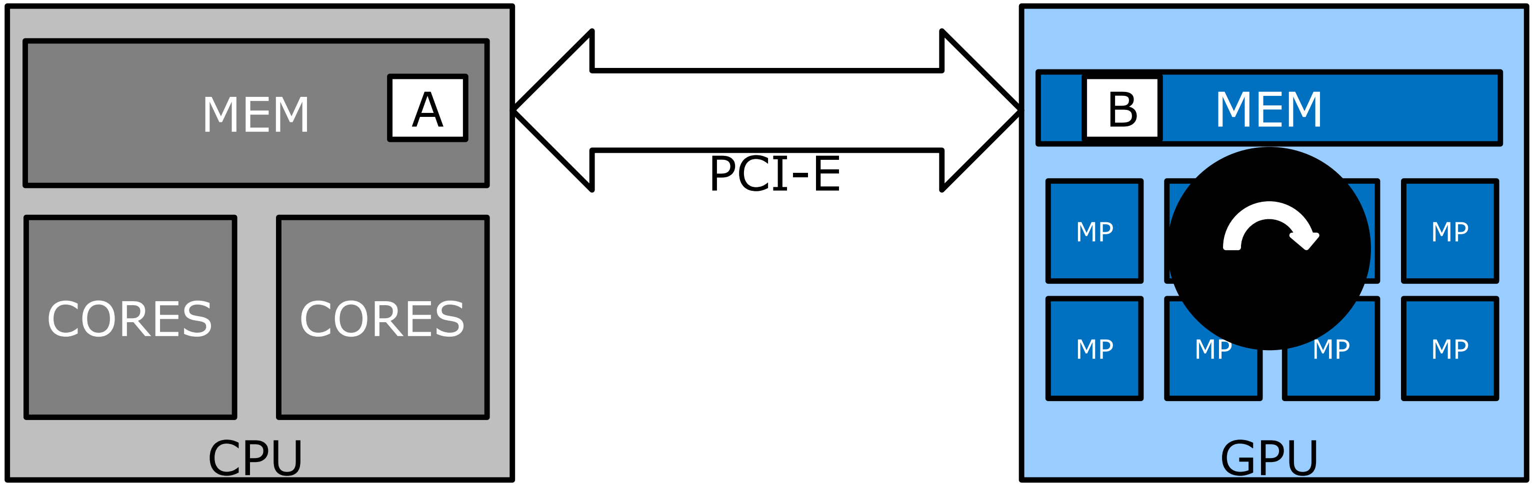
- (1) Memory Allocation : **cudaMalloc(&B, …)**
- (2) H2D transfer : **cudaMemcpy(B, A, …)**
- (3) Execute : **kernel<<< … >>>(…)**
- (4) D2H transfer : **cudaMemcpy(A, B, …)**
- (5) Free Memory : **cudaFree(B)**



How to Compute on GPU

- **5 steps to offload computation on the GPU:**

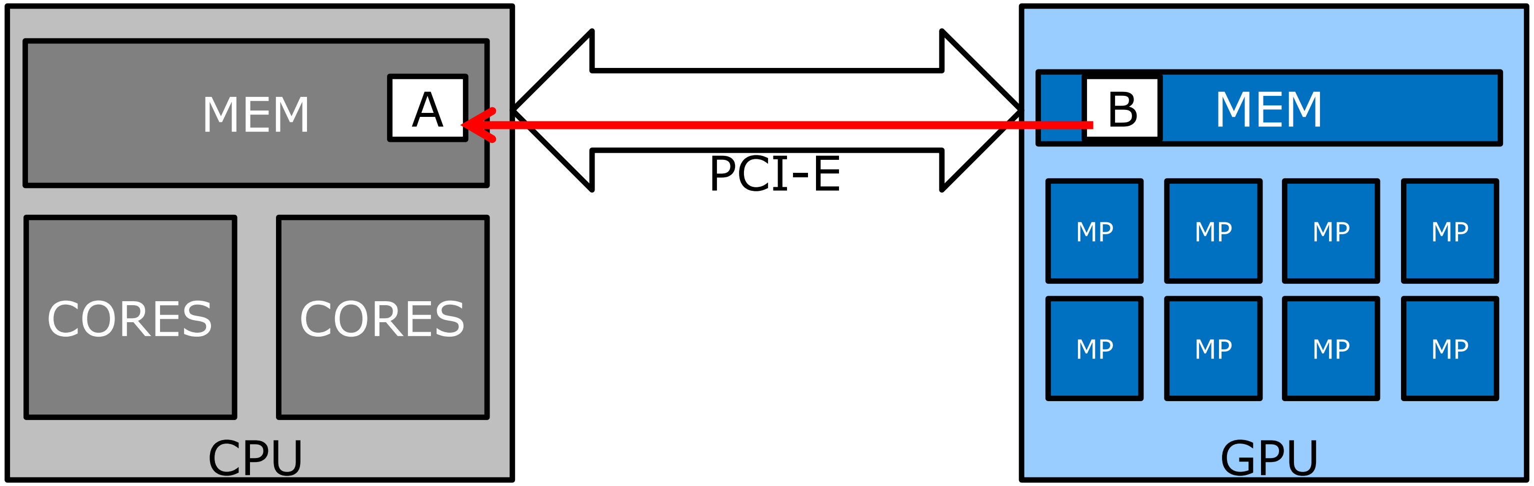
- (1) Memory Allocation : **cudaMalloc(&B, …)**
- (2) H2D transfer : **cudaMemcpy(B, A, …)**
- (3) Execute : **kernel<<< … >>>(…)**
- (4) D2H transfer : **cudaMemcpy(A, B, …)**
- (5) Free Memory : **cudaFree(B)**



How to Compute on GPU

- **5 steps to offload computation on the GPU:**

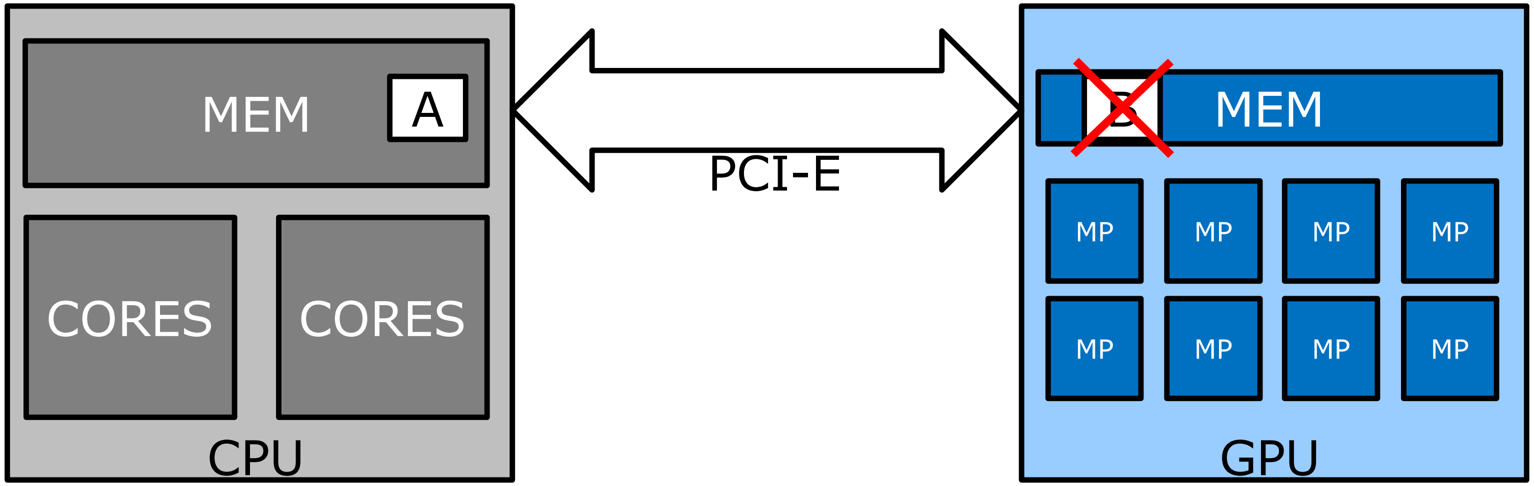
- (1) Memory Allocation : **cudaMalloc(&B, …)**
- (2) H2D transfer : **cudaMemcpy(B, A, …)**
- (3) Execute : **kernel<<< … >>>(…)**
- (4) D2H transfer : **cudaMemcpy(A, B, …)**
- (5) Free Memory : **cudaFree(B)**



How to Compute on GPU

- **5 steps to offload computation on the GPU:**

- (1) Memory Allocation : **cudaMalloc(&B, …)**
- (2) H2D transfer : **cudaMemcpy(B, A, …)**
- (3) Execute : **kernel<<< … >>>(…)**
- (4) D2H transfer : **cudaMemcpy(A, B, …)**
- (5) Free Memory : **cudaFree(B)**

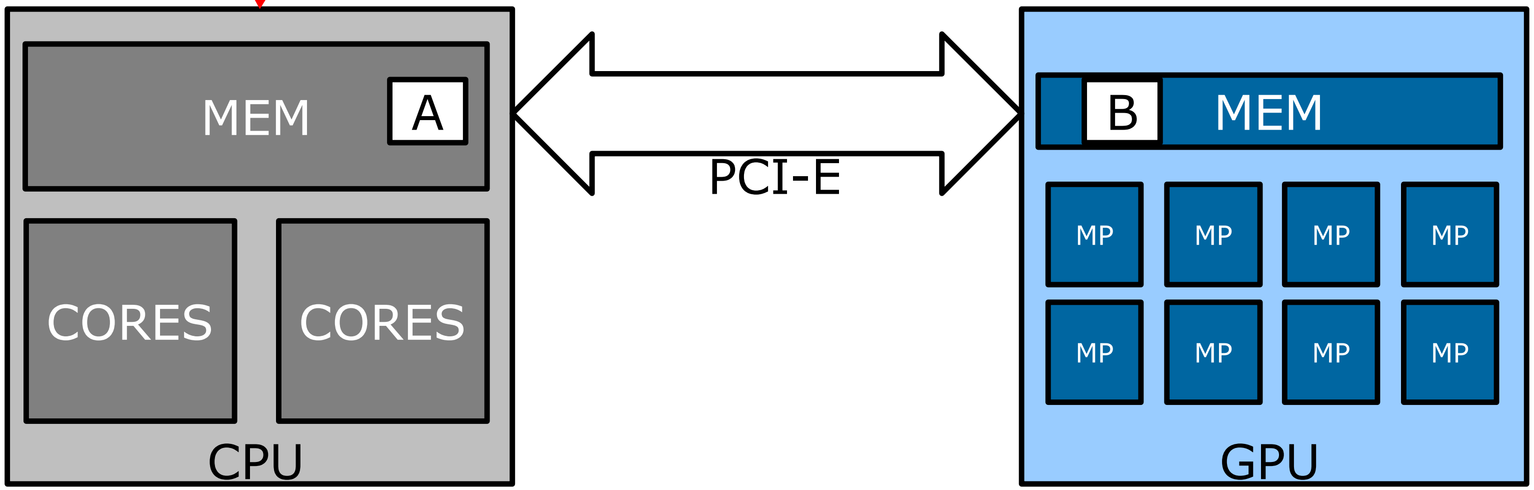


How to Compute on GPU

- **5 steps to offload computation on the GPU:**

- (1) Memory Allocation : `cudaMalloc( &B, … )`
- (2) H2D transfer : `cudaMemcpy(B, A, … )`
- (3) Execute : `kernel<<< … >>>( … )`
- (4) D2H transfer : `cudaMemcpy(A, B, … )`
- (5) Free Memory : `cudaFree( B )`

HOST CODE
CPU calls to GPU driver

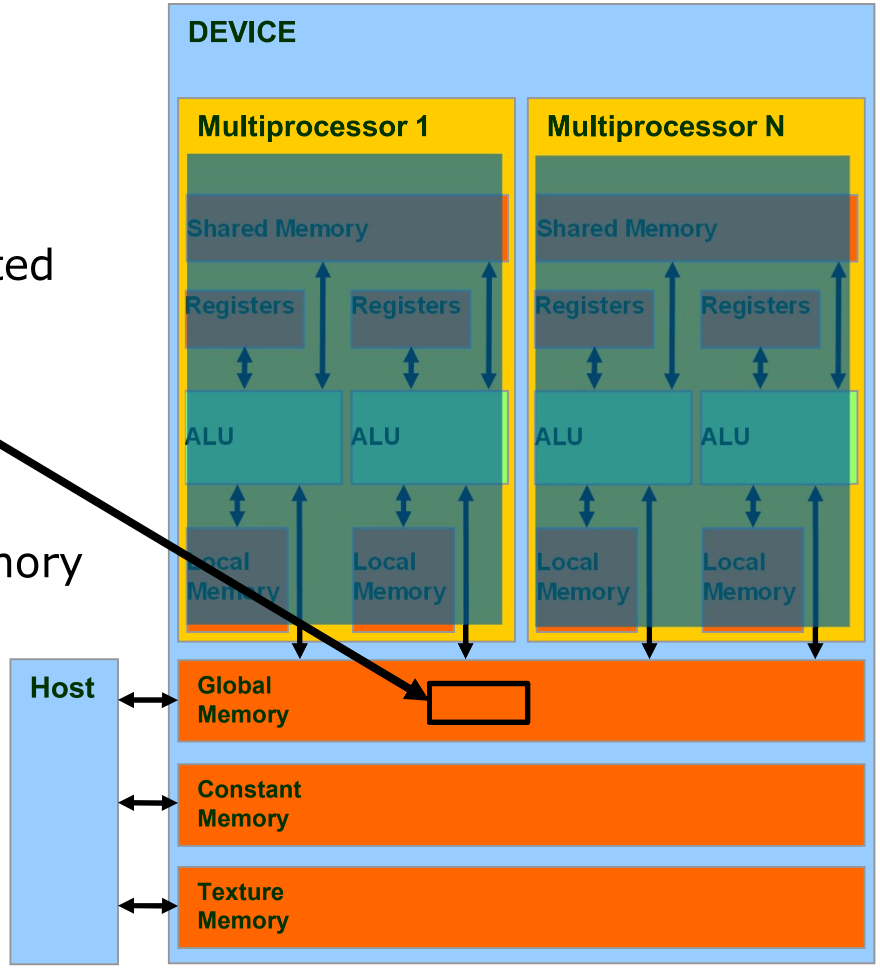


Allocation & Data Movement

16/09/2019

Memory Allocation

- **cudaMalloc()**
  - Allocates object in the device Global Memory
  - Requires two parameters
    - Address of a pointer to the allocated object
    - Size of allocated object
- **cudaFree()**
  - Frees object from device Global Memory
    - Pointer to freed object



Memory Allocation Example

- Allocate a 1024 * 1024 single precision float matrix:

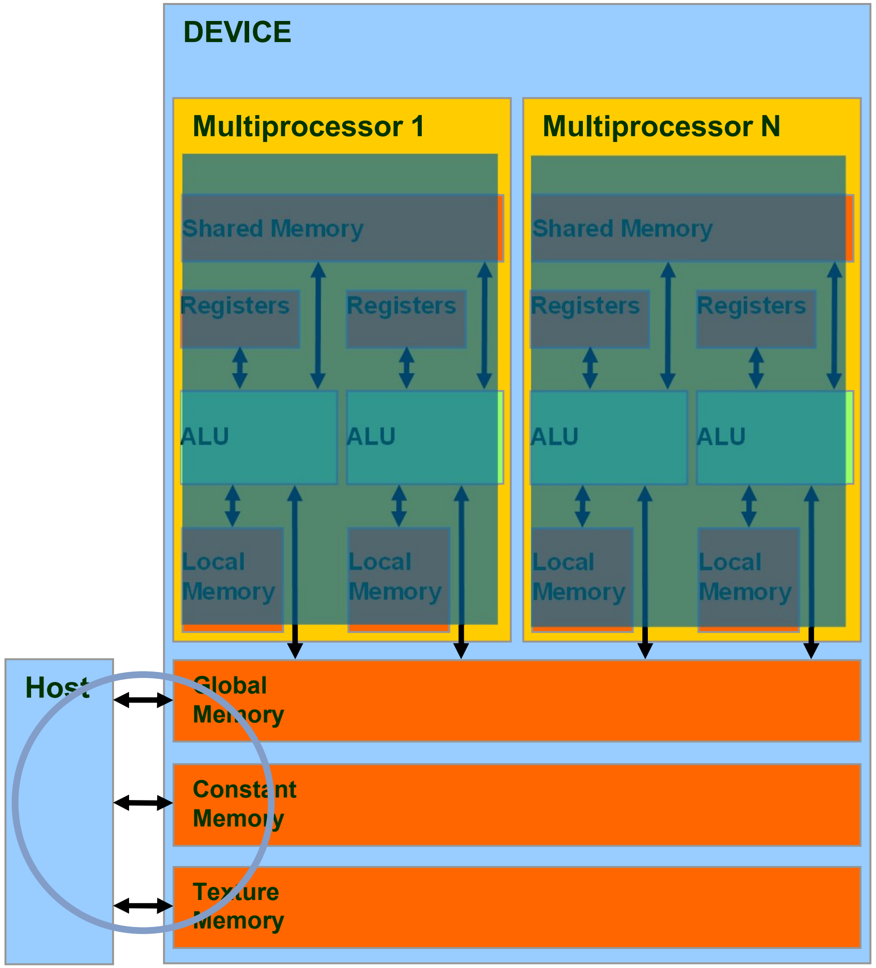
```
float* MyMatrixOnDevice;
int size = 1024 *1024 * sizeof(float);

cudaMalloc( (void**) &MyMatrixOnDevice, size);
```

- Free the matrix:

```
cudaFree(MyMatrixOnDevice);
```


Data Transfer

- **cudaMemcpy()**
  - memory data transfer
  - Requires 4 parameters
    1. Pointer to destination
    2. Pointer to source
    3. Number of bytes copied
    4. Type of transfer
       - Host to Host
       - Host to Device
       - Device to Host
       - Device to Device
- Asynchronous variant supported



Data Transfer Example

- Transfer of a 1024 * 1024 single precision float array:
  - host_ptr: pointer on host memory
  - device_ptr: pointer on GPU memory

- Send data from CPU to the GPU:

```
cudaMemcpy(device_ptr, host_ptr,  size, cudaMemcpyHostToDevice);
```

- Send data from GPU to the CPU:

```
cudaMemcpy(host_ptr, device_ptr, size, cudaMemcpyDevicetoHost);
```

How to Check Errors for Synchronous Calls

HOST CODE

```
#define CudaSafeCall( err )   __cudaSafeCall( err, __FILE__, __LINE__ )

void __cudaSafeCall( cudaError err, const char *file, const int line ){
#if defined(DEBUG) || defined(_DEBUG)
    if ( cudaSuccess != err )
    {
    fprintf( stderr, "cudaSafeCall() failed at %s:%i : %s\n",file, line, cudaGetErrorString(err));
    exit( err );
    }
#endif
}
```

```
CudaSafeCall ( cudaMalloc(…) );
CudaSafeCall ( cudaMemcpy(…) );
CudaSafeCall ( cudaFree(…) );
```

How to Check Errors for Asynchronous Calls

HOST CODE

```
#define CudaCheckError()                __cudaCheckError( __FILE__, __LINE__ )

void __cudaCheckError( const char *file, const int line ){
#if defined(DEBUG) || defined(_DEBUG)
  cudaError err = cudaGetLastError();
  if ( cudaSuccess != err )
  {
    fprintf( stderr, "cudaCheckError() failed at %s:%i : %s\n",file, line, cudaGetErrorString( err));
    exit( err );
  }
#endif
}
```

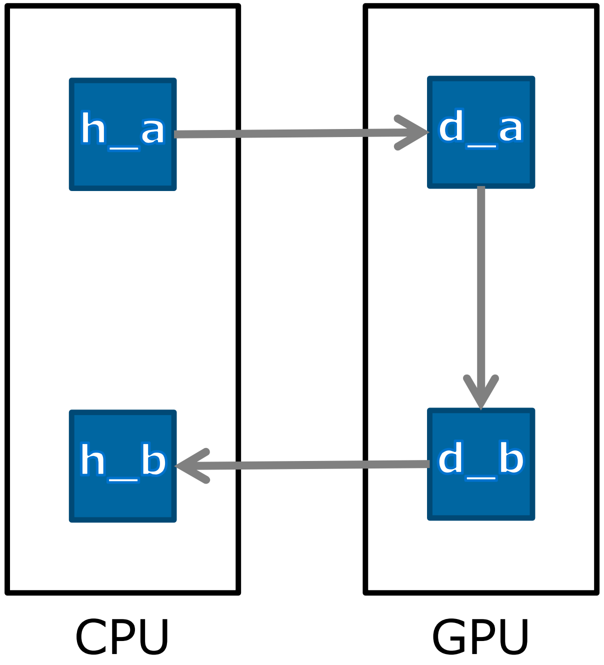
```
Kernel<<<..., ...>>>(…);
cudaDeviceSynchronize();
cudaCheckError();
```

LAB: Data Movement

- Complete the main.cu file

- Steps:

  - allocate d_a and d_b on GPU
  - copy h_a into d_a
  - copy d_a into d_b
  - copy d_b into h_b
  - free d_a and d_b

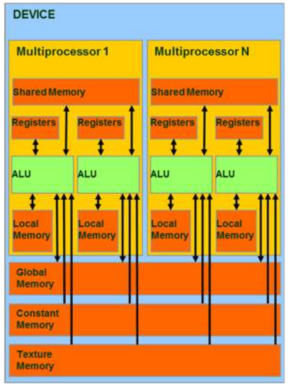


Kernel Execution

16/09/2019

Kernel Execution

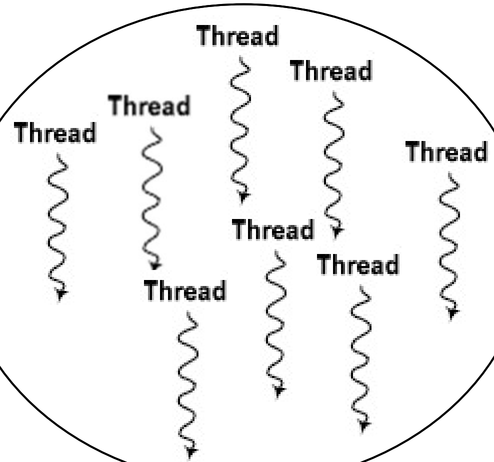
A hardware abstract view



GPU functions

```
__global__ void myFunction( ... )
{

  ...

}
```

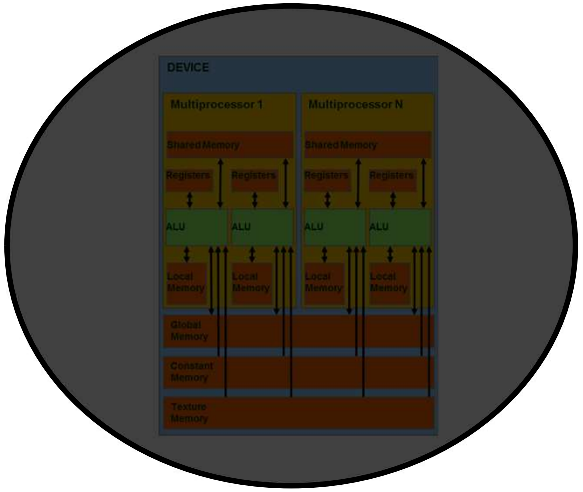
Pools of threads



Bull
atos technologies

Kernel Execution

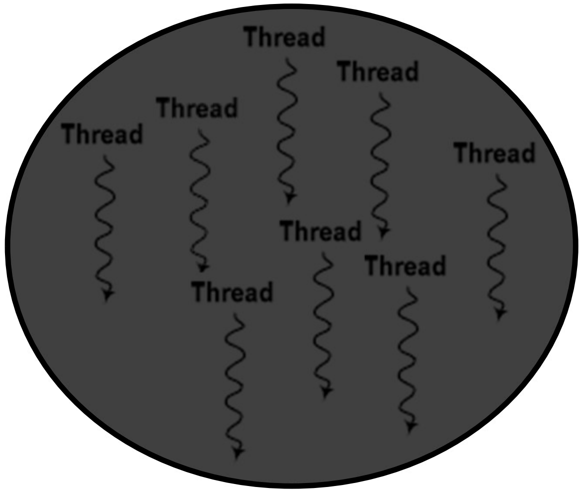
A hardware abstract view



GPU functions

```
__global__ void myFunction( ... )
{

  ...

}
```

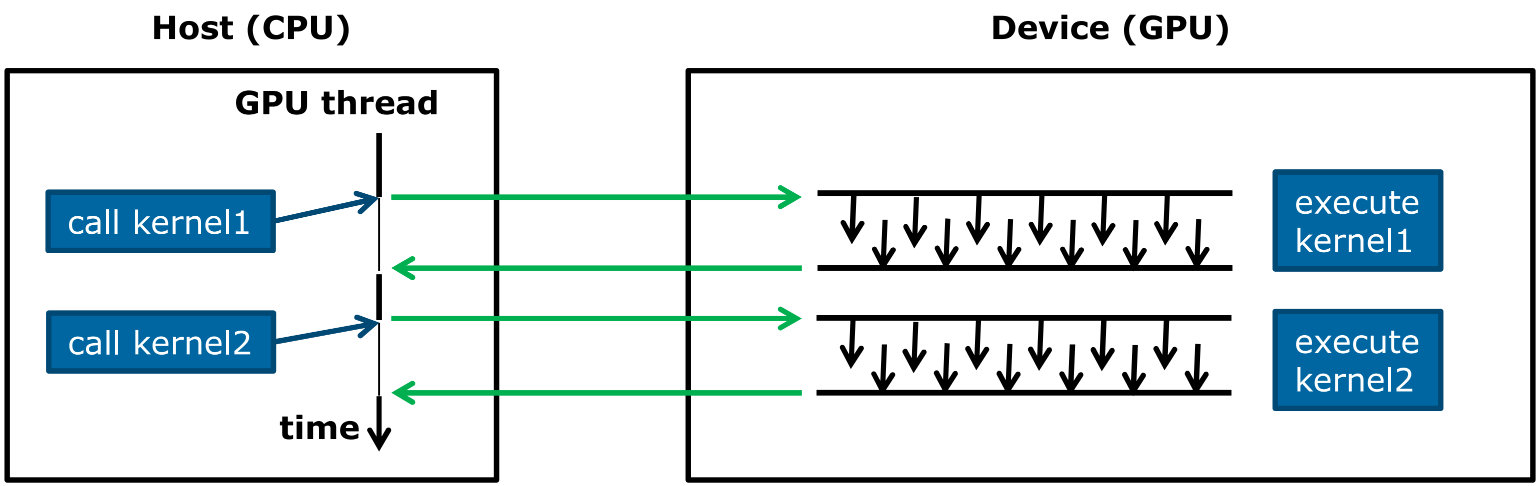
Pools of threads



How to Execute Code on GPU

- Execution is always launched **from CPU** (offload)

- The function run on a GPU is called a **kernel**
  - CPU continues its execution while kernel is running on GPU
  - 1 GPU can run multiple kernels concurrently (streams / Multi-Process Service)



How to Execute Code on GPU

- A kernel is a **function** executed N times in parallel by N different CUDA threads on the device

- A kernel is defined using the **__global__** declaration specifier
  - It must return void

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C) {
  ...
}
```

- A kernel will be executed by multiple CUDA threads organized in **a grid of thread blocks**

- Each thread that executes the kernel is given a **unique *thread ID*** that is **accessible within the kernel** through built-in variables.

KERNEL CALL & CUDA FUNCTIONS

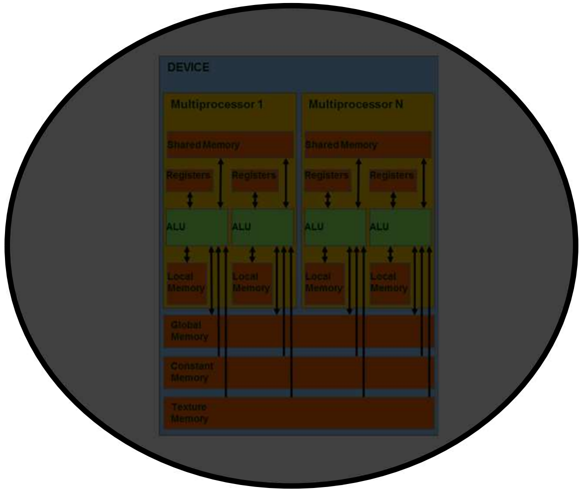
- Keywords specified before function definition:
  - **__host__** : cpu code (keyword not necessary)
  - **__global__** : kernel declaration
  - **__device__** : function running inside a GPU kernel

- A CUDA kernel can only make call to device fonctions:

```
//__global__ defines a kernel (a function executed on GPU by several threads)
__global__ void Kernel_name(…){
  …
  cuda_func(…);
  …
}


//__device__ defines a function callable from a kernel (a gpu version of a function)
__device__ void cuda_func(…) {
  …
}
```

Kernel Execution

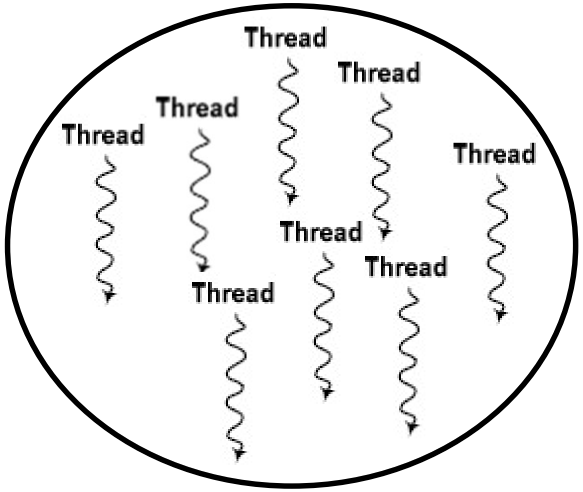
A hardware abstract view



GPU functions

```
__global__ void myFunction( ... )
{

  ...

}
```

Grid of threads



How to Execute Code on GPU

- The grid of threads that execute a given kernel is specified using the **<<< … >>>** execution configuration syntax

```
// Kernel call

VecAdd<<< numBlocks,  threadsPerBlock [, …, …] >>> (A,B,C)
```

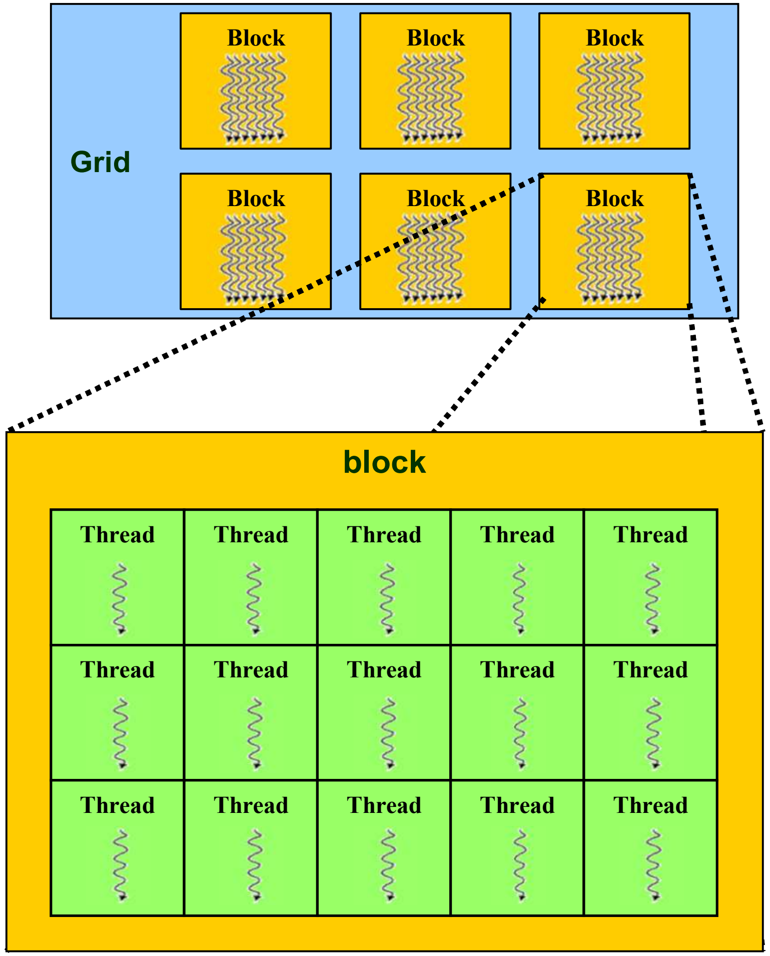
- Programmer specifies the amount of threads
  - threadsPerBlocks: the number of threads in a thread block
  - numBlocks: the number of thread blocks in the grid

- **All threadblocks** contain the **same constant amount of threads**

- The total amount of threads to run on the GPU is:
  - #total_threads = threadsPerBlocks * numBlocks

Grid of Threads

▶ All blocks contain the same amount of threads



QUIZ

- Execution is always launched from

QUIZ

- Execution is always launched from **CPU**

QUIZ

- Execution is always launched from **CPU**

- The function run on a GPU is called a

QUIZ

- Execution is always launched from **CPU**
- The function run on a GPU is called a **kernel**

QUIZ

- Execution is always launched from **CPU**
- The function run on a GPU is called a **kernel**
- A kernel is defined using the declaration specifier

QUIZ

- Execution is always launched from **CPU**
- The function run on a GPU is called a **kernel**
- A kernel is defined using the **__global__** declaration specifier

QUIZ

- Execution is always launched from **CPU**
- The function run on a GPU is called a **kernel**
- A kernel is defined using the **__global__** declaration specifier
- A kernel is executed N times in parallel by N different

QUIZ

- Execution is always launched from **CPU**
- The function run on a GPU is called a **kernel**
- A kernel is defined using the **__global__** declaration specifier
- A kernel is executed N times in parallel by N different **CUDA threads**

QUIZ

- Execution is always launched from **CPU**
- The function run on a GPU is called a **kernel**
- A kernel is defined using the **__global__** declaration specifier
- A kernel is executed N times in parallel by N different **CUDA threads**
- CUDA threads are divided up into

QUIZ

- Execution is always launched from **CPU**
- The function run on a GPU is called a **kernel**
- A kernel is defined using the **__global__** declaration specifier
- A kernel is executed N times in parallel by N different **CUDA threads**
- CUDA threads are divided up into **threadblocks**

QUIZ

- Execution is always launched from **CPU**
- The function run on a GPU is called a **kernel**
- A kernel is defined using the **__global__** declaration specifier
- A kernel is executed N times in parallel by N different **CUDA threads**
- CUDA threads are divided up into **threadblocks**
- A threadblock is a group of

QUIZ

- Execution is always launched from **CPU**
- The function run on a GPU is called a **kernel**
- A kernel is defined using the **__global__** declaration specifier
- A kernel is executed N times in parallel by N different **CUDA threads**
- CUDA threads are divided up into **threadblocks**
- A threadblock is a group of **threads**

QUIZ

- Execution is always launched from **CPU**
- The function run on a GPU is called a **kernel**
- A kernel is defined using the **__global__** declaration specifier
- A kernel is executed N times in parallel by N different **CUDA threads**
- CUDA threads are divided up into **threadblocks**
- A threadblock is a group of **threads**
- All threadblocks contain the amount of threads

QUIZ

- Execution is always launched from **CPU**
- The function run on a GPU is called a **kernel**
- A kernel is defined using the **__global__** declaration specifier
- A kernel is executed N times in parallel by N different **CUDA threads**
- CUDA threads are divided up into **threadblocks**
- A threadblock is a group of **threads**
- All threadblocks contain the **same** amount of threads

QUIZ

- Execution is always launched from **CPU**
- The function run on a GPU is called a **kernel**
- A kernel is defined using the **__global__** declaration specifier
- A kernel is executed N times in parallel by N different **CUDA threads**
- CUDA threads are divided up into **threadblocks**
- A threadblock is a group of **threads**
- All threadblocks contain the **same** amount of threads
- The contains all the thread blocks

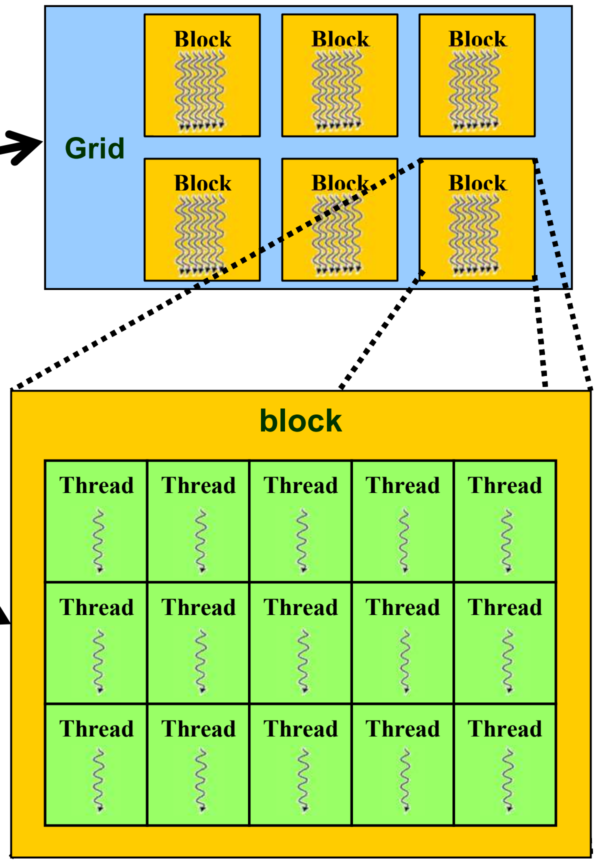
QUIZ

- Execution is always launched from **CPU**
- The function run on a GPU is called a **kernel**
- A kernel is defined using the **__global__** declaration specifier
- A kernel is executed N times in parallel by N different **CUDA threads**
- CUDA threads are divided up into **threadblocks**
- A threadblock is a group of **threads**
- All threadblocks contain the **same** amount of threads
- The **grid** contains all the thread blocks

HOW TO MANAGE THE THREADS ?

- Programmer specifies threads dimensions
  - The programmer sets the size of each dimension
  - The **dim3** CUDA defined type is used for that:

```
dim3 dimBlock(5, 3, 1);
dim3 dimGrid(3, 2, 1);

Kernel_name<<< dimGrid, dimBlock >>>(…)
```



- or

```
dim3 dimBlock, dimGrid;
dimBlock.x = 5;
dimBlock.y = 3;
dimGrid.x = 3;
dimGrid.y = 2;

Kernel_name<<< dimGrid, dimBlock >>>(…)
```

LAB: hello world

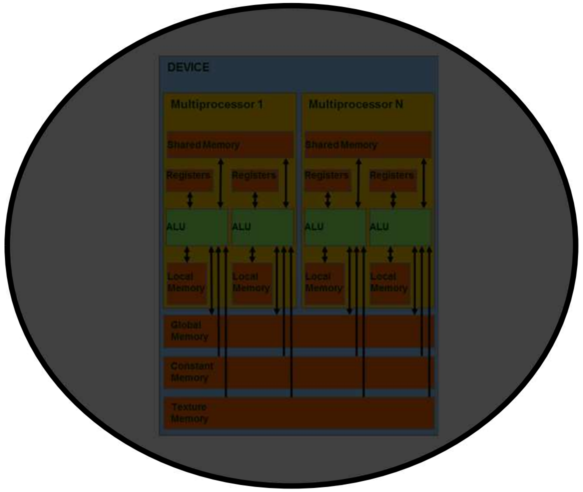
- Call a kernel performing *printf(''hello world\n'')* with 2 different CUDA grid
- What did you see?

LAB: Add in Scalar

- Call a kernel which does an accumulation in a scalar
- What did you see?

Kernel Execution

A hardware abstract view



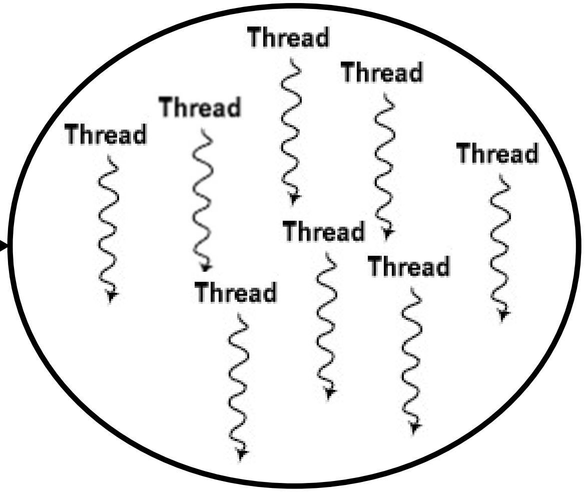
GPU functions

```
__global__ void myFunction( ... )
{
  ...
}
```

<<< M, N >>>

?

Pools of threads

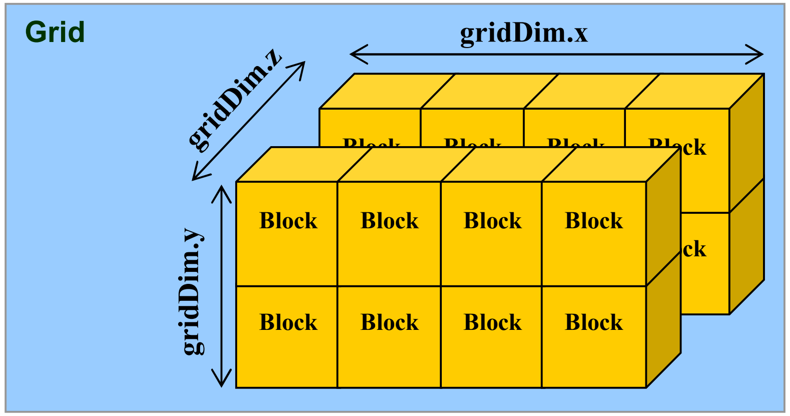


How to Differentiate Threads in a Kernel

- The number of threadblocks in the grid can be retrieved inside the kernel through the built-in variables : **gridDim.{x,y,z}**

```
dim3 nbThreads(4,3,3);
dim3 nbBlocks(4,2,2);
mykernel<<< nbBlocks, nbThreads >>>(…)
```

```
__global__ void mykernel(…) {

  int nb_blocks_x = gridDim.x;
  int nb_blocks_y = gridDim.y;
  int nb_blocks_z = gridDim.z;
  …
}
```



- All the threads executing the kernel have the same values for gridDim.{x,y,z}

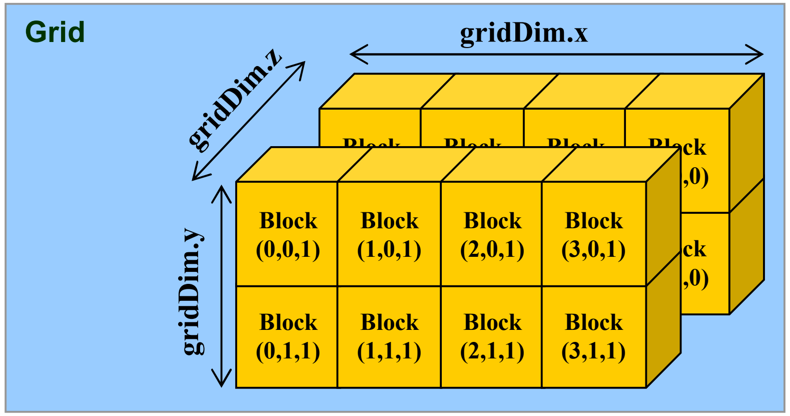
```
// In this example:
nb_blocks_x = 4
nb_blocks_y = 2
nb_blocks_z = 2
```

How to Differentiate Threads in a Kernel

- The position (index) of threadblocks in the grid can be retrieved inside the kernel through the built-in variables : **blockIdx.{x,y,z}**

```
dim3 nbThreads(4,3,3);
dim3 nbBlocks(4,2,2);
mykernel<<< nbBlocks, nbThreads >>>(…)
```

```
__global__ void mykernel(…) {

  int block_index_x = blockIdx.x;
  int block_index_y = blockIdx.y;
  int block_index_z = blockIdx.z;
  ...
}
```



- Each thread within a given thread bloc have the same value for blockIdx.{x,y,z}

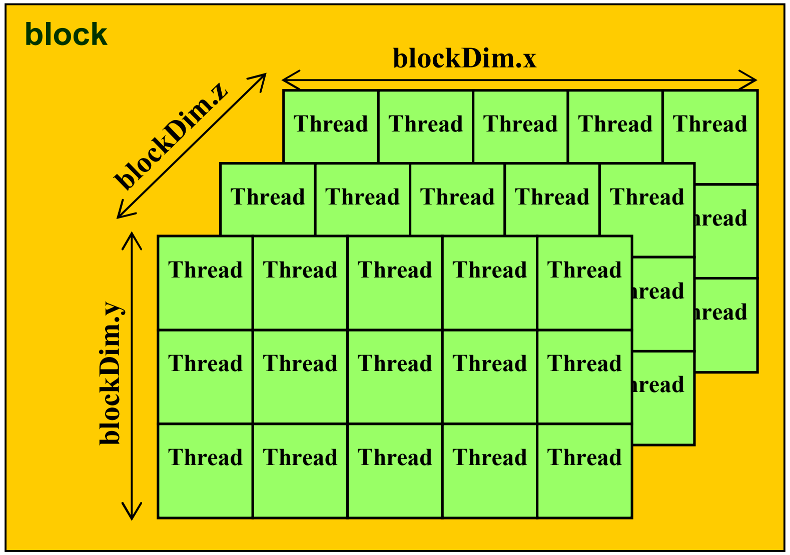
- Threads from different blocks have different values for blockIdx.{x,y,z}

How to Differentiate Threads in a Kernel

- The number of thread in a block can be retrieved inside the kernel through the built-in variables : **blockDim.{x,y,z}**

```
dim3 nbThreads(5,3,3);
dim3 nbBlocks(4,2,2);
mykernel<<< nbBlocks, nbThreads >>>(…)
```

```
__global__ void mykernel(…) {

  int nb_threads_x = blockDim.x;
  int nb_threads_y = blockDim.y;
  int nb_threads_z = blockDim.z;
  …
}
```



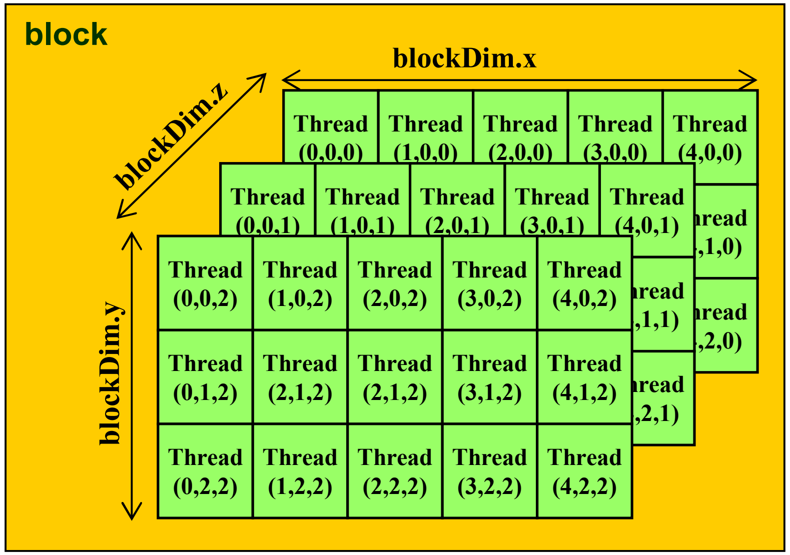
- All the threads executing the kernel have the same values for blockDim.{x,y,z}

How to Differentiate Threads in a Kernel

- The position (index) of thread in a block can be retrieved inside the kernel through the built-in variables : **threadIdx.{x,y,z}**

```
dim3 nbThreads(5,3,3);
dim3 nbBlocks(4,2,2);
mykernel<<< nbBlocks, nbThreads >>>(...)
```

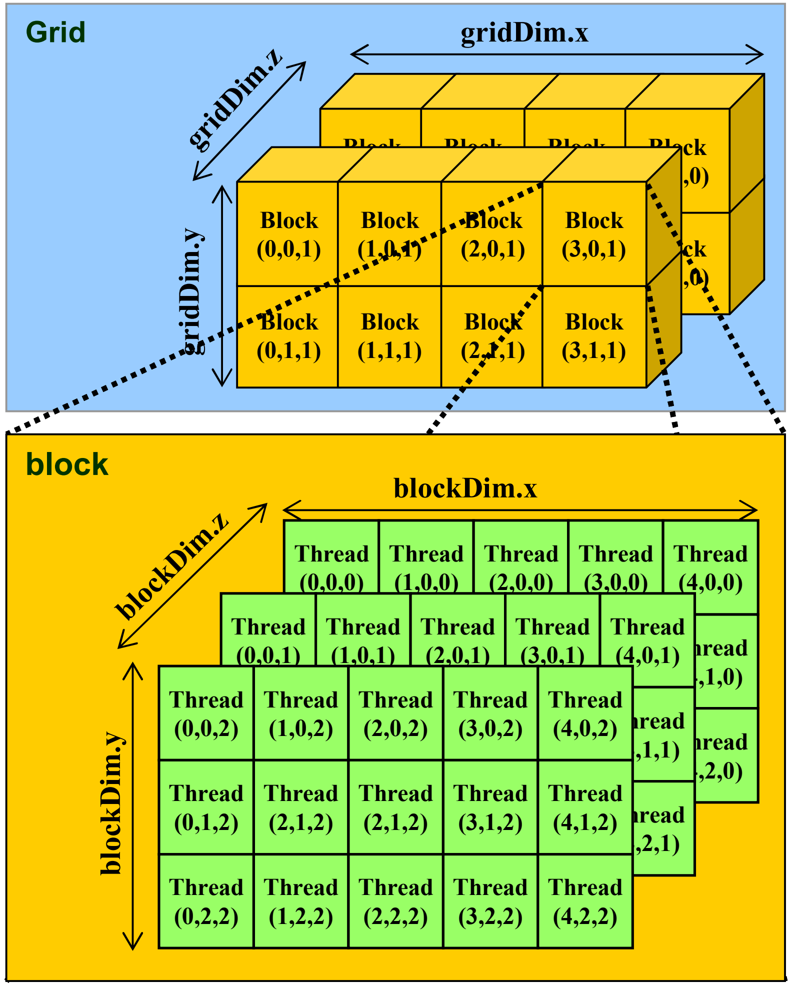
```
__global__ void mykernel(...) {

  int thread_index_x = threadIdx.x;
  int thread_index_y = threadIdx.y;
  int thread_index_z = threadIdx.z;
  ...
}
```



- In a same block, each thread have a unique value for the triplet *(threadIdx.x, threadIdx.y, threadIdx.z)*

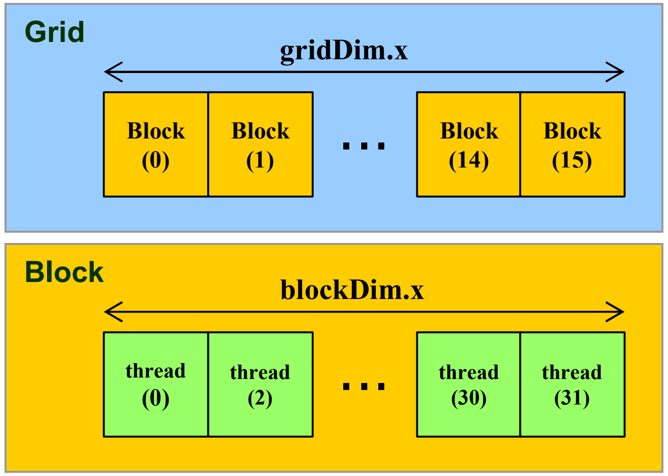
How to Differentiate Threads in a Kernel

- Within a kernel, each thread has access to:
  - Its thread ID :
    threadIdx.x, threadIdx.y, threadIdx.z
  - Its Block ID :
    blockIdx.x, blockIdx.y, blockIdx.z
  - The Block dimensions :
    blockDim.x, blockDim.y, blockDim.z
  - The Grid dimensions :
    gridDim.x, gridDim.y, gridDim.z



Simple 1D grid & 1D blocks

```
__global__ void mykernel(int *array) {

  int index_x = threadIdx.x + blockIdx.x * blockDim.x;

  array[index_x] = index_x;

}


int main(){
  ...
  size_t size = 512;
  int *d_array;
  cudaMalloc((void **)&d_array, size*sizeof(int))
  ...

  dim3 blocksize(32);
  dim3 gridsize(size / blocksize.x);

  mykernel<<<gridsize, blocksize>>>(d_array)
  ...

}
```




```
//results

d_array[0] = 0
d_array[1] = 1
d_array[2] = 2

...

d_array[1022] = 1022
d_array[1023] = 1023
```

LAB: 1D Addition Simple

- Complete the main.cu file to compute a 1D addition.

LAB: 1D Addition

- The size of the array is no longer a multiple of the blocksize:
  - find the formula to have the needed number of blocks in the grid
  - avoid memory overflow

Right Number of Blocks and Protected Memory Accesses

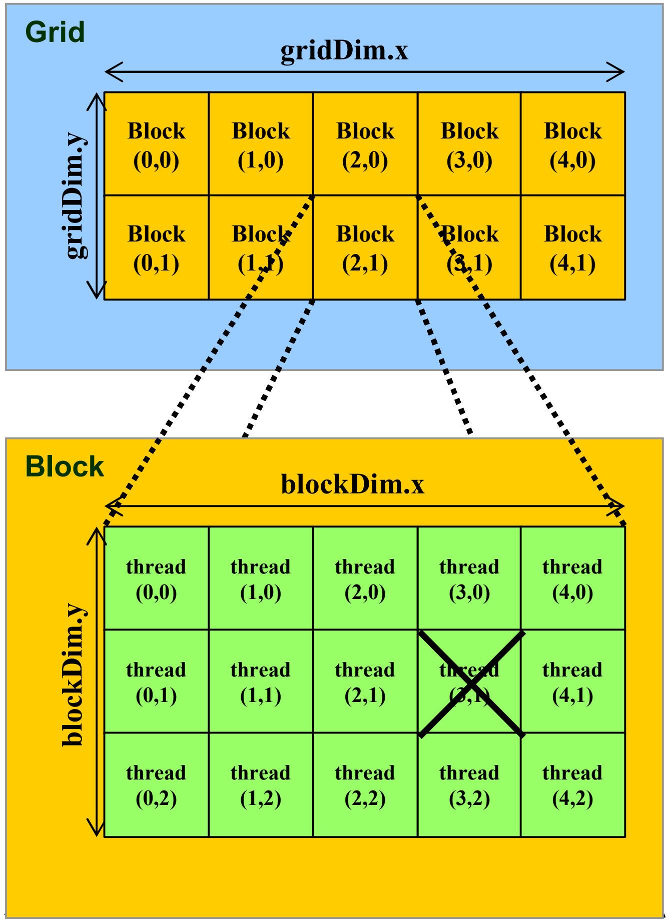
```
__global__ void mykernel(int size, int *array) {

  int index_x = threadIdx.x + blockIdx.x * blockDim.x;

  if(index_x < size)
    array[index_x] = index_x;

}


int main(){
  ...
  size_t size = 512;
  int *d_array;
  cudaMalloc((void **)&d_array, size*sizeof(int))
  ...

  dim3 blocksize(32);
  dim3 gridsize( (size+ blocksize.x-1) / blocksize.x);

  mykernel<<<gridsize, blocksize>>>(d_array)
  ...

}
```

In case there are more threads than elements.
To avoid segmentation faults !

Exemples of 2D Grid Linearization

- 2D THREAD INDEXING

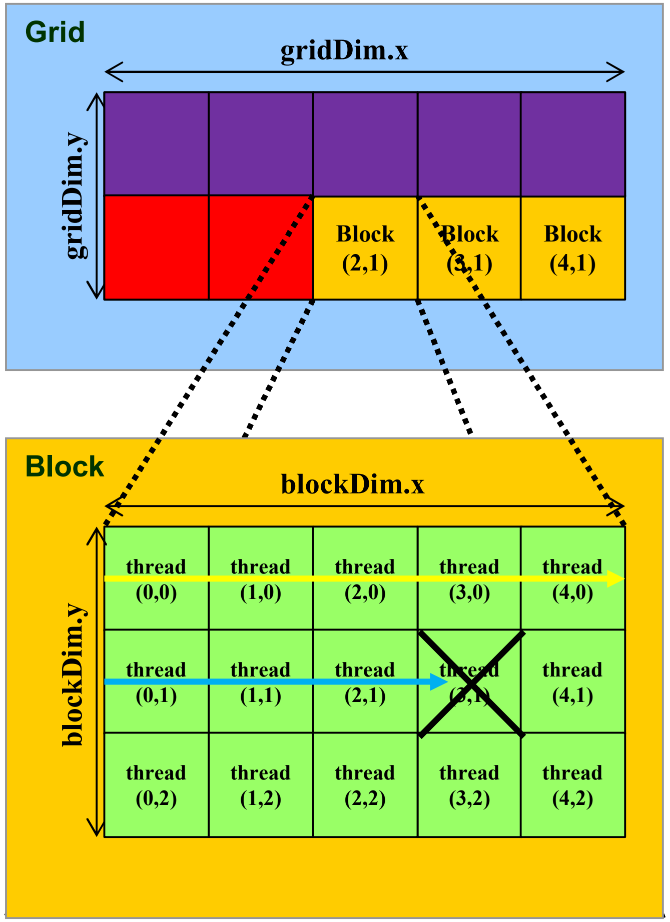
index = ...



Exemples of 2D Grid Linearization

▶ 2D THREAD INDEXING

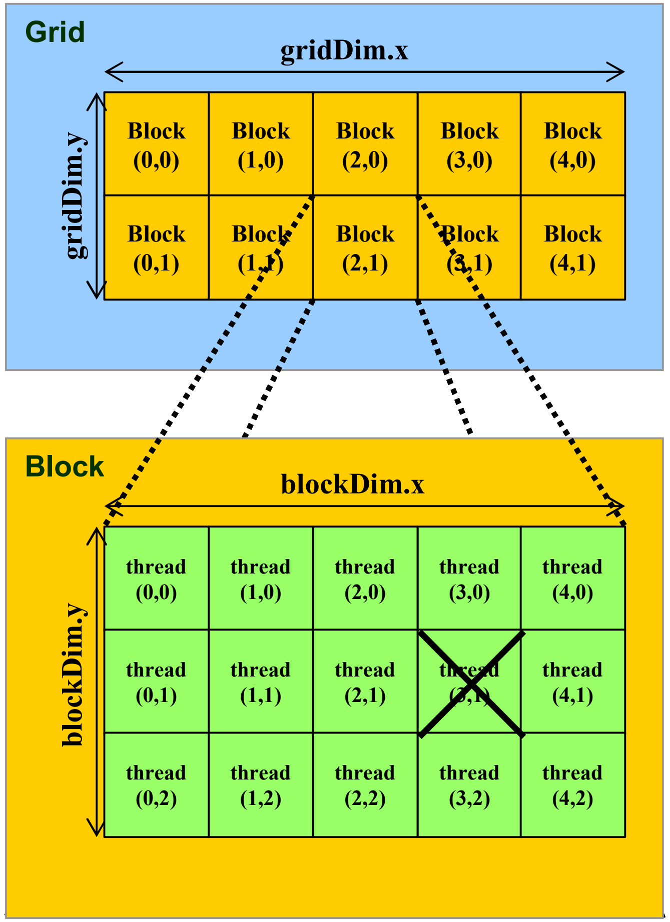
index = threadIdx.x
+ blockDim.x * threadIdx.y
+ blockDim.x * blockDim.y * blockIdx.x
+ blockDim.x * blockDim.y * gridDim.x * blockIdx.y



Exemples of 2D Grid Linearization

- 2D THREAD INDEXING

index = ...

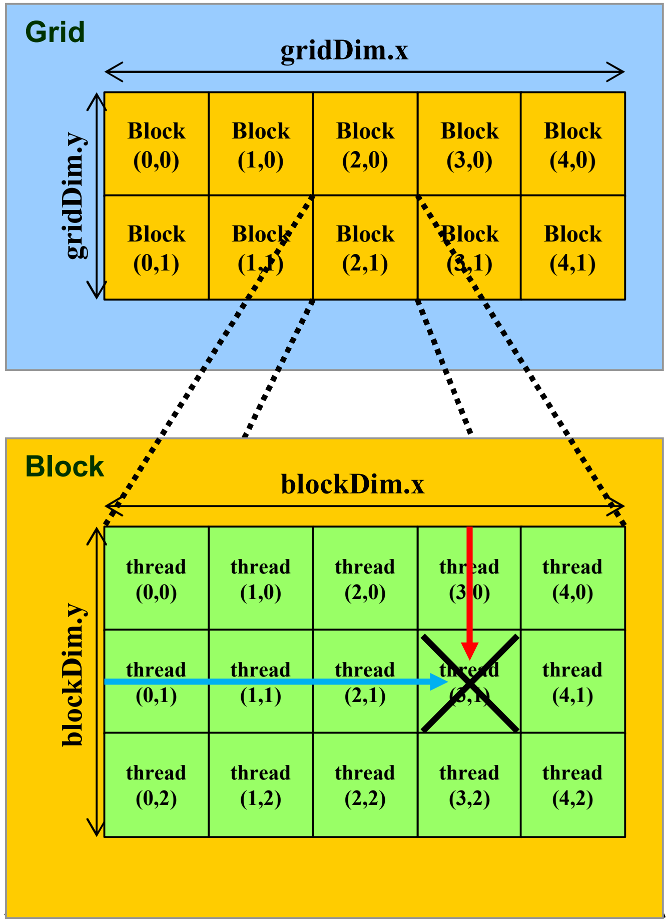


Exemples of 2D Grid Linearization

▸ 2D THREAD INDEXING

index_x = …
index_y = …

index = …

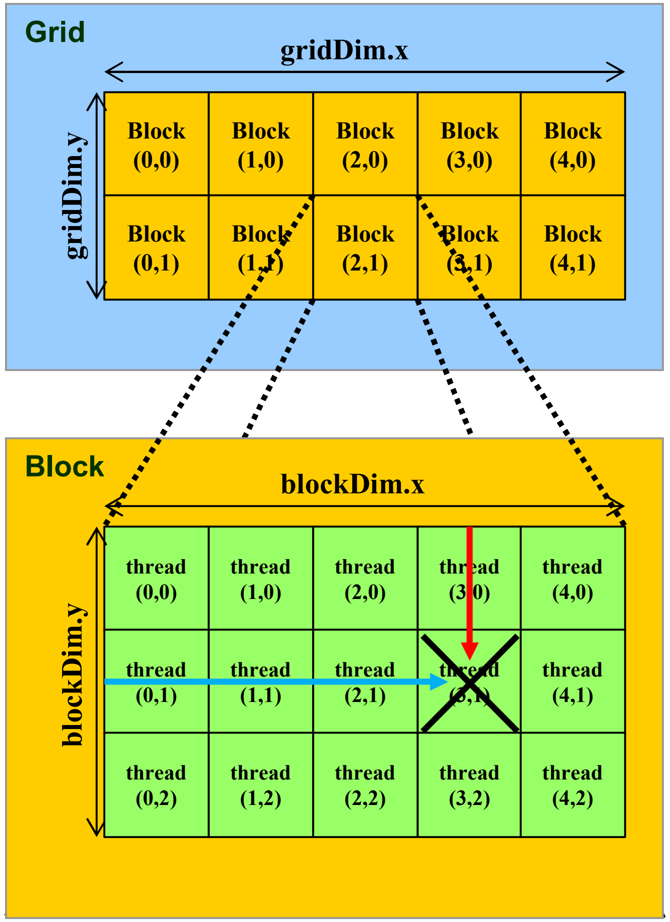


Exemples of 2D Grid Linearization

▶ 2D THREAD INDEXING

index_x = threadIdx.x
index_y = threadIdx.y

index = ...

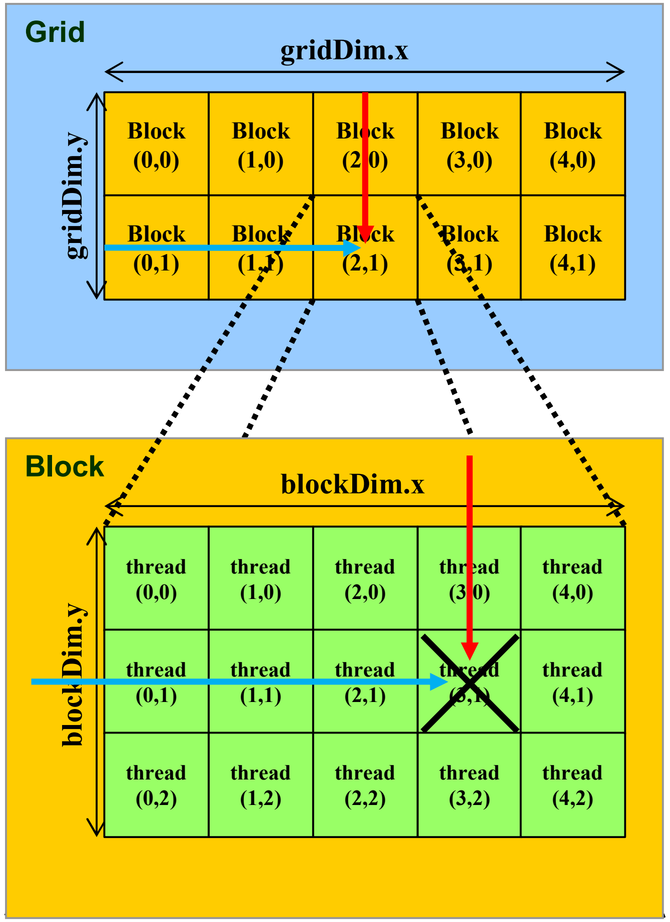


Exemples of 2D Grid Linearization

▶ 2D THREAD INDEXING

index_x = threadIdx.x + …
index_y = threadIdx.y + …

index = …

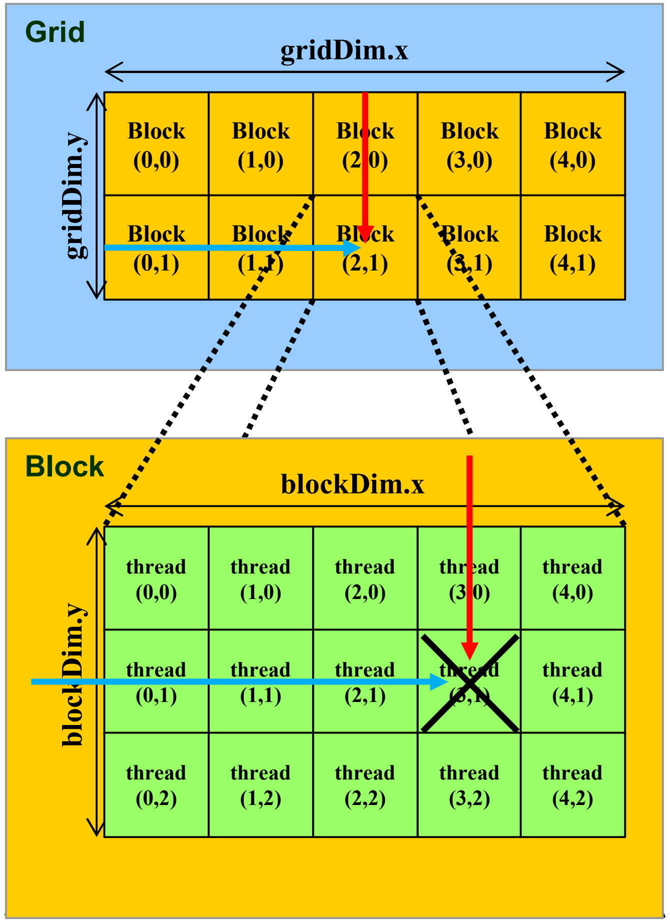


Exemples of 2D Grid Linearization

- 2D THREAD INDEXING

index_x = threadIdx.x + blockIdx.x * blockDim.x
index_y = threadIdx.y + blockIdx.y * blockDim.y

index = …

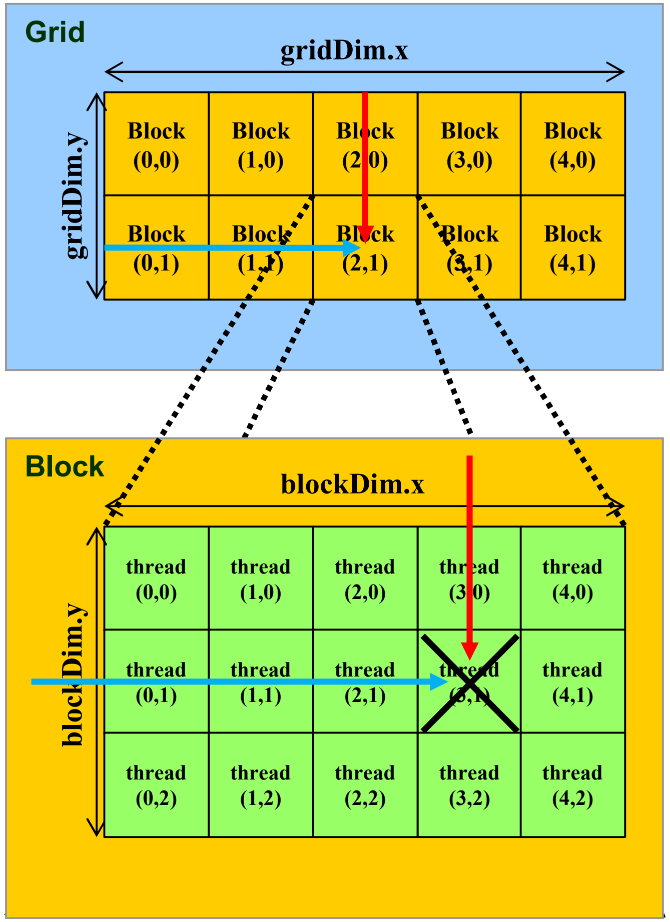


Exemples of 2D Grid Linearization

- 2D THREAD INDEXING

index_x = threadIdx.x + blockIdx.x * blockDim.x
index_y = threadIdx.y + blockIdx.y * blockDim.y

index = index_x + ...



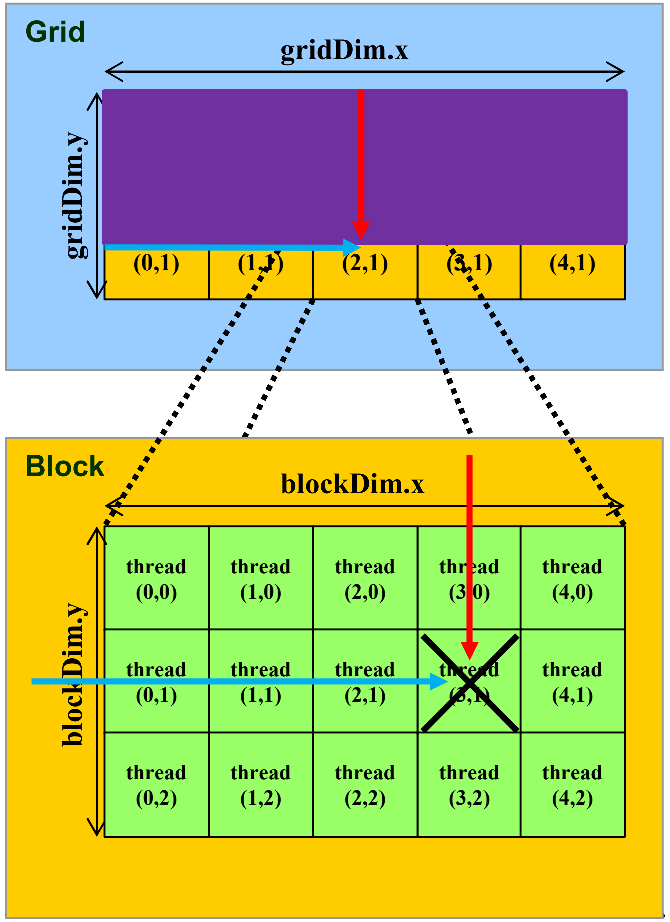
Bull atos technologies

Exemples of 2D Grid Linearization

- 2D THREAD INDEXING

index_x = threadIdx.x + blockIdx.x * blockDim.x
index_y = threadIdx.y + blockIdx.y * blockDim.y

index = index_x + ...

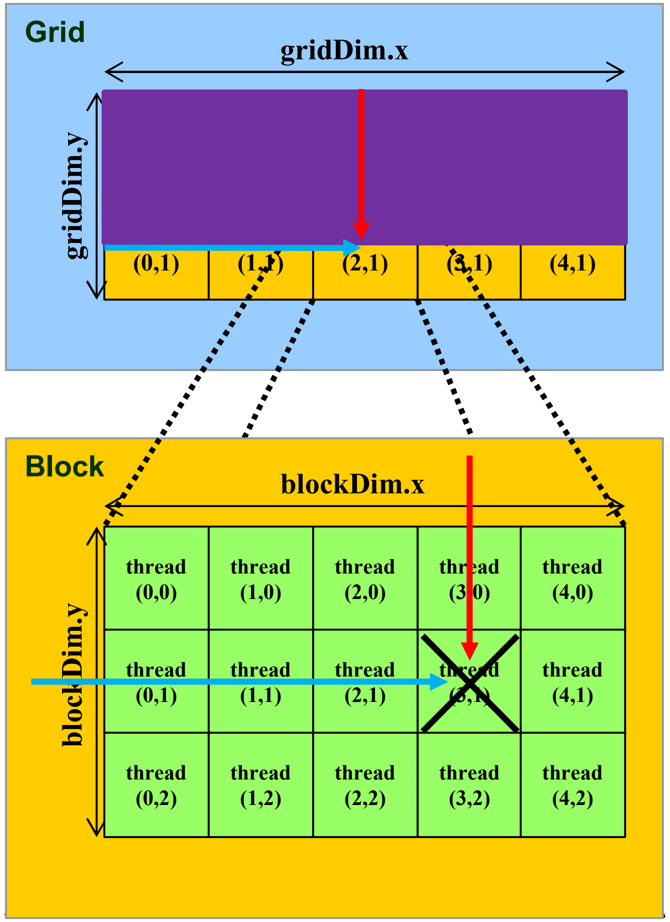


Exemples of 2D Grid Linearization

- 2D THREAD INDEXING

index_x = threadIdx.x + blockIdx.x * blockDim.x
index_y = threadIdx.y + blockIdx.y * blockDim.y

index = index_x + index_y * gridDim.x * blockDim.x

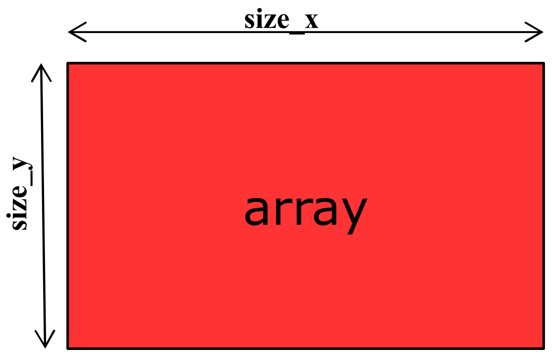


LAB: 2D Addition

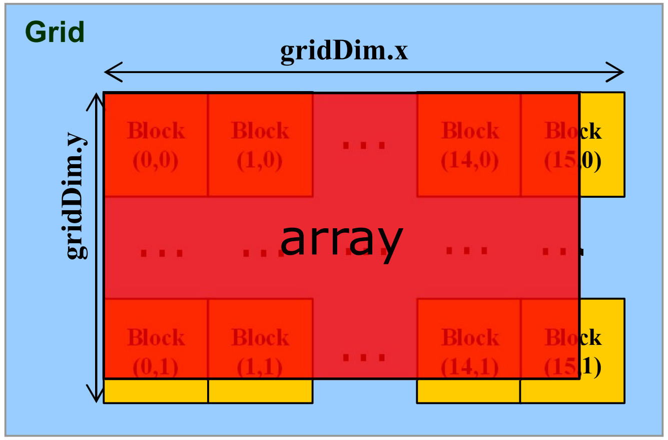
- Complete the main.cu file to compute a 2D addition

Example of 2D Array Accesses

- array:
  - size_x * size_y



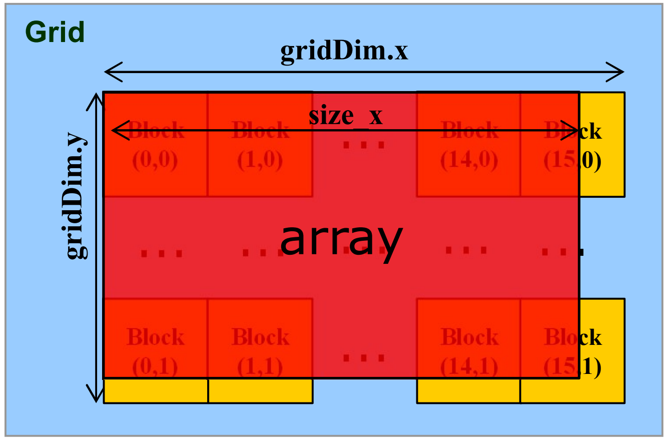
- gridsize:
  - gridsize.x = (size_x + blocksize.x -1) / blocksize.x
  - gridsize.y = (size_y + blocksize.y -1) / blocksize.y

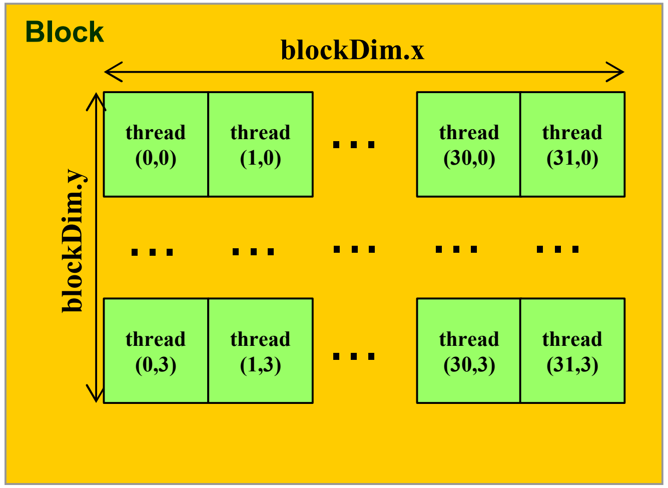


- blocksize:
  - blocksize.x * blocksize.y

Example of 2D Array Accesses

```
__global__ void mykernel(int size_x, int size_, y int *array) {

  int index_x = threadIdx.x + blockIdx.x * blockDim.x;
  int index_y = threadIdx.y + blockIdx.y * blockDim.y;
  int index = index_x + index_y * size_x;

  if(index_x < size_x && index_y < size_y)
    array[index] = index;

}


int main(){
  ...
  size_t size_x = 500;
  size_t size_y = 254;
  int *d_array;
  cudaMalloc((void **)&d_array, size_x*size_y*sizeof(int))
  ...

  dim3 blocksize(32,4);
  dim3 gridsize;
  gridsize.x =( (size_x+ blocksize.x-1) / blocksize.x);
  gridsize.y =( (size_y+ blocksize.y-1) / blocksize.y);
  mykernel<<<gridsize, blocksize>>>(size_x,size_y,d_array)
  ...

}
```



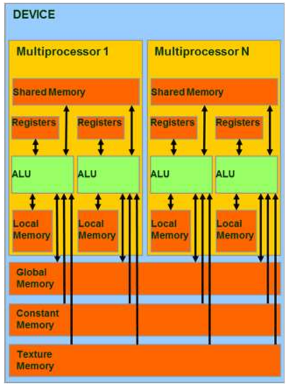


Hardware Execution

16/09/2019

Kernel Execution

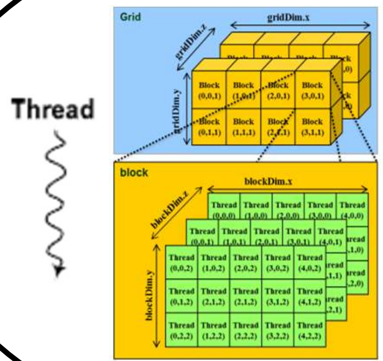
A hardware abstract view



GPU functions

```
__global__ void myFunction( ... )
{

  ...

}
```

Pools of threads



Hardware Execution

- Workload organization (Remainder):
  - The workload is divided into blocks of threads : threadblocks
  - All threadblocks contain the same constant amount of threads
  - CUDA threads are lightweight : small creation & context switch overheard
  - The grid contains all the thread blocks

- Workload execution :
  - A thread block is **executed on** only **one** Streaming Multiprocessor (**SM**)
  - Thread blocks execute **independently from each other**
  - Thread blocks **don’t have a definite order** of execution
  - One SM can run more than one thread block
    - if enough resources are available (occupancy)

- Scalable to all CUDA capable GPUs
  - so a GPU with more SM should execute in less time

CUDA : Hardware Execution

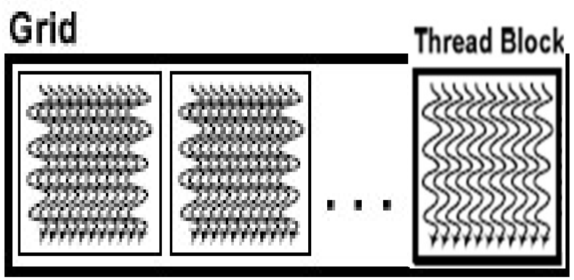
- THREAD BLOCK scheduling
  - Blocks are enumerated and distributed to SMs with available execution capacity
  - When block terminates, new block is launched on vacated SM

- ACTIVE (resident) thread block
  - A block is active until all threads in that block have completed
  - Resources (registers, shared memory) are allocated as long as a block is active
  - Context switching is very fast because resources do not need to be saved and restored

CUDA : Hardware Execution

BlockSize = 1024 threads

612
threadblocks
to schedule



Hardware limitations :
1024 threads /block
2048 threads /SM

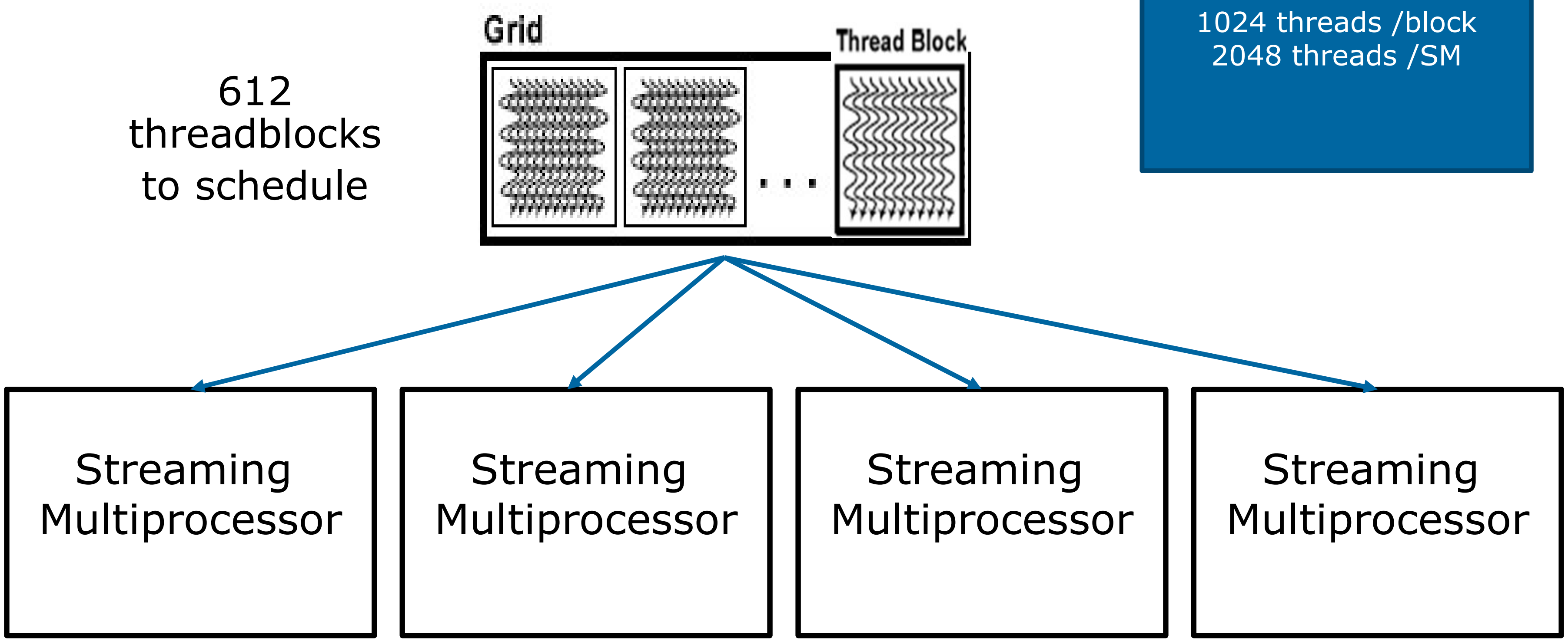
Streaming Multiprocessor	Streaming Multiprocessor	Streaming Multiprocessor	Streaming Multiprocessor

CUDA : Hardware Execution

BlockSize = 1024 threads

Hardware limitations :
1024 threads /block
2048 threads /SM

612
threadblocks
to schedule

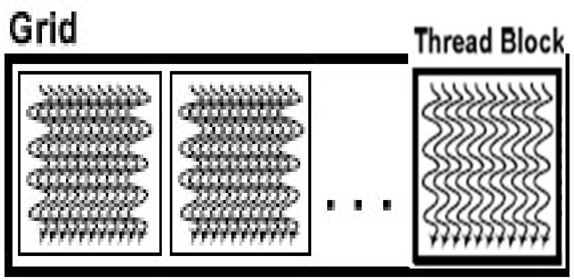


CUDA : Hardware Execution

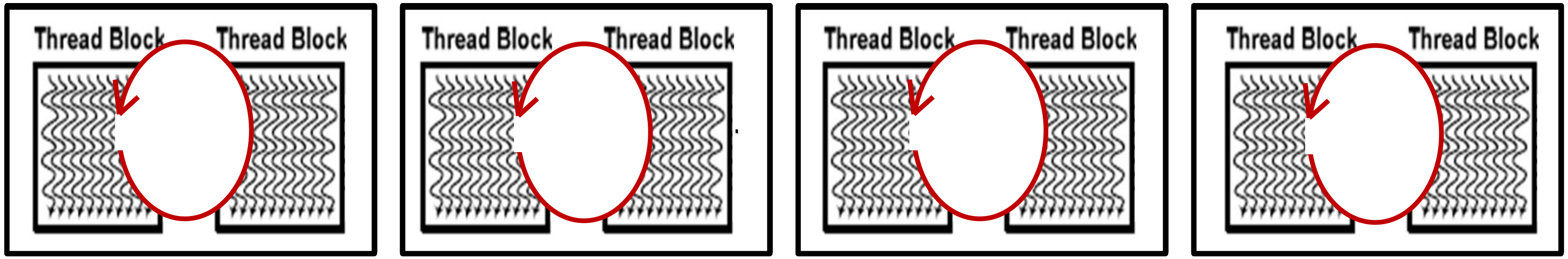
BlockSize = 1024 threads

Hardware limitations :
1024 threads /block
2048 threads /SM

still 604 (612-8)
threadblocks
to schedule

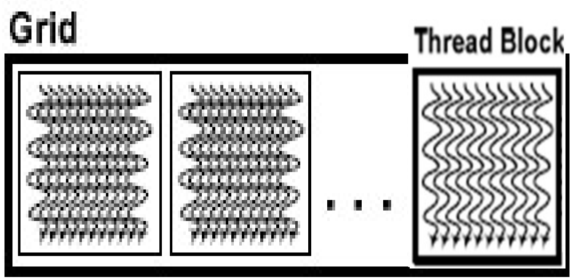


8 threadblocks executing



CUDA : Hardware Execution

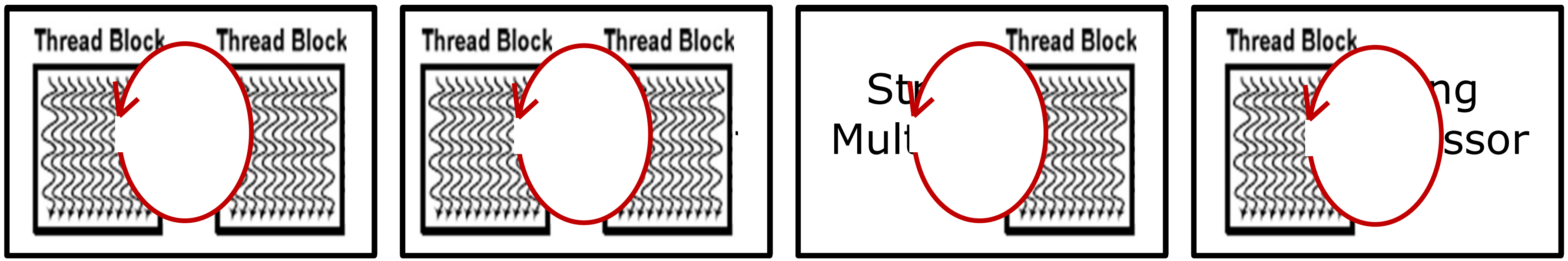
BlockSize = 1024 threads



Hardware limitations :
1024 threads /block
2048 threads /SM

still 604
threadblocks
to schedule

6 threadblocks executing
2 threadblocks executed

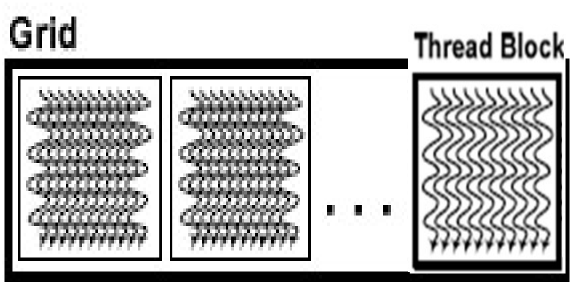


CUDA : Hardware Execution

BlockSize = 1024 threads

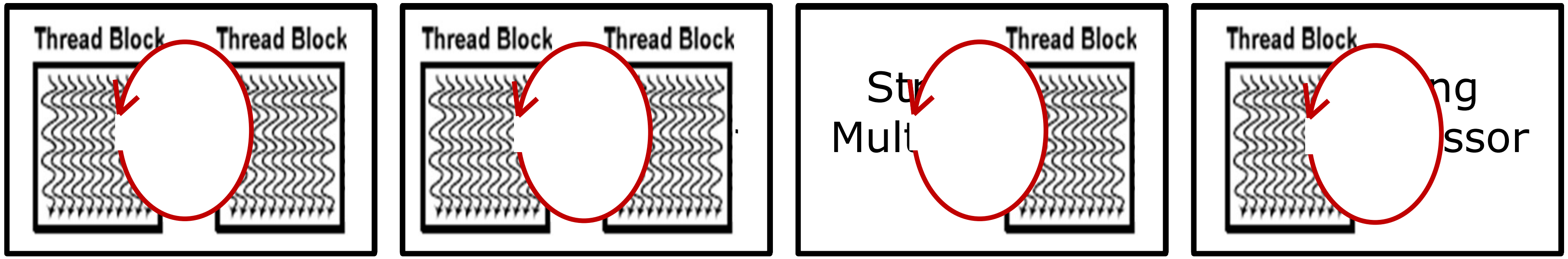
Hardware limitations :
1024 threads /block
2048 threads /SM

still 602
threadblocks
to schedule



8 threadblocks executing

2 threadblocks executed

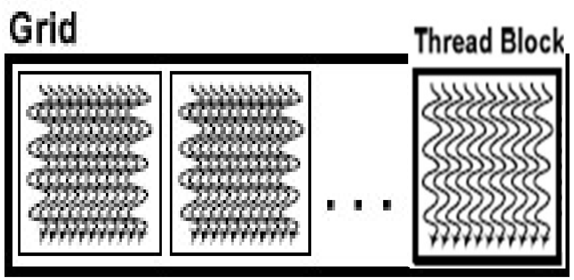


CUDA : Hardware Execution

BlockSize = 1024 threads

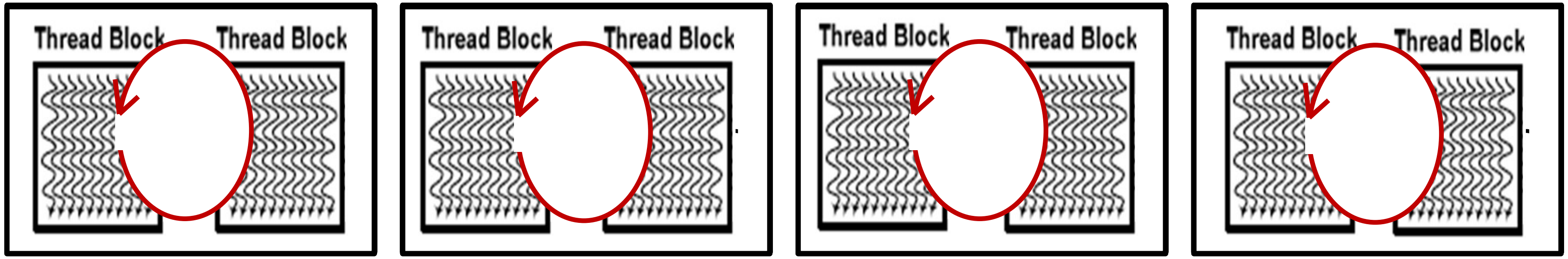
Hardware limitations :
1024 threads /block
2048 threads /SM

still 602
threadblocks
to schedule



8 threadblocks executing

2 threadblocks executed

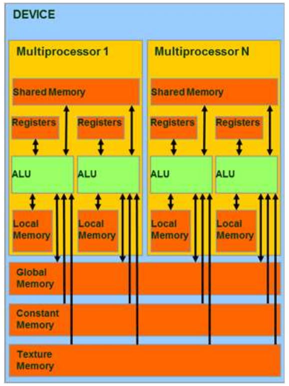


Memory Hierarchy

16/09/2019

Kernel Execution

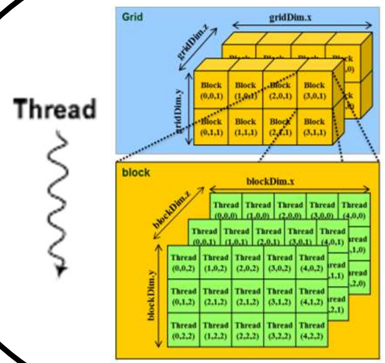
A hardware abstract view



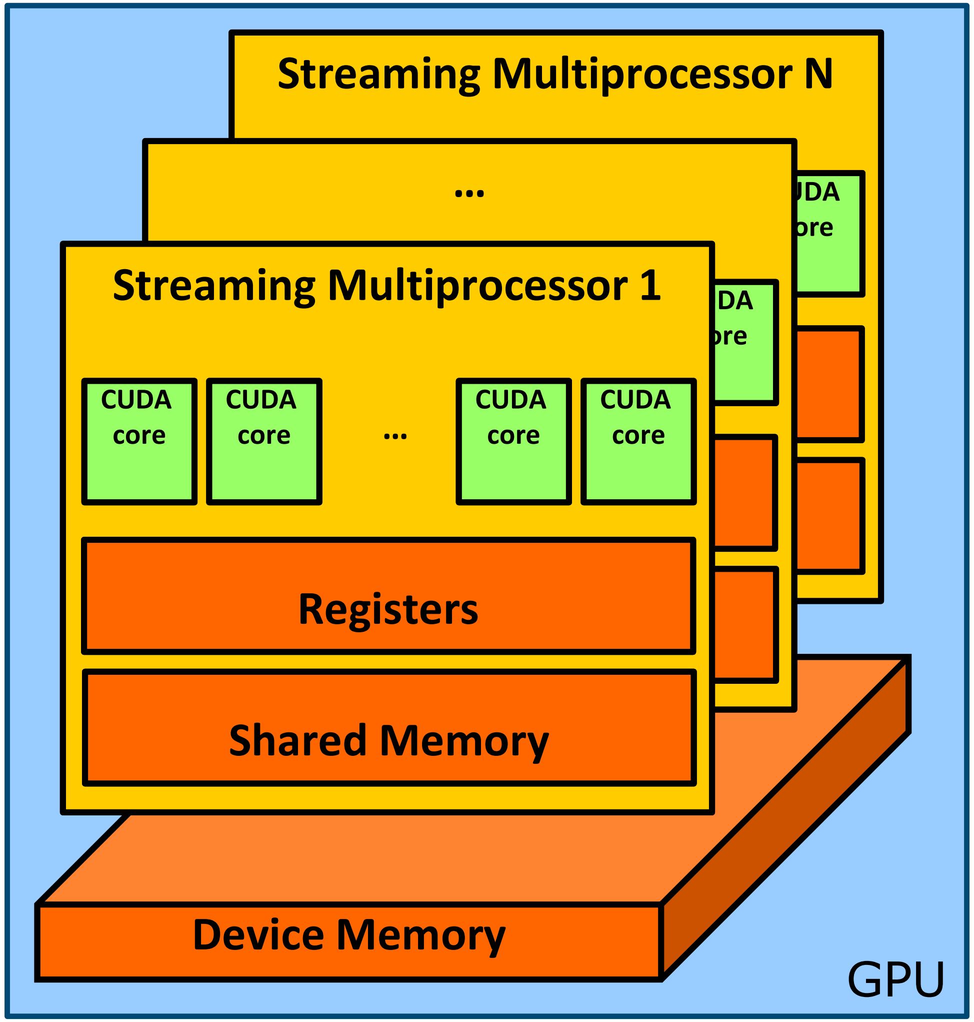
GPU functions

```
__global__ void myFunction( ... )
{

  ...

}
```

Pools of threads

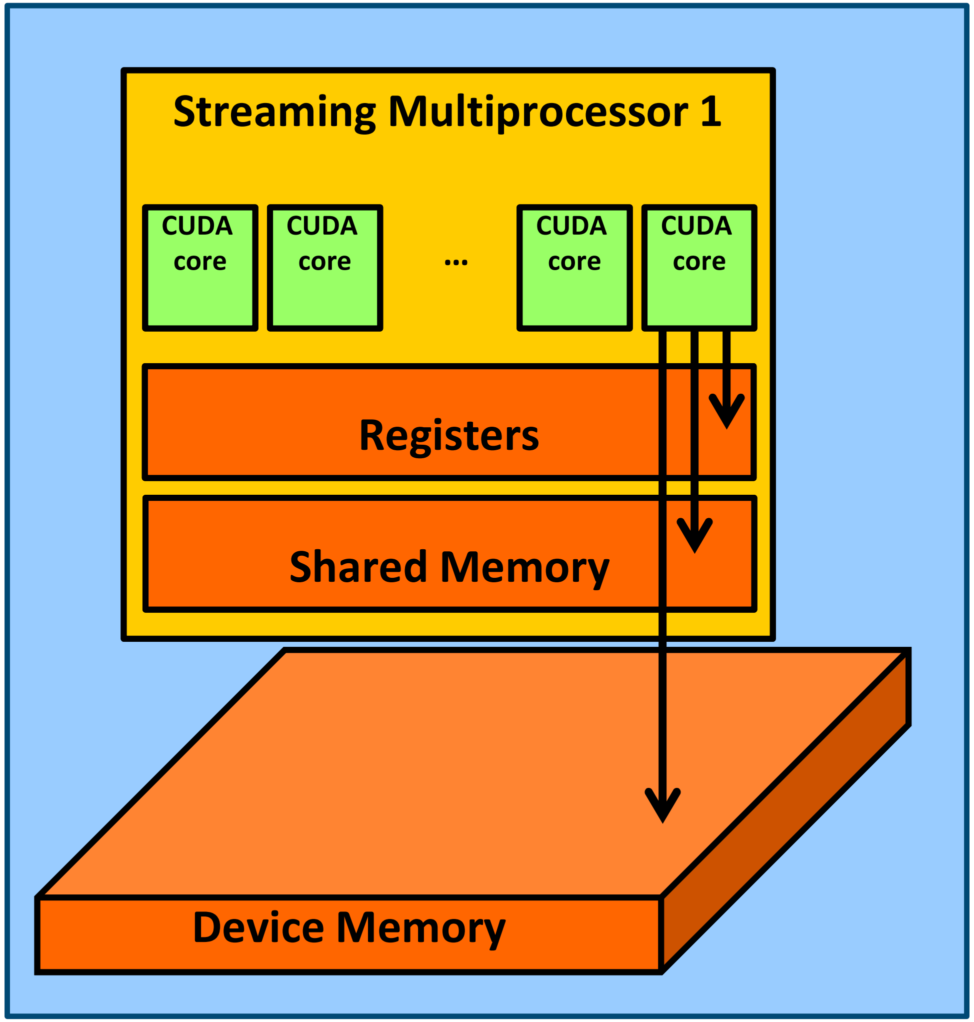


Memory Hierarchy

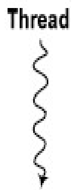


- **Device** contains
  - Multiprocessors
  - Memory
- **Multiprocessors** contains
  - ALUs
  - Registers
  - Shared Memory
  - Access to Local Memory
  - Access to Global Memory

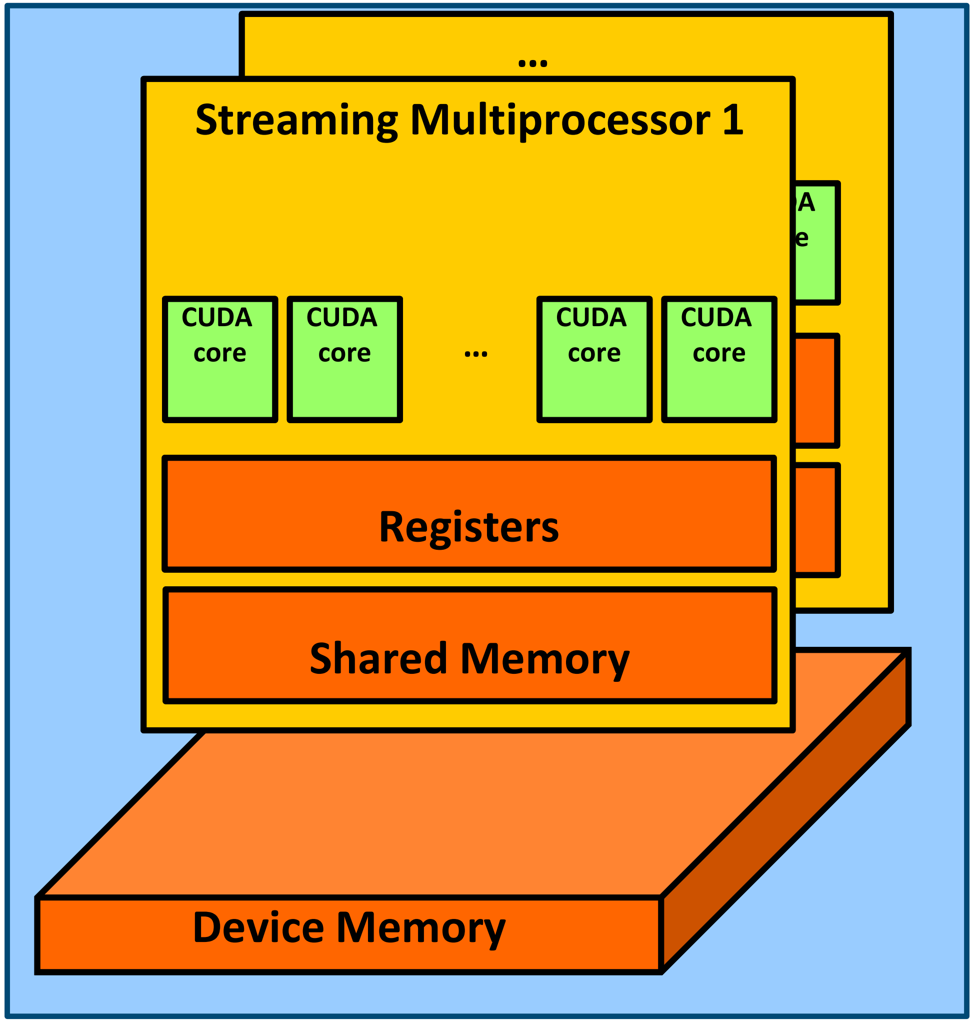
Memory Hierarchy



- A thread runs on 1 CUDA core
- A thread has access to:
  - registers
  - shared memory
  - device memory

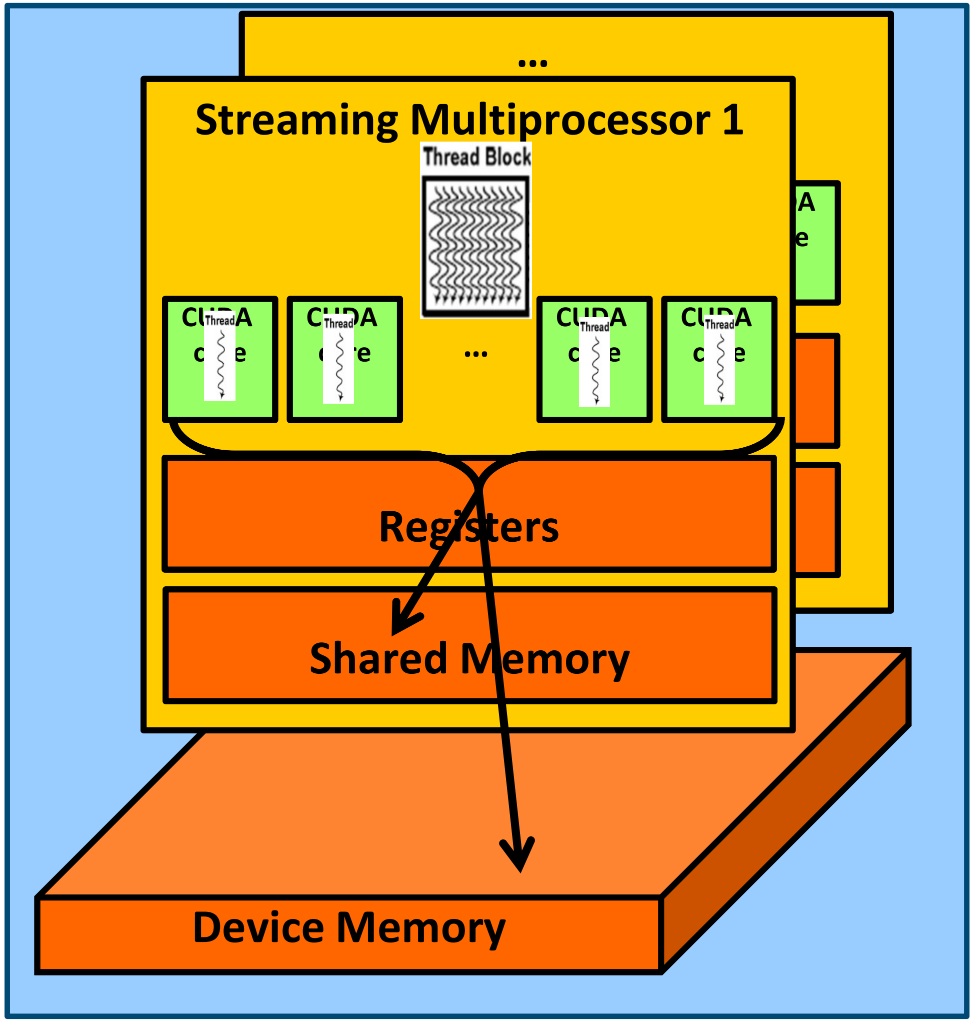


Memory Hierarchy



- A thread block runs on 1 streaming multiprocessor

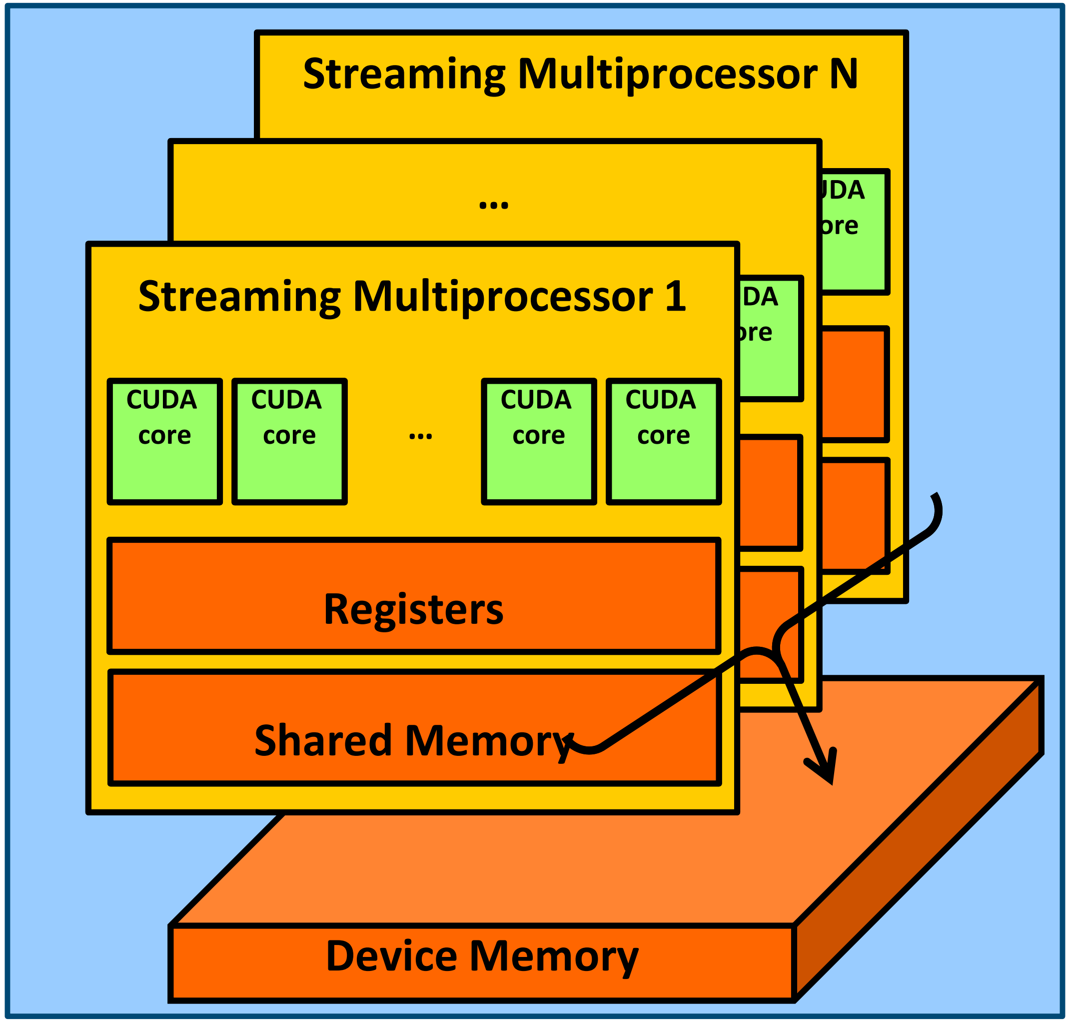
Memory Hierarchy



- A thread block runs on 1 streaming multiprocessor
- Threads in a same block communicate through :
  - shared memory
  - device memory

registers and shared memory data lifetime
=
thread block lifetime

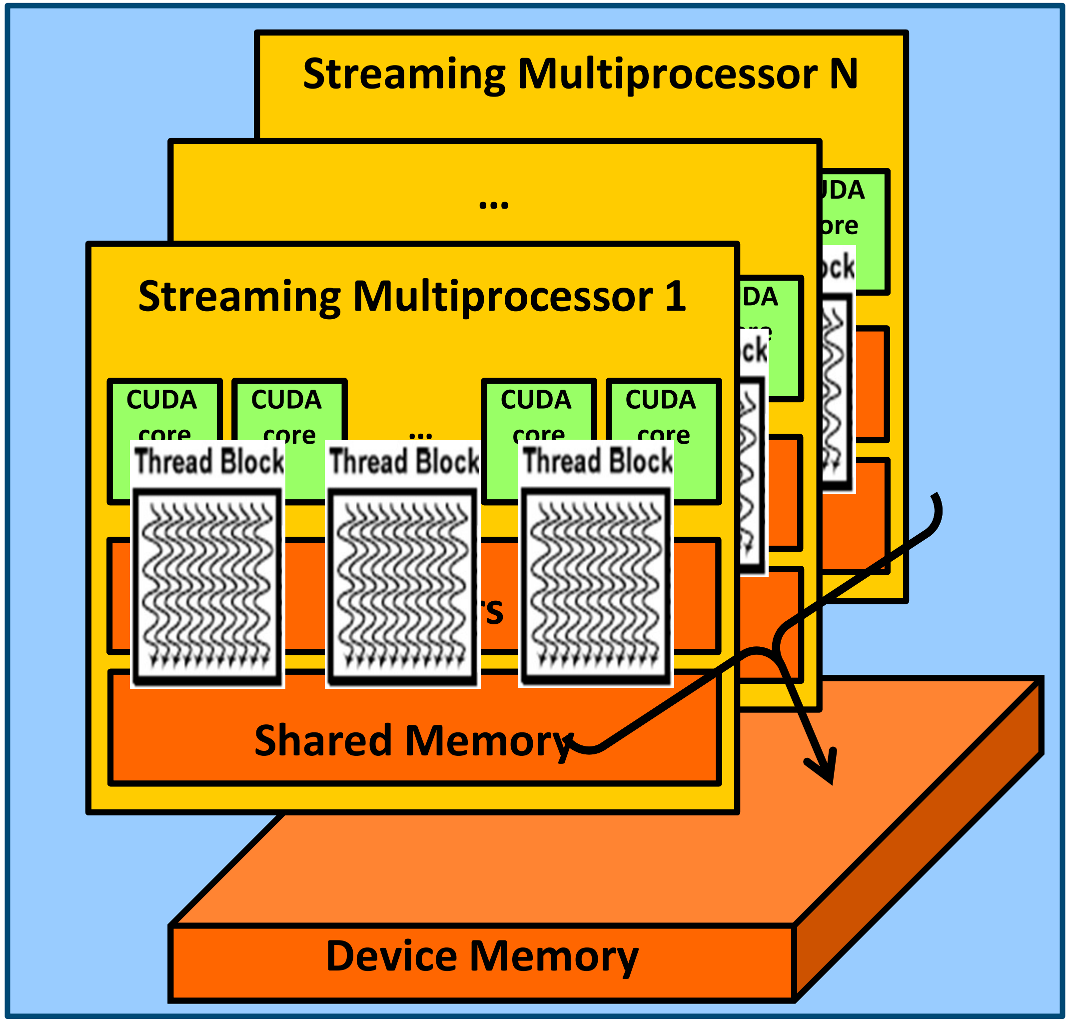
Memory Hierarchy



- A multiprocessor can run multiple thread block
- Threads in different blocks communicate through :
  - device memory

device memory data lifetime
=
GPU thread (context) lifetime

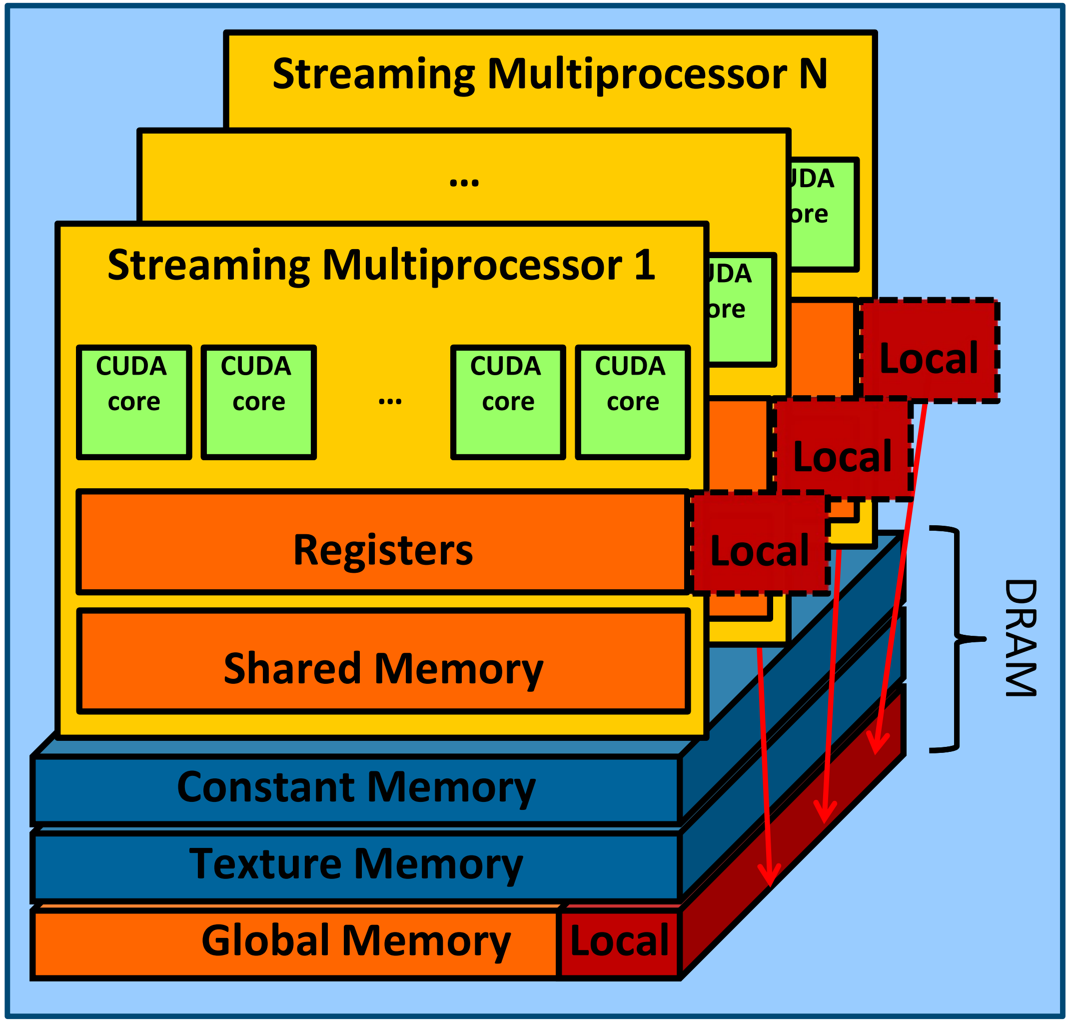
Memory Hierarchy



- A multiprocessor can run multiple thread block
- Threads in different blocks communicate through :
  - device memory

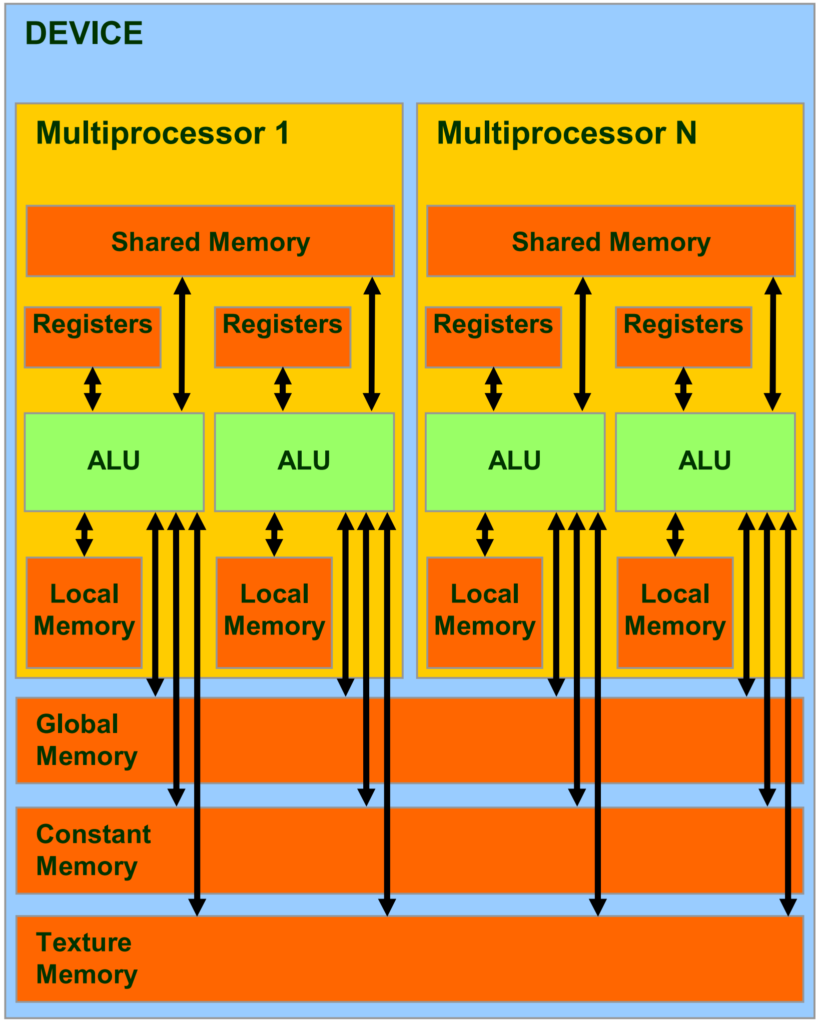
device memory data lifetime
=
GPU thread (context) lifetime

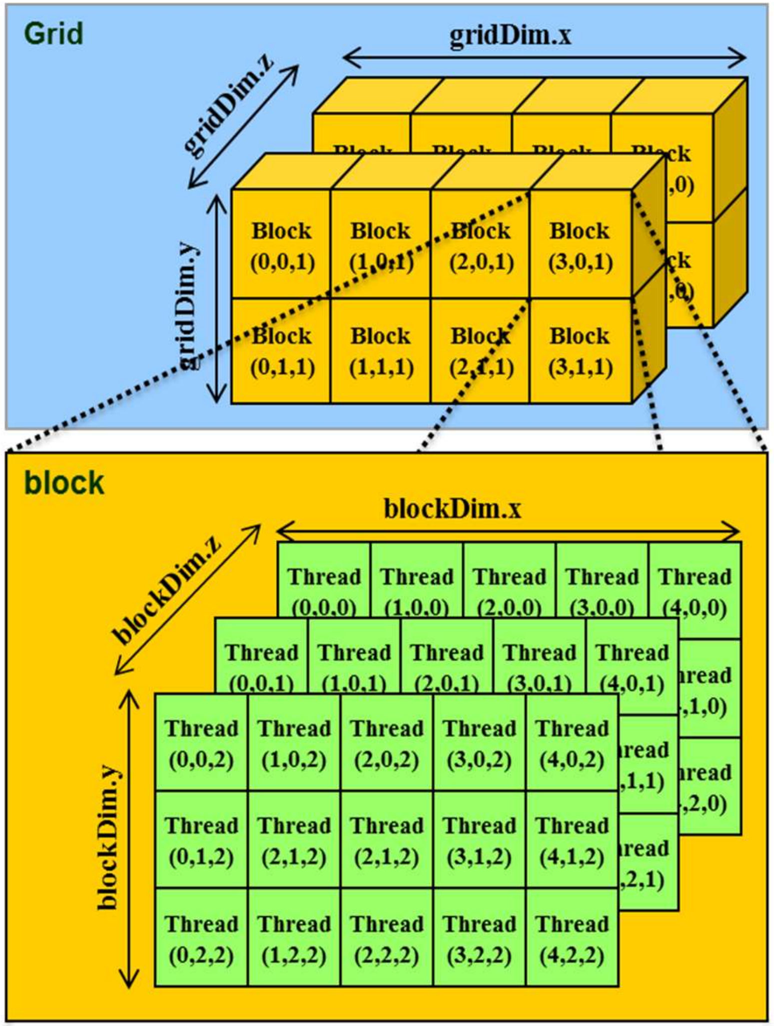
Memory Hierarchy



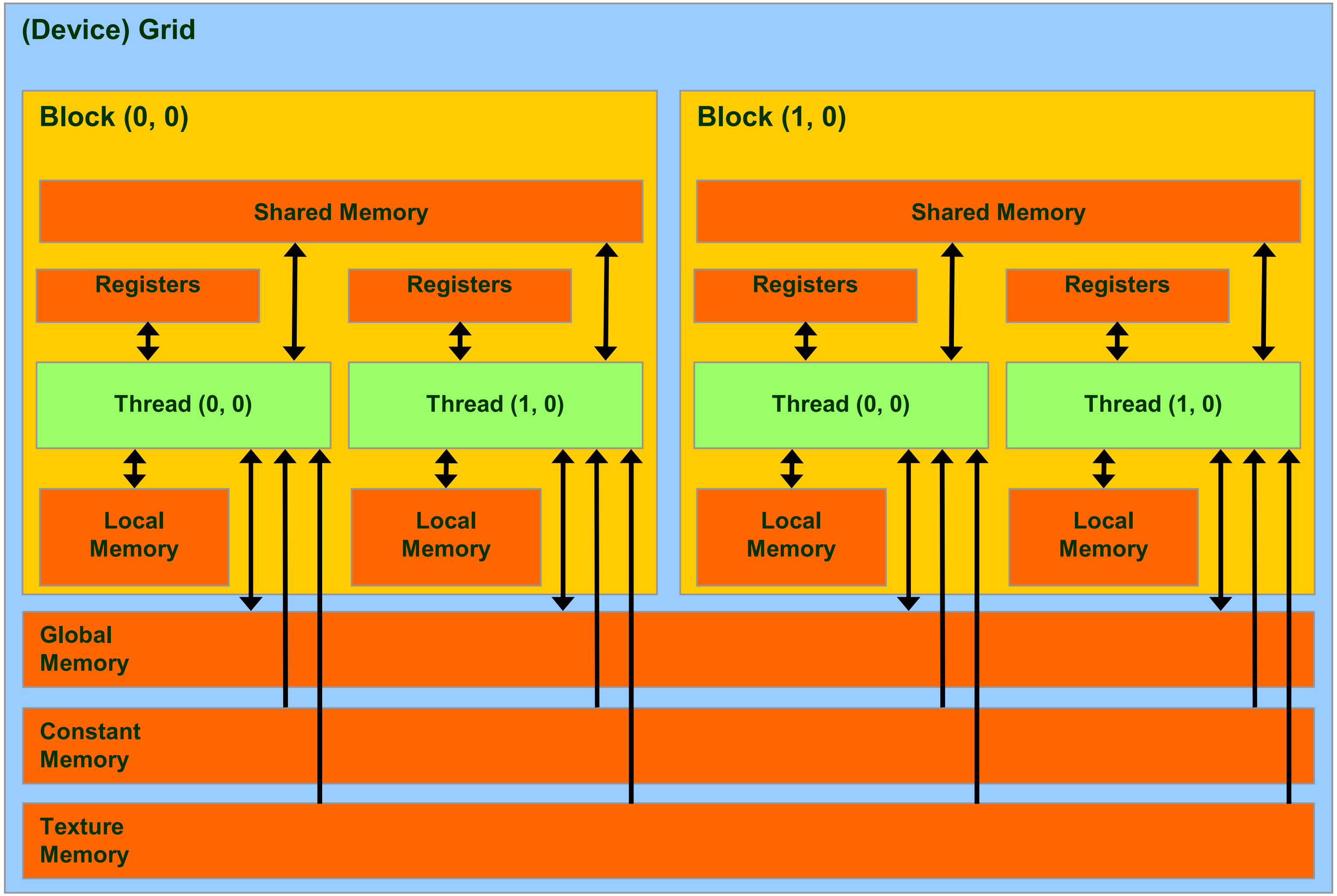
- Read-Only Cached Memory
  - Constant Memory
  - Texture Memory
- Logical Memory Space
  - Local Memory :
    Registers spilled to Global Memory

Memory Hierarchy





Memory Hierarchy



Copyright

Thanks

For more information please contact:

Georges-Emmanuel Moulard
M+ 33 6 85529054 georges-emmanuel.moulard@atos.net

Paul Karlshöfer
paul.Karlshoefer@atos.net

16/09/2019