UNIVERSITÀ
DEGLI STUDI
FIRENZE

Parallel Programming

Prof. Marco Bertini

UNIVERSITÀ
DEGLI STUDI
FIRENZE

Data parallelism: GPU computing

UNIVERSITÀ
DEGLI STUDI
FIRENZE

CUDA

CUDA: Compute Unified Device Architecture

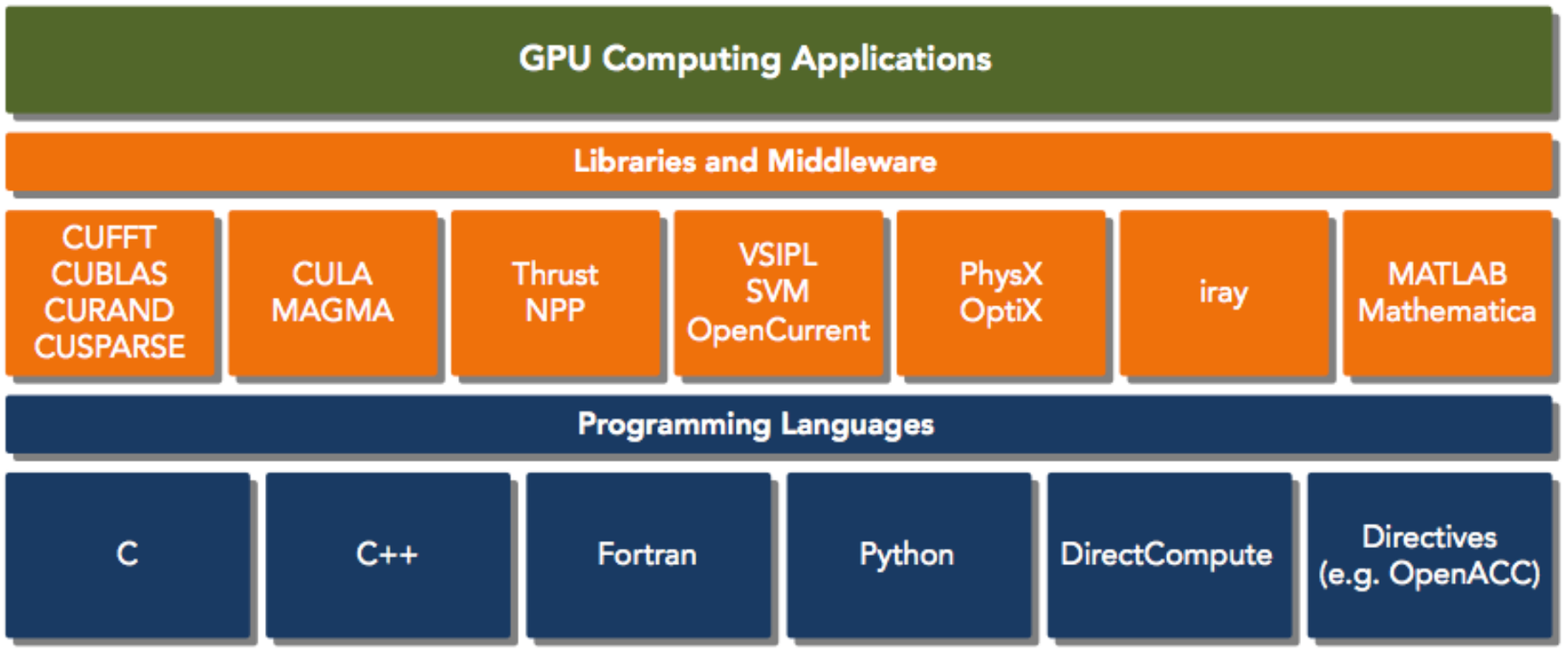
- It enables a general purpose programming model on NVIDIA GPUs. Current CUDA SDK is 12.6.

- Enables explicit GPU memory management
  - The GPU is viewed as a compute **device** that:
    - Is a co-processor to the CPU (or **host**)
    - Has its own DRAM (global memory in CUDA parlance)
    - Runs many threads in parallel

The CUDA platform

- The CUDA platform is accessible through CUDA-accelerated libraries, compiler directives, application programming interfaces, and extensions to industry-standard programming languages, including C, C++, Fortran, and Python

- CUDA C is an extension of standard ANSI C with a handful of language extensions to enable heterogeneous programming, and also straightforward APIs to manage devices, memory, and other tasks.



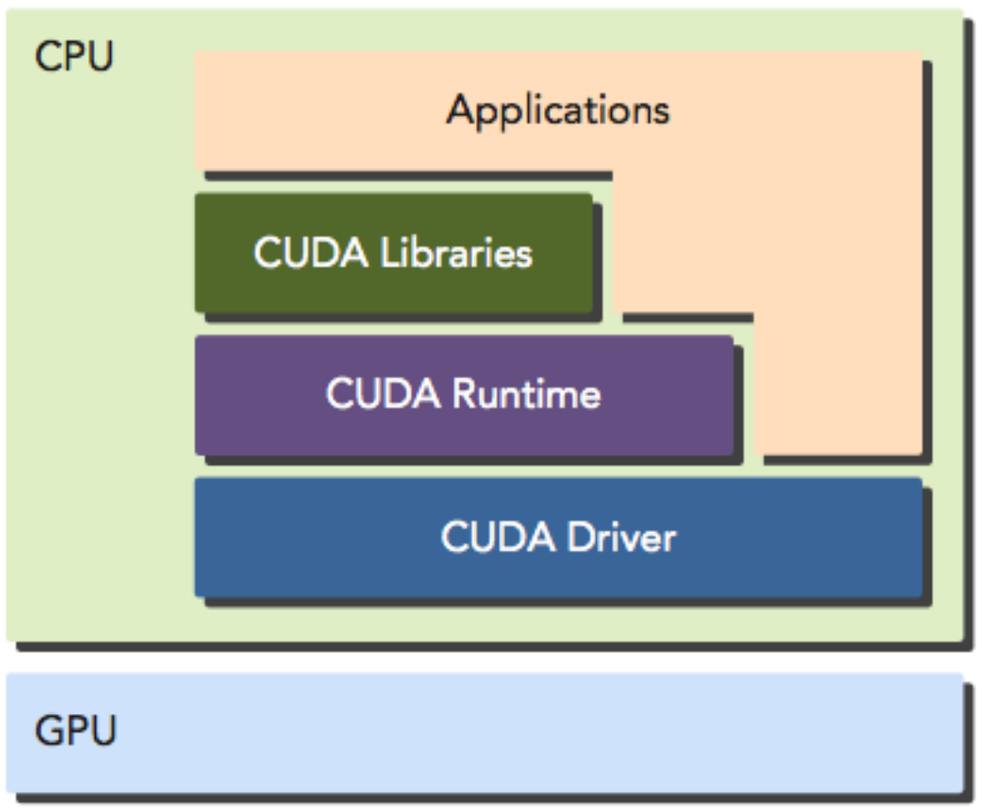
CUDA and C++11

- Since CUDA 7.x it is possible to use modern C++11 features also in the device code:
  - Lambda
  - Auto (useful when using template libraries like Thrust)
  - Range-based for loops
  - Rvalues and move semantics

CUDA APIs

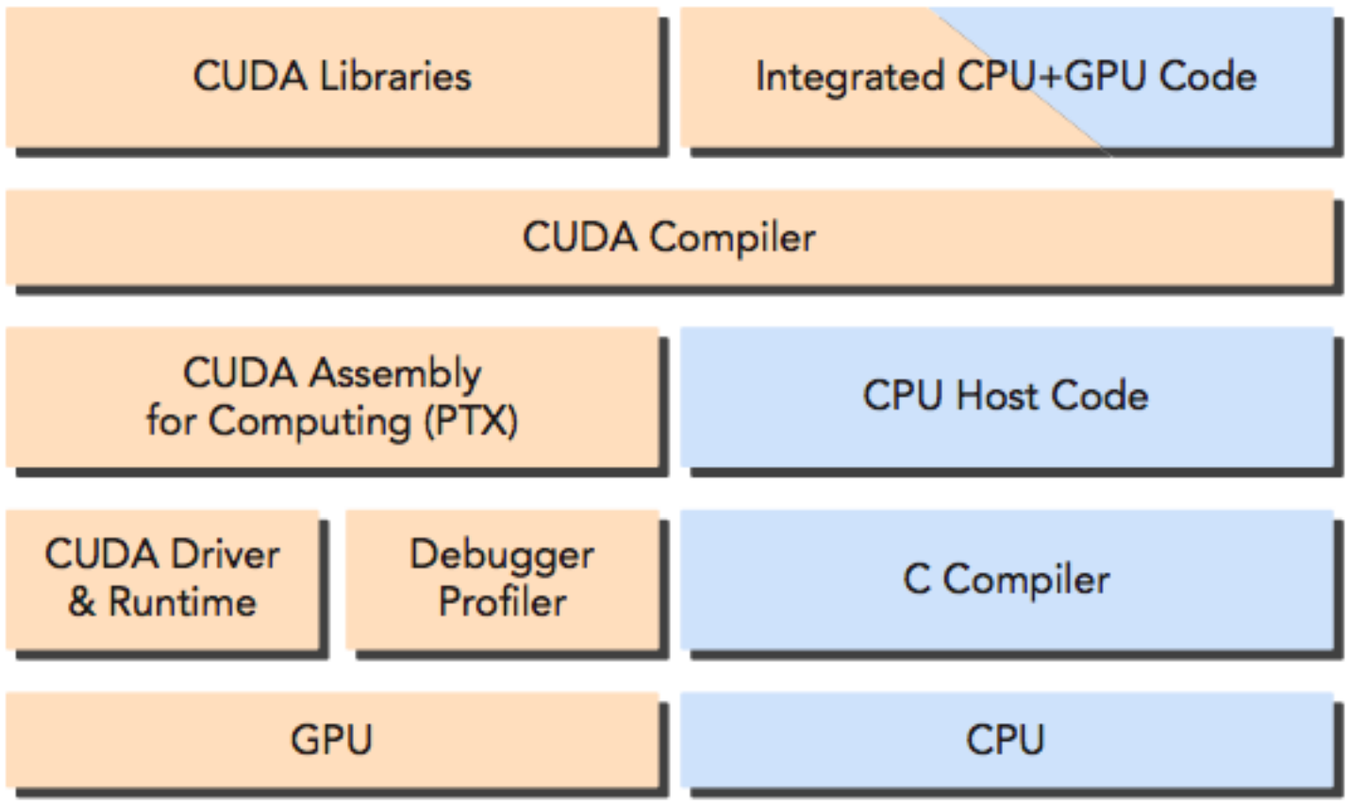
- CUDA provides two API levels for managing the GPU device and organizing threads:
  - CUDA Driver API
  - CUDA Runtime API
- The driver API is a low-level API and is relatively hard to program, but it provides more control over how the GPU device is used.

The runtime API is a higher-level API implemented on top of the driver API. Each function of the runtime API is broken down into more basic operations issued to the driver API.



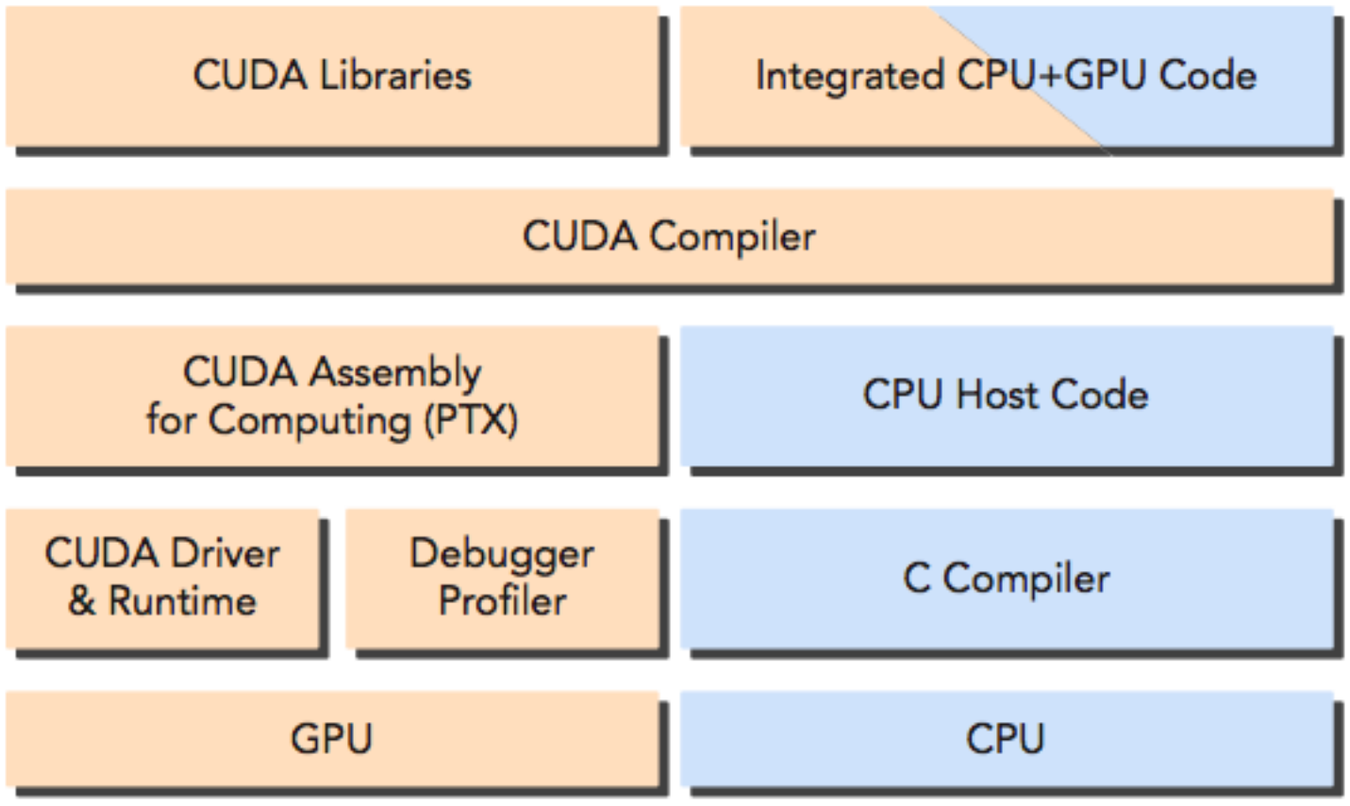
A CUDA program

- A CUDA program consists of a mixture of the following two parts:
  - The host code runs on CPU.
  - The device code runs on GPU.
- NVIDIA's CUDA nvcc compiler separates the device code from the host code during the compilation process.

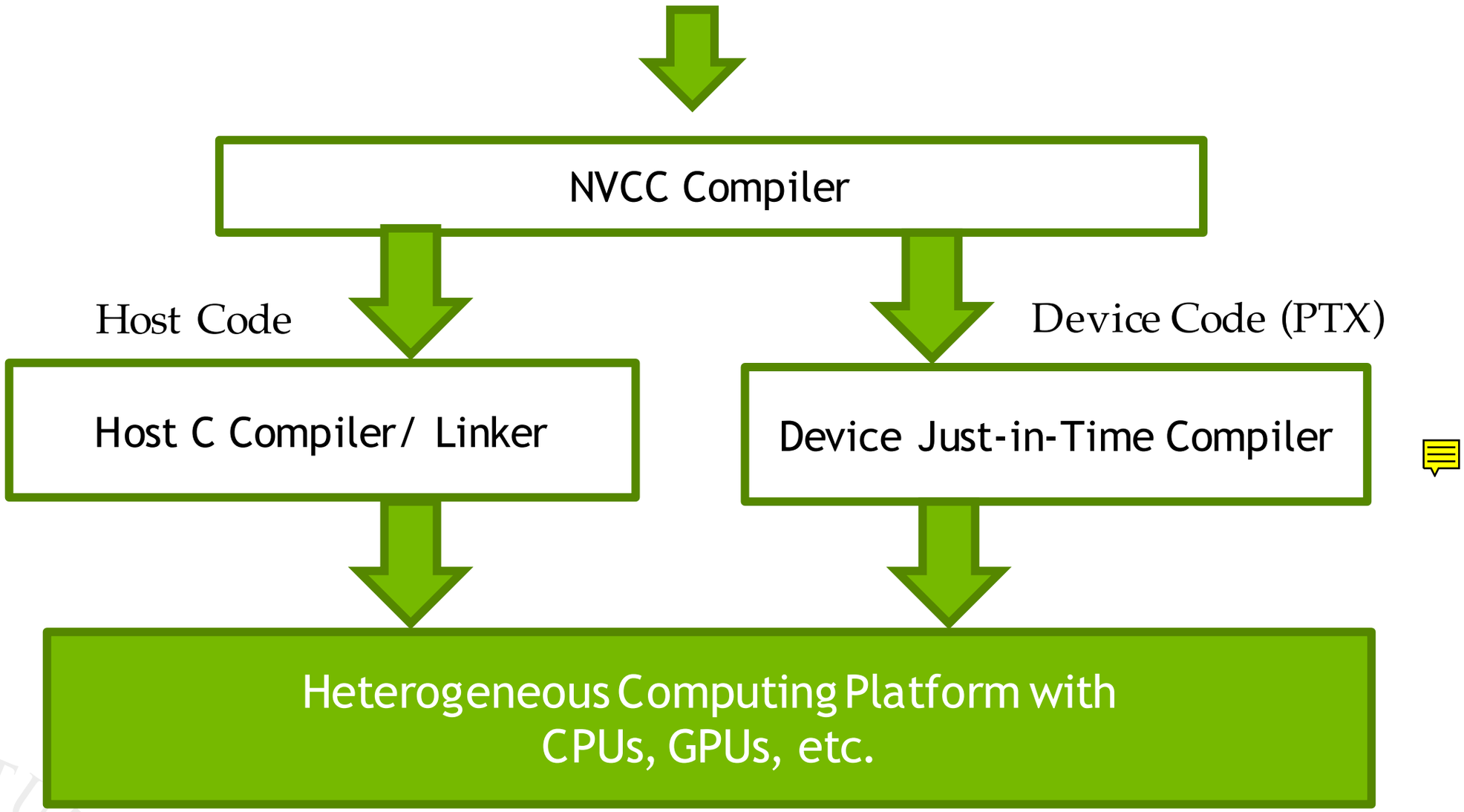


A CUDA program

- A CUDA program consists of a mixture of the following two parts:
  - The host code runs on CPU.
  - The device code runs on GPU.
- NVIDIA provides the NSight IDE (based on Eclipse) to ease development of C/C++/CUDA programming



Compiling a CUDA program



PTX and JIT

- The CUDA compiler driver `nvcc`, uses a two-stage compilation model:

1. compiles source device code to PTX virtual assembly

2. compiles the PTX to binary code for the target architecture.

But the CUDA driver can execute the second stage compilation at run time, compiling the PTX virtual assembly “Just In Time” to run it.

Fat binary

- To avoid the JIT cost at runtime it is possible to include the binary code for one or more architectures in the application binary along with PTX code.

- The CUDA run time looks for code for the present GPU architecture in the binary, and runs it if found. If binary code is not found but PTX is available, then the driver compiles the PTX code.

- In this way deployed CUDA applications can support new GPUs when they come out.

JIT caching

- Another approach to mitigate JIT overhead is to cache the binaries generated by JIT compilation.

- When the device driver just-in-time compiles PTX code for an application, it automatically caches a copy of the generated binary code to avoid repeating the compilation in later invocations of the application.

- The cache is automatically invalidated when the device driver is upgraded.

  - `CUDA_CACHE_PATH` specifies the directory location of compute cache files

What a programmer expresses in CUDA

- Computation partitioning (where does computation occur?)
  - Declarations on functions `__host__`, `__global__`, `__device__`
  - Mapping of thread programs to device: `compute <<<gs, bs>>>(<args>)`
- Data partitioning (where does data reside, who may access it and how?)
  - Declarations on data `__shared__`, `__device__`, `__constant__`, …
- Data management and orchestration
  - Copying to/from host: e.g., `cudaMemcpy(h_obj,d_obj, cudaMemcpyDevicetoHost)`
- Concurrency management
  - E.g. `__synchthreads()`

CUDA C: C extension + API

- Declspecs
  - global, device, shared, local, constant
- Keywords
  - threadIdx, blockIdx
- Intrinsics
  - __syncthreads
- Runtime API
  - Memory, symbol, execution management
- Function launch

```
__device__ float filter[N];

__global__ void convolve (float *image)  {

  __shared__ float region[M];
  ...

region[threadIdx] = image[i];

  __syncthreads();
  ...

  image[j] = result;
}

// Allocate GPU memory
void *myimage = cudaMalloc(bytes)


// 100 blocks, 10 threads per block
convolve<<<100, 10>>> (myimage);
```

Languages

- The host code is written in ANSI C, and the device code is written using CUDA C.
- You can put all the code in a single source file, or you can use multiple source files to build your application or libraries.
- The NVIDIA C Compiler (`nvcc`) generates the executable code for both the host and device.
- Typical CUDA C extension is `.cu`

CUDA program structure

- A typical CUDA program structure consists of five main steps:

1. Allocate GPU memories.
2. Copy data from CPU memory to GPU memory.
3. Invoke the CUDA functions (called **kernel**) to perform program-specific computation.
4. Copy data back from GPU memory to CPU memory.
5. Destroy GPU memories.

CUDA program structure

- A typical CUDA program structure consists of five main steps:

1. Allocate GPU memories.
2. Copy data from CPU memory to GPU memory.

As the developer, you can express a **kernel** as a sequential program. Behind the scenes, CUDA manages scheduling programmer-written kernels on GPU threads.

From the host, you define how your algorithm is mapped to the device based on application data and GPU device capability.

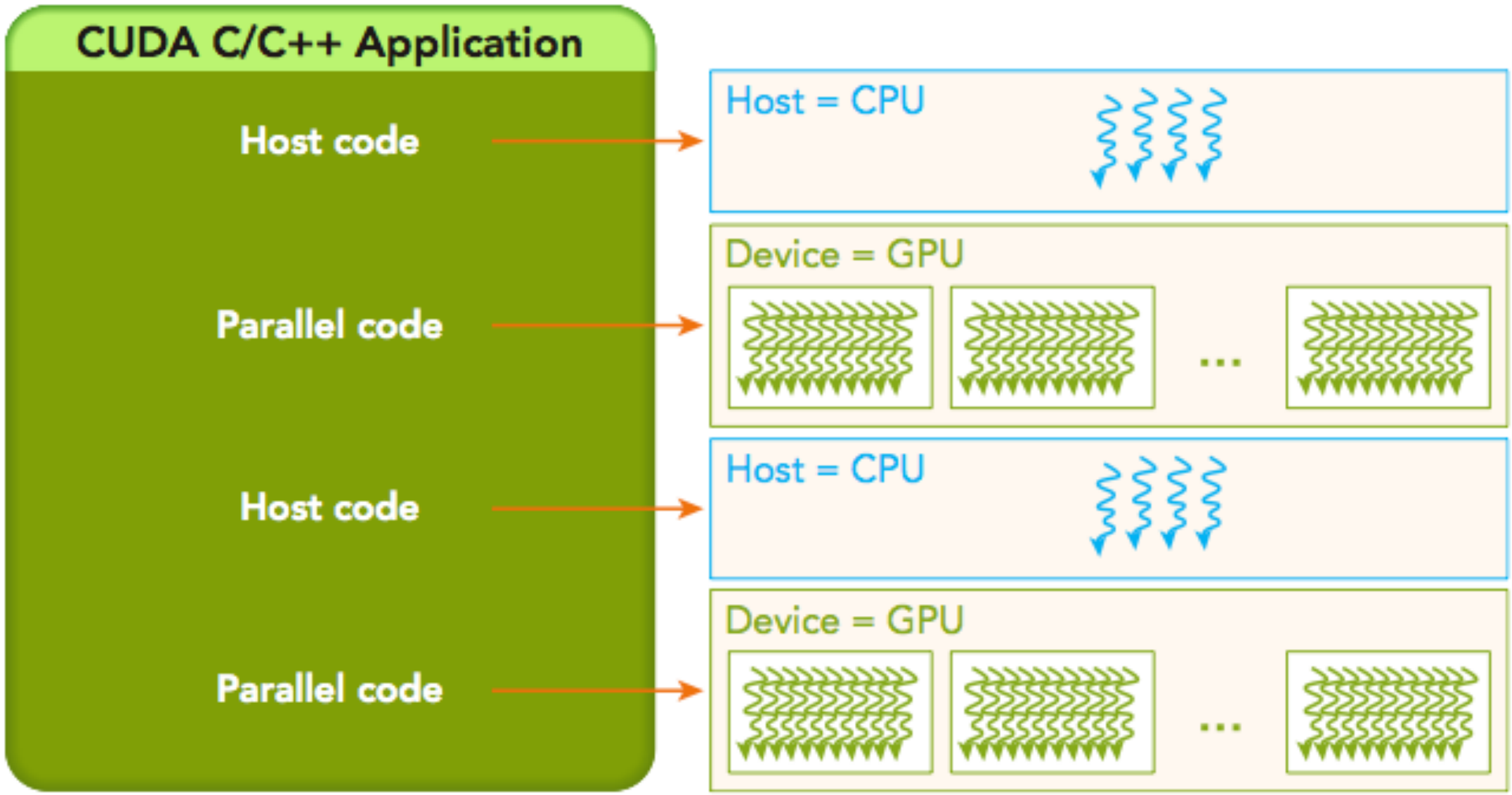
4. Copy data back from GPU memory to CPU memory.
5. Destroy GPU memories.

CUDA program structure

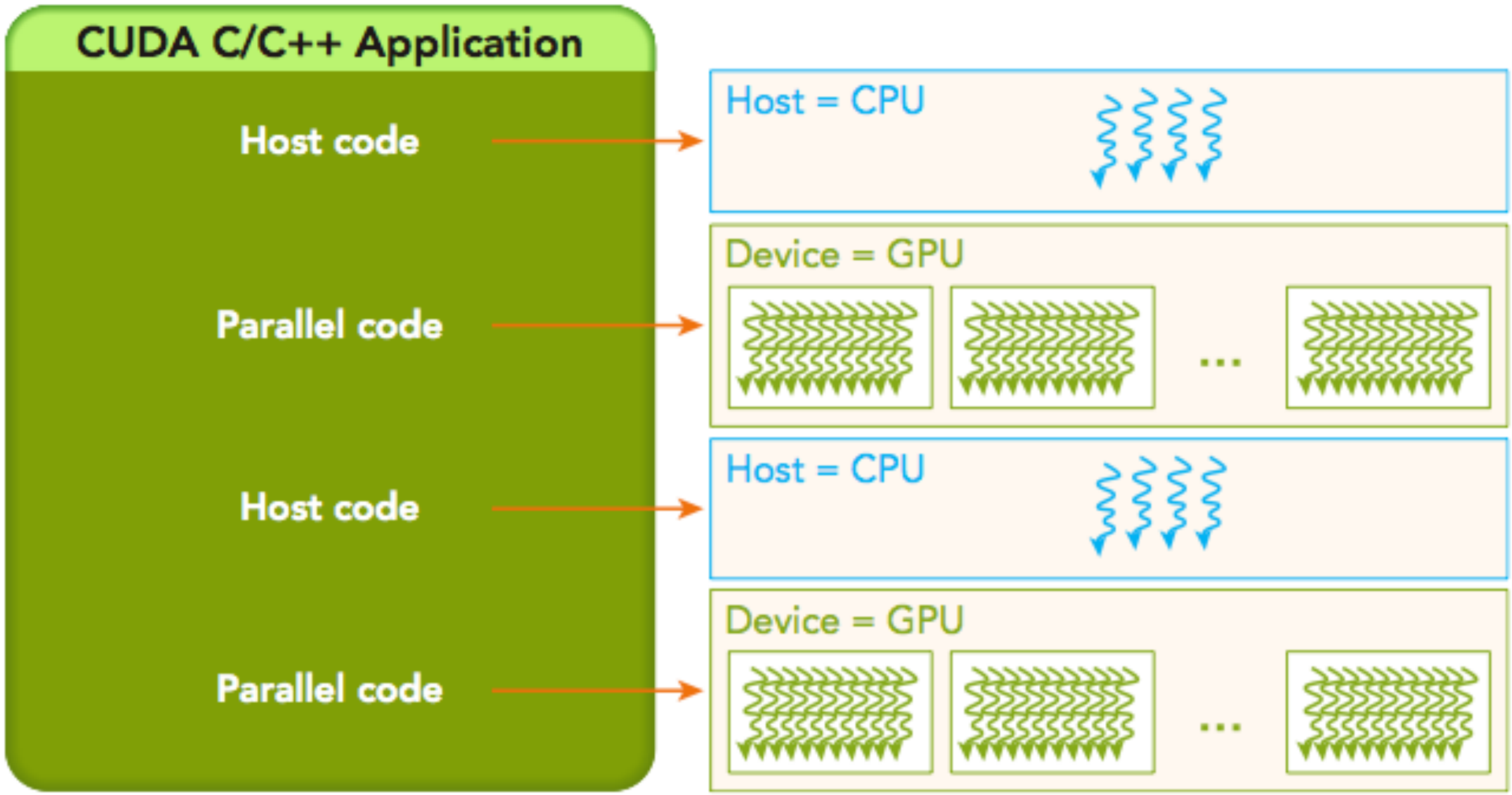
- The host can operate independently of the device for most operations. When a kernel has been launched, control is returned immediately to the host, freeing the CPU to perform additional tasks complemented by data parallel code running on the device.

- The CUDA programming model is primarily asynchronous so that GPU computation performed on the GPU can be overlapped with host-device communication.

CUDA program structure



CUDA program structure



All the threads that are generated by a kernel during an invocation are collectively called a **grid**.

Grid

- The programmer decides how to organize a **grid**, to improve parallelization

- When all threads of a kernel complete their execution, the corresponding grid terminates, and the execution continues on the host until another kernel is invoked.

- Grids are organized into **blocks**.

UNIVERSITÀ
DEGLI STUDI
FIRENZE

Hello world

CUDA Function Declarations

	Executed on the:	Only callable from the:
`__device__ float myDeviceFunc()`	device	device
`__global__ void myKernelFunc()`	device	host
`__host__ float myHostFunc()`	host	host

- `__global__` defines a kernel function, launched by host, executed on the device
  - Must return `void`
- By default, all functions in a CUDA program are `__host__` functions if they do not have any of the CUDA keywords in their declaration.

CUDA Function Declarations

	Executed on the:	Only callable from the:
`__device__ float myDeviceFunc()`	device	device
`__global__ void myKernelFunc()`	device	host
`__host__ float myHostFunc()`	host	host

One can use both `__host__` and `__device__` in a function declaration.

This combination triggers the compilation system to generate two versions of the same function.

One is executed on the host and can only be called from a host function.
The other is executed on the device and can only be called from a device or kernel function.

Hello world

```
#include "greeter.h"

#include <stdio.h>
#include <cuda_runtime_api.h>

__global__ void helloFromGPU() {
    printf("Hello World from GPU thread %d!\n", threadIdx);
}

int main(int argc, char **argv) {
    greet(std::string("Pinco"));

    helloFromGPU<<<1, 10>>>();

    // destroy and clean up all resources associated with current device
    //                                                + current process.
    cudaDeviceReset(); // CUDA functions are async...
                       // the program would terminate before CUDA kernel prints
                       // cudaDeviceSynchronize(); is an alternative, use it in
                       //   functions.
    return 0;
}
```

Hello world

Provides declaration of `greet(std::string name)`.
Definition is in `greeter.cpp`

```
#include "greeter.h"

#include <stdio.h>
#include <cuda_runtime_api.h>

__global__ void helloFromGPU() {
    printf("Hello World from GPU thread %d!\n", threadIdx);
}

int main(int argc, char **argv) {
    greet(std::string("Pinco"));

    helloFromGPU<<<1, 10>>>();

    // destroy and clean up all resources associated with current device
    //                                                + current process.
    cudaDeviceReset(); // CUDA functions are async...
                       // the program would terminate before CUDA kernel prints
                       // cudaDeviceSynchronize(); is an alternative, use it in
                       //   functions.
    return 0;
}
```

Hello world

```
#include "greeter.h"

#include <stdio.h>
#include <cuda_runtime_api.h>

__global__ void helloFromGPU() {
    printf("Hello World from GPU thread %d!\n", threadIdx);
}

int main(int argc, char **argv) {
    greet(std::string("Pinco"));

    helloFromGPU<<<1, 10>>>();

    // destroy and clean up all resources
    //                                    + current process.
    cudaDeviceReset(); // CUDA functions are async...
                       // the program would terminate before CUDA kernel prints
                       // cudaDeviceSynchronize(); is an alternative, use it in
                       //   functions.
    return 0;
}
```

Provides declaration of `greet(std::string name)`. Definition is in `greeter.cpp`

10 CUDA threads running on the GPU. This uses is 1 grid.

UNIVERSITÀ
DEGLI STUDI
FIRENZE

Managing memory

Memory model

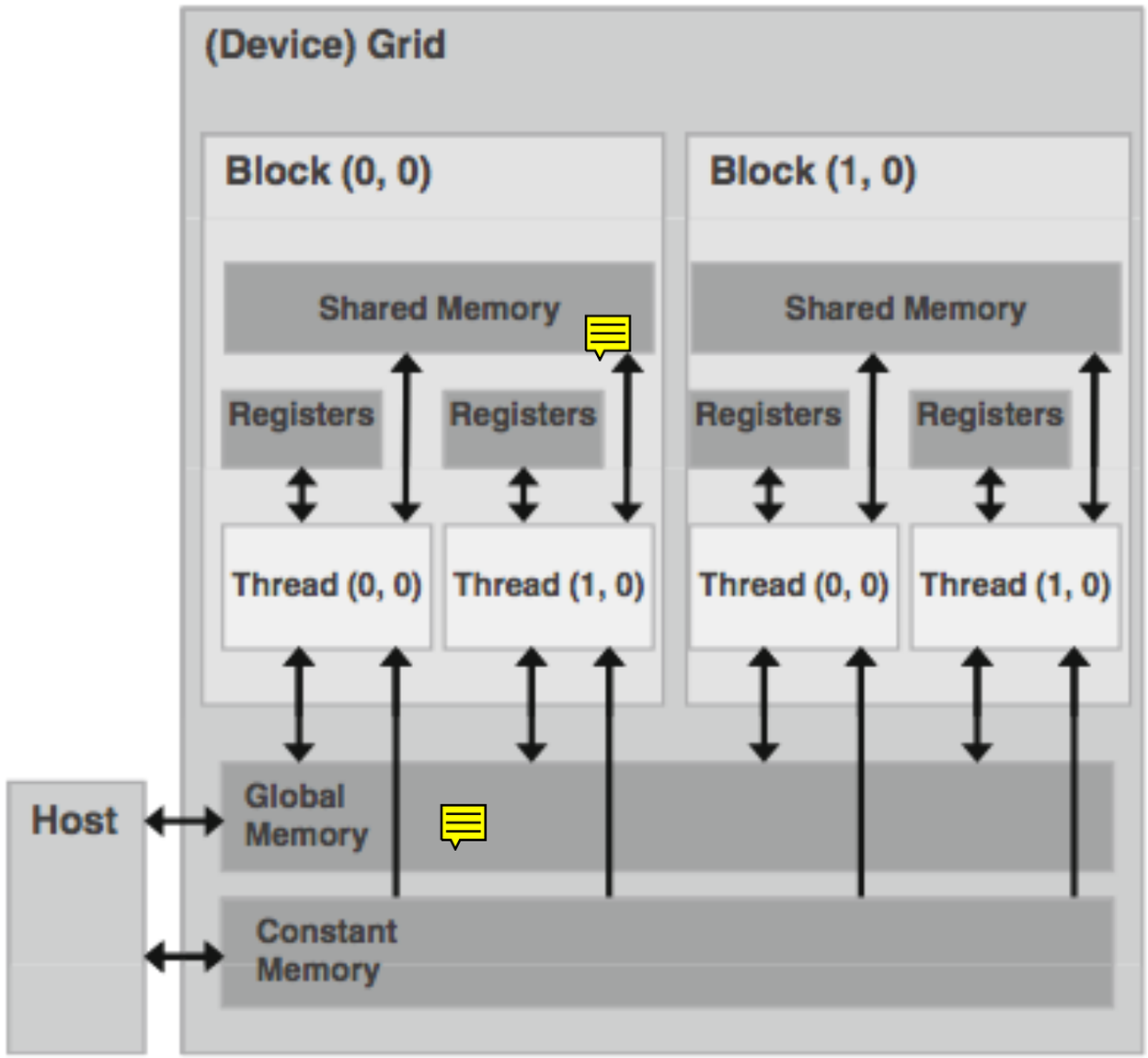
- The CUDA programming model assumes a system composed of a host and a device, each with its own separate memory.

- Kernels operate out of device memory. To allow you to have full control and achieve the best performance, the CUDA runtime provides functions to allocate device memory, release device memory, and transfer data between the host memory and device memory.

Standard C function	CUDA C function
malloc	cudaMalloc
memcpy	cudaMemcpy
memset	cudaMemset
free	cudaFree

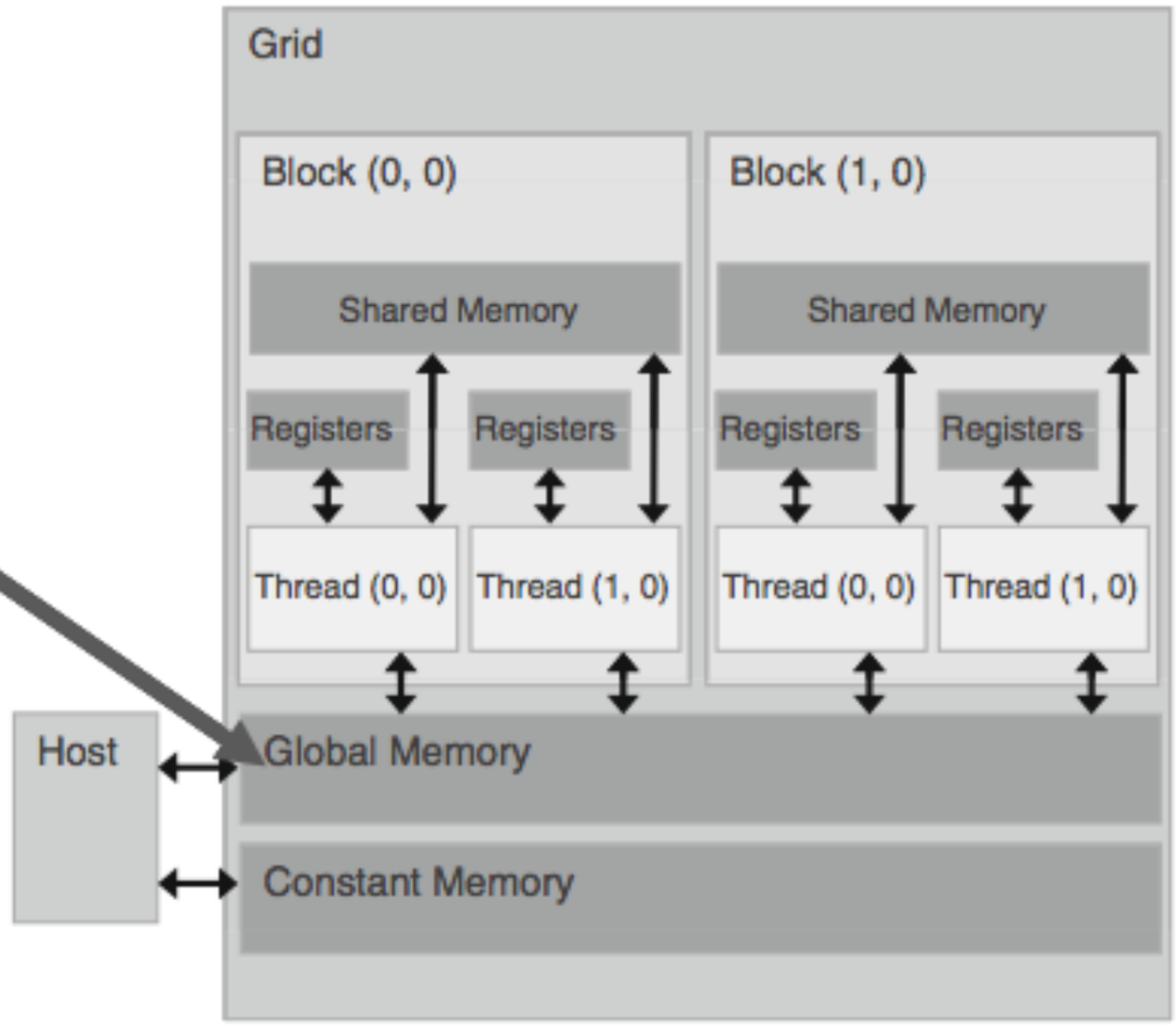
CUDA device memory model

- Device code can:
  - R/W per-thread registers
  - R/W per-thread local memory
  - R/W per-block shared memory
  - R/W per-grid global memory
  - Read only per-grid constant memory
- Host code can
  - Transfer data to/from per-grid global and constant memories



Example of CUDA API

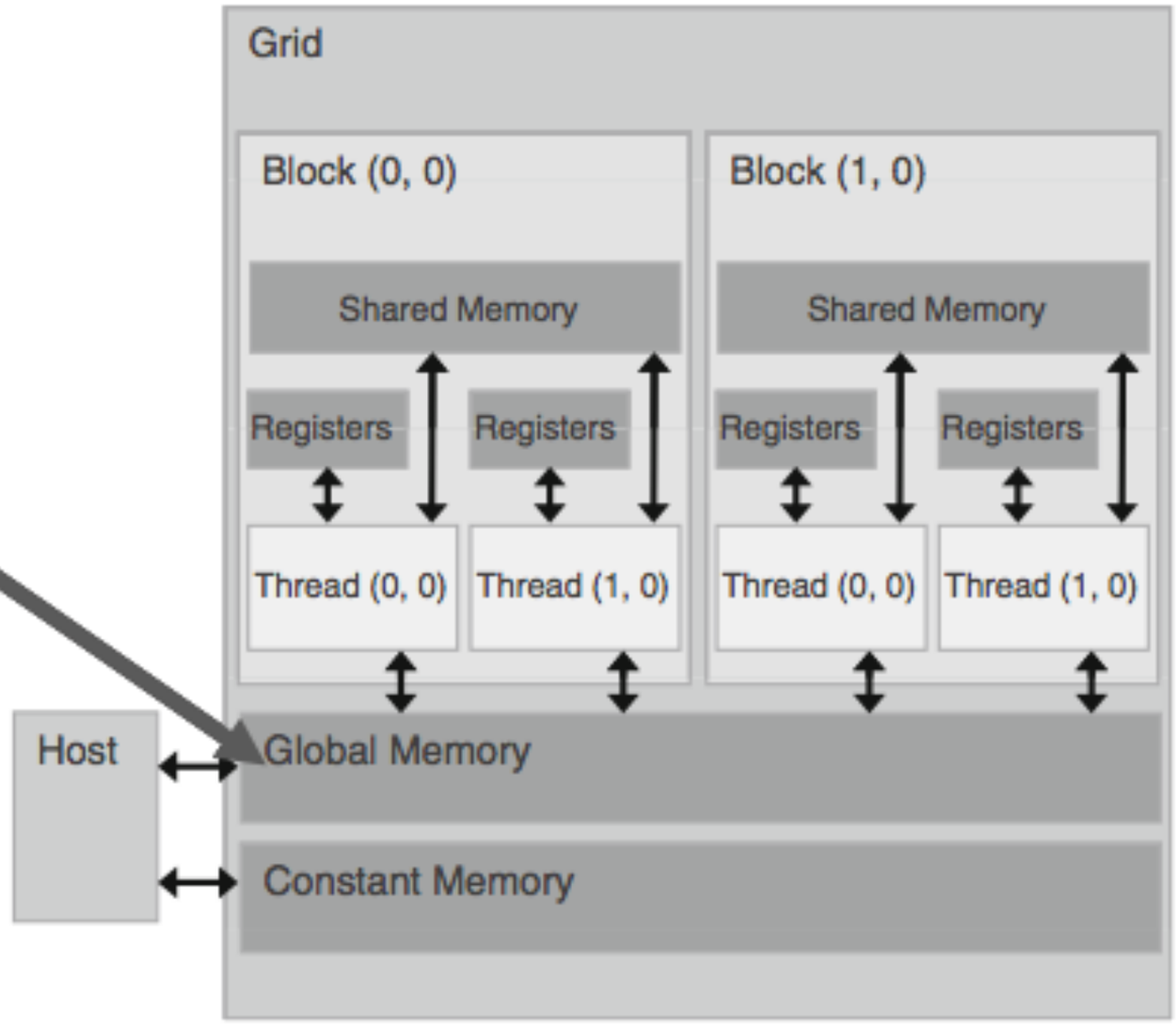
- cudaMalloc()
  - Allocates object in the device global memory
  - Two parameters
    - **Address of a pointer** to the allocated object
    - **Size of** of allocated object in terms of bytes
- cudaFree()
  - Frees object from device global memory
    - Pointer to freed object



Exam

```
float *Md;
int size = width * width * sizeof(float);
cudaMalloc((void**)&Md, size);
// ...
cudaFree(Md);
```

- cudaMalloc()
  - Allocates object in the device global memory
  - Two parameters
    - **Address of a pointer** to the allocated object
    - **Size of** of allocated object in terms of bytes
- cudaFree()
  - Frees object from device global memory
    - Pointer to freed object



cudaMalloc

- `cudaError_t cudaMalloc ( void** devPtr, size_t size )`

- This function allocates a linear range of device memory with the specified `size` in bytes. The allocated memory is returned through `devPtr`.

- if GPU memory is successfully allocated, it returns:
  - `cudaSuccess`

- Otherwise, it returns:
  - `cudaErrorMemoryAllocation`

CUDA errors

- You can convert an error code to a human-readable error message with the following CUDA run- time function:

- `char* cudaGetErrorString(cudaError_t error)`

- A common practice is to wrap CUDA calls in utility functions that manage the returned error

```
#define CUDA_CHECK_RETURN(value) CheckCudaErrorAux(__FILE__,__LINE__,
#value, value)

static void CheckCudaErrorAux (const char *file, unsigned line, const char
*statement, cudaError_t err) {
    if (err == cudaSuccess)
        return;
    std::cerr << statement<<" returned " << cudaGetErrorString(err) <<
                 "("<<err<< ") at "<<file<<":"<<line << std::endl;
    exit (1);
}
```

Use as: `CUDA_CHECK_RETURN(cudaFunction(parameters));`

- `char* cudaGetErrorString(cudaError_t error)`

- A common practice is to wrap CUDA calls in utility functions that manage the returned error

```
#define CUDA_CHECK_RETURN(value) CheckCudaErrorAux(__FILE__,__LINE__,
#value, value)

static void CheckCudaErrorAux (const char *file, unsigned line, const char
*statement, cudaError_t err) {
    if (err == cudaSuccess)
        return;
    std::cerr << statement<<" returned " << cudaGetErrorString(err) <<
                 "("<<err<< ") at "<<file<<":"<<line << std::endl;
    exit (1);
}
```

Use as: `CUDA_CHECK_RETURN(cudaFunction(parameters));`

```
#define CHECK(call)                                                    \
{                                                                      \
    const cudaError_t error = call;                                    \
    if (error != cudaSuccess)                                          \
    {                                                                  \
        fprintf(stderr, "Error: %s:%d, ", __FILE__, __LINE__);         \
        fprintf(stderr, "code: %d, reason: %s\n", error,               \
                cudaGetErrorString(error));                            \
        exit(1);                                                       \
    }                                                                  \
}
```

cudaMemcpy

- `cudaError_t cudaMemcpy ( void* dst, const void* src, size_t count, cudaMemcpyKind kind )`

- This function copies the specified bytes from the source memory area, pointed to by `src`, to the destination memory area, pointed to by `dst`, with the direction specified by kind, where kind takes one of the following types:
  - cudaMemcpyHostToHost
  - cudaMemcpyHostToDevice
  - cudaMemcpyDeviceToHost
  - cudaMemcpyDeviceToDevice
- This function exhibits **synchronous** behavior because the host application blocks until `cudaMemcpy` returns and the transfer is complete.

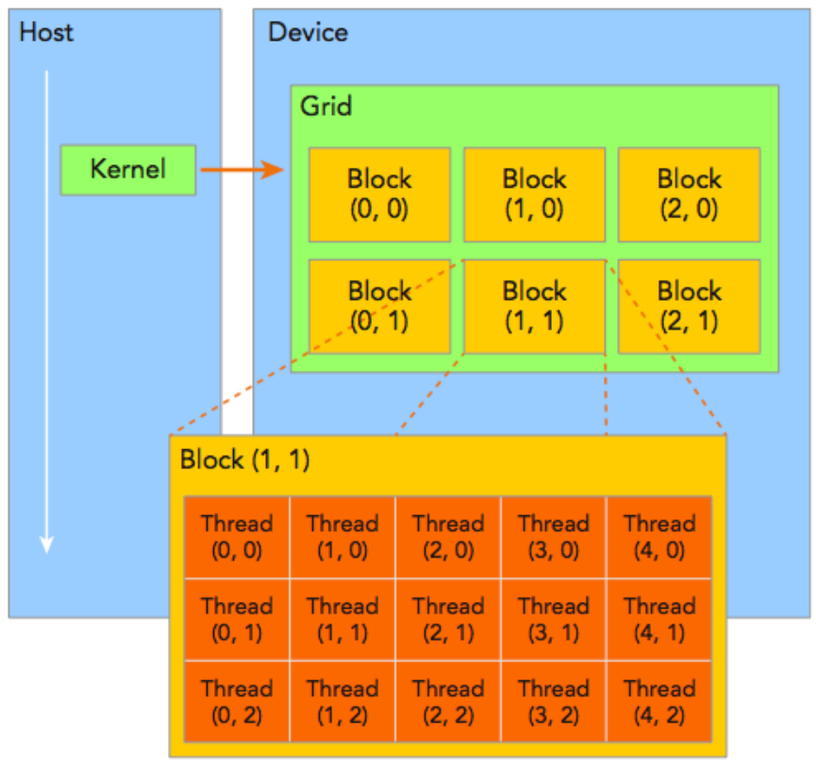
UNIVERSITÀ
DEGLI STUDI
FIRENZE

Organizing threads

Thread hierarchy

- When a kernel function is launched from the host side, execution is moved to a device where a large number of threads are generated and each thread executes the statements specified by the kernel function.

- CUDA exposes a thread hierarchy abstraction to enable you to organize your threads. This is a two-level thread hierarchy decomposed into blocks of threads and grids of blocks



Thread memory and coop

- All threads spawned by a single kernel launch are collectively called a **grid**.
  All threads in a grid share the same global memory space.
  A grid is made up of many thread **blocks**. A thread block is a group of threads that can cooperate with each other using:
    - Block-local synchronization
    - Block-local shared memory
- Threads from different blocks cannot cooperate.
- Threads rely on the following two unique coordinates to distinguish themselves from each other:
    - blockIdx (block index within a grid)
    - threadIdx (thread index within a block)

3D threadIdx

- `threadIdx` and `blockIdx` is a 3-component vector (`uint3`), so that threads can be identified using a one-dimensional, two-dimensional, or three-dimensional thread index, forming a one-dimensional, two-dimensional, or three-dimensional block of threads, called a thread block.

- This provides a natural way to invoke computation across the elements in a domain such as a vector, matrix, or volume.

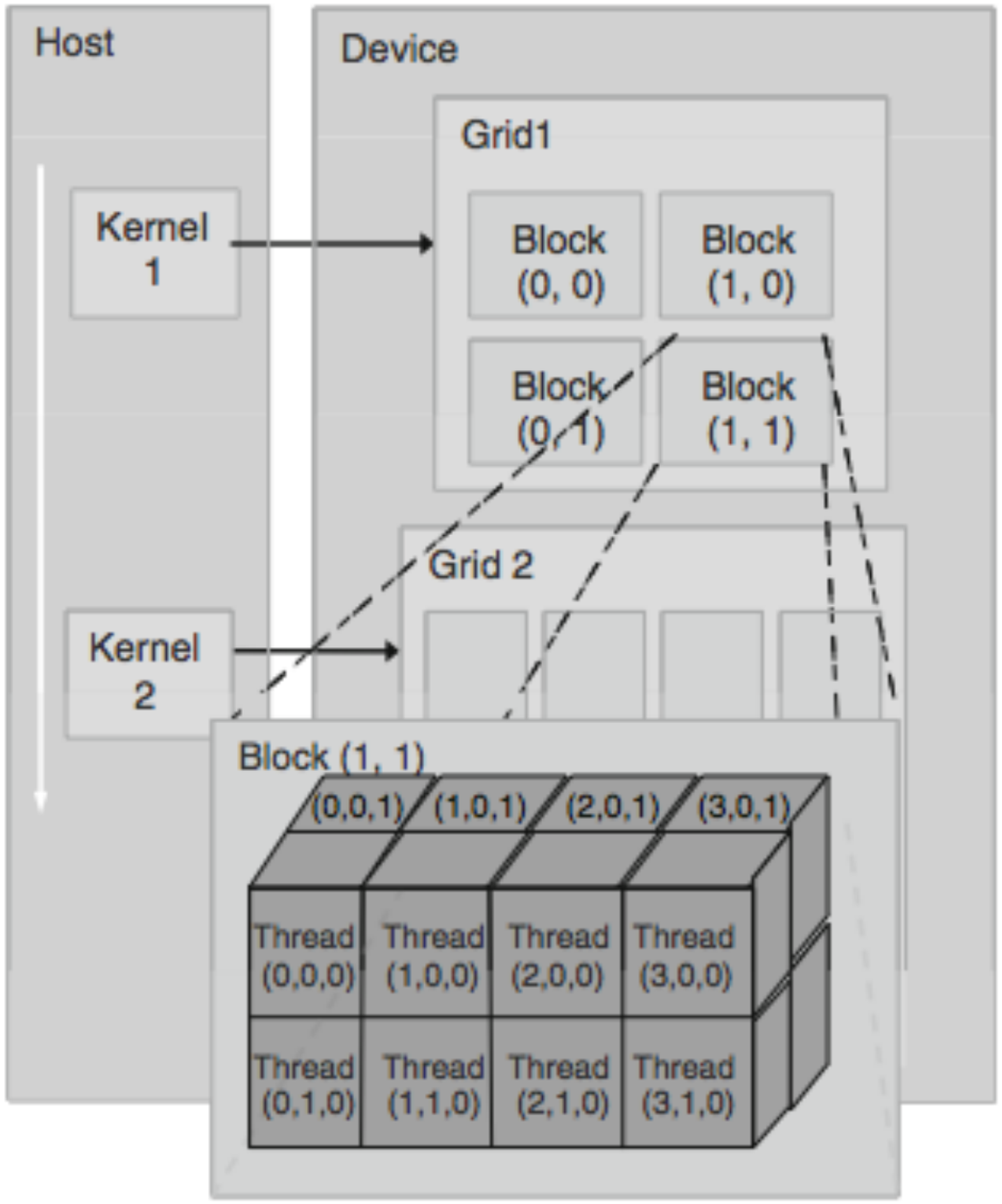
3D threadIdx

- `threadIdx` and `blockIdx` is a 3-component vector (`uint3`), so that threads can be identified using a one-dimensional, two-dimensional, or three-dimensional thread index, forming a one-dimensional, two-dimensional, or three-dimensional block of threads, called a thread block.

threadIdx and blockIdx are accessible through the fields x, y, and z respectively.

```
blockIdx.x
blockIdx.y
blockIdx.z

threadIdx.x
threadIdx.y
threadIdx.z
```

- A thread block is a batch of threads that can cooperate with each other by:
  - Synchronizing their execution
    - For hazard-free shared memory accesses
  - Efficiently sharing data through a low-latency shared memory
- Two threads from two different blocks cannot cooperate




```
threadIdx.x
threadIdx.y
threadIdx.z
```

Creating threads

- Each CUDA thread grid typically is comprised of thousands to millions of lightweight GPU threads per kernel invocation.

- Creating enough threads to fully utilize the hardware often requires a large amount of data parallelism; for example, each element of a large array might be computed in a separate thread.

- `kernel_name <<<grid, block>>>(argument list);`

- The first value in the execution configuration is `grid`, i.e. the **grid dimension**, the number of blocks to launch. The second value is `block`, i.e. the **block dimension**, the number of threads within each `block`. By specifying the grid and block dimensions, you configure:

  - The total number of threads for a kernel

  - The layout of the threads you want to employ for a kernel

Synchronizing threads

- A kernel call is asynchronous with respect to the host thread. After a kernel is invoked, control returns to the host side immediately.

- You can call the following function to force the host application to wait for all kernels to complete:

- `cudaError_t cudaDeviceSynchronize(void);`

UNIVERSITÀ
DEGLI STUDI
FIRENZE

Demo: organizing threads

Summing a vector

- CPU:

```
void sumArraysOnHost(float *A, float *B, float *C, int N) {
    for (int idx = 0; idx < N; idx++)
        C[idx] = A[idx] + B[idx];
}
```

- GPU

```
__global__ void sumArraysOnGPU(float *A, float *B, float *C,
int N) {
    int i = threadIdx.x;
    if (i < N)
      C[i] = A[i] + B[i];
}
```

Summing a vector

- CPU:

```
void sumArraysOnHost(float *A, float *B, float *C, int N) {
    for (int idx = 0; idx < N; idx++)
```

Supposing a vector with the length of 32 elements, you can invoke the kernel with 32 threads as follows:

-
```
sumArraysOnGPU<<<1, 32>>>(float *A, float *B, float *C, 32);
```

```
__global__ void sumArraysOnGPU(float *A, float *B, float *C,
int N) {
    int i = threadIdx.x;
    if (i < N)
      C[i] = A[i] + B[i];
}
```

Summing a vector

- CPU:

```
void sumArraysOnHost(float *A, float *B, float *C, int N) {
    for (int idx = 0; idx < N; idx++)
```

Supposing a vector with the length of 32 elements, you can invoke the kernel with 32 threads as follows:

-

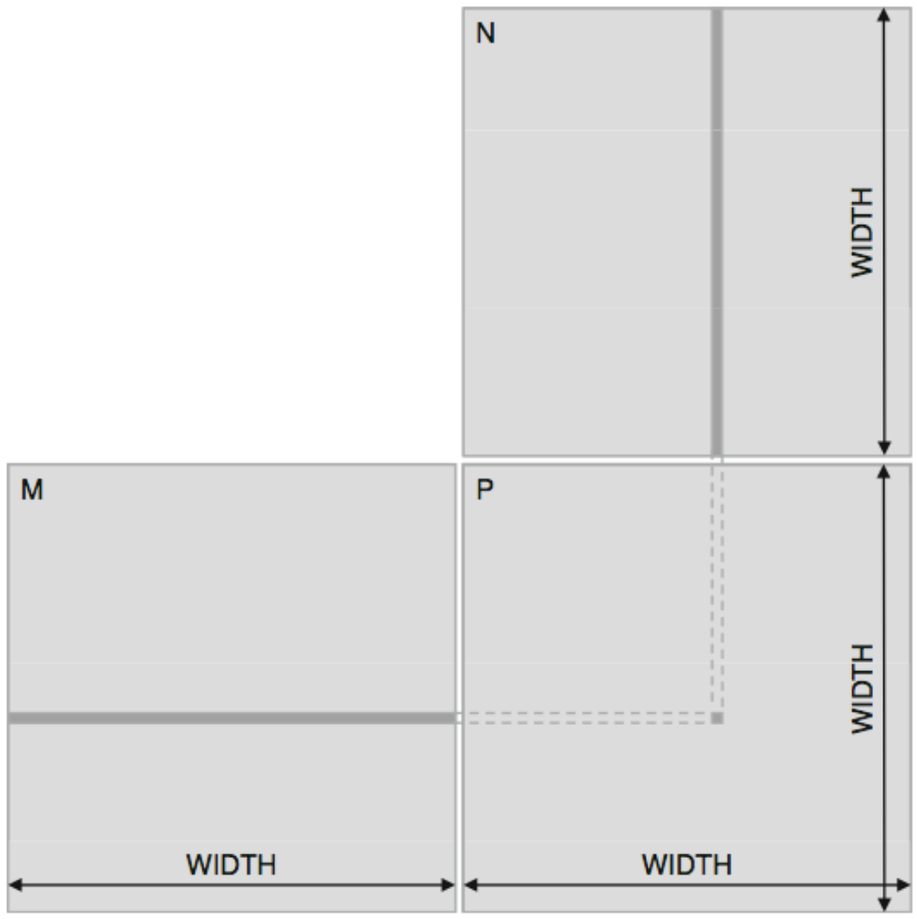
```
sumArraysOnGPU<<<1, 32>>>(float *A, float *B, float *C, 32);
```

Simplified version:

```
__global__ void sumArraysOnGPU(float *A, float *B, float *C) {
        int i = threadIdx.x;
        C[i] = A[i] + B[i];
}
```

Matrix multiplication: CPU

```
void MatrixMultiplication(float* M, float* N, float* P, int width) {

   for (int i = 0; i < width; ++i)

     for (int j = 0; j < width; ++j) {

        float sum = 0;

       for (int k  = 0; k < width; ++k) {

          float a = M[i * width + k];

          float b = N[k * width + j];

          sum += a * b;

       }

     P[i * width + j] = sum;

   }

}
```



Matrix multiplication: GPU

```
__global__ void MatrixMulKernel(float* Md, float* Nd,
float* Pd, int width) {

  int tx = threadIdx.x;
  int ty = threadIdx.y;

  float PValue = 0;

  for(int k=0; k<width; ++k) {
    float MdElem = Md[ty * width + k];
    float NdElem = Nd[k * width + tx];
    PValue += MdElem * NdElem;
  }

  Pd[ty * width + tx] = PValue;
}
```

Matrix multiplication: GPU

```
__global__ void MatrixMulKernel(float* Md, float* Nd,
float* Pd, int width) {

  int tx = threadIdx.x;
  int ty = threadIdx.y;

  float PValue = 0;

  for(int k=0; k<width; ++k) {
    float MdElem = Md[ty * width + k];
    float NdElem = Nd[k * width + tx];
    PValue += MdElem * NdElem;
  }

  Pd[ty * width + tx] = PValue;
}
```

Instead of two cycles on `i` and `j`, the CUDA threading hardware generates all of the `threadIdx.x` and `threadIdx.y` values for each thread.
Each thread uses its `threadIdx.x` and `threadIdx.y` to identify the row of `Md` and the column of `Nd` to perform the dot product operation.

Matrix multiplication: GPU

```
__global__ void MatrixMulKernel(float* Md, float* Nd,
float* Pd, int width) {

  int tx = threadIdx.x;
  int ty = threadIdx.y;

  float PValue = 0;

  for(int k=0; k<width; ++k) {
    float MdElem = Md[ty * width + k];
    float NdElem = Nd[k * width + tx];
    PValue += MdElem * NdElem;
  }

  Pd[ty * width + tx] = PValue;
```

Instead of two cycles on `i` and `j`, the CUDA threading hardware generates all of the `threadIdx.x` and `threadIdx.y` values for each thread.
Each thread uses its `threadIdx.x` and `threadIdx.y` to identify the row of `Md` and the column of `Nd` to perform the dot product operation.

$Thread_{2,3}$ will perform a dot product between column 2 of `Nd` and row 3 of `Md` and write the result into element (2,3) of `Pd`.
This way, the threads collectively generate all the elements of the `Pd` matrix.

Memory access

- M and N must be copied to the Md and Nd matrices allocated in the GPU

- Pd must be copied from te device back to the host

- Once all these operations are concluded it's possible to `cudaFree` Md, Nd and Pd

Limitations

- All these examples use only 1 block, but there's limit on the number of threads per block

- Indexing no longer as simple as using only `threadIdx.x` / `threadIdx.y`

  - One will have to account for the size of the block as well

UNIVERSITÀ
DEGLI STUDI
FIRENZE

CUDA occupancy

NVIDIA tool

- NVIDA provided an Excel sheet to compute the CUDA occupancy of a kernel
  - https://docs.nvidia.com/cuda/archive/12.1.0/cuda-occupancy-calculator/index.html
    - Now deprecated, use Nvidia Compute tool
- Select CUDA compute capabilities and a few parameters depending on the GPU card used (e.g. shred memory size, CUDA version, caching mode)
  - Some params depend on CUDA compute capability
- Set resource usage

UNIVERSITÀ DEGLI STUDI FIRENZE

CUDA occupancy example

CUDA Occupancy Calculator

Just follow steps 1, 2, and 3 below! (or click here for help)

1.) Select Compute Capability (click):	6,2	(Help)
1.b) Select Shared Memory Size Config (bytes)	65536	
1.d) Select Global Load Caching Mode	L1+L2 (ca)	
2.) Enter your resource usage:		
Threads Per Block	256	(Help)
Registers Per Thread	32	
User Shared Memory Per Block (bytes)	2048	

(Don't edit anything below this line)

3.) GPU Occupancy Data is displayed here and in the graphs:		
Active Threads per Multiprocessor	2048	(Help)
Active Warps per Multiprocessor	64	
Active Thread Blocks per Multiprocessor	8	
Occupancy of each Multiprocessor	100%	

Physical Limits for GPU Compute Capability:	6,2
Threads per Warp	32
Max Warps per Multiprocessor	64
Max Thread Blocks per Multiprocessor	32
Max Threads per Multiprocessor	2048
Maximum Thread Block Size	1024
Registers per Multiprocessor	65536
Max Registers per Thread Block	65536
Max Registers per Thread	255
Shared Memory per Multiprocessor (bytes)	65536
Max Shared Memory per Block	49152
Register allocation unit size	256
Register allocation granularity	warp
Shared Memory allocation unit size	256
Warp allocation granularity	4
Shared Memory Per Block (bytes) (CUDA runtime use)	0

Allocated Resources	Per Block	Limit Per SM	= Allocatable Blocks Per SM
Warps (Threads Per Block / Threads Per Warp)	8	64	8
Registers (Warp limit per SM due to per-warp reg count)	8	64	8
Shared Memory (Bytes)	2048	49152	32

Note: SM is an abbreviation for (Streaming) Multiprocessor

Maximum Thread Blocks Per Multiprocessor	Blocks/SM	* Warps/Block	= Warps/SM
Limited by Max Warps or Max Blocks per Multiprocessor	8	8	64
Limited by Registers per Multiprocessor	8	8	64
Limited by Shared Memory per Multiprocessor	32		

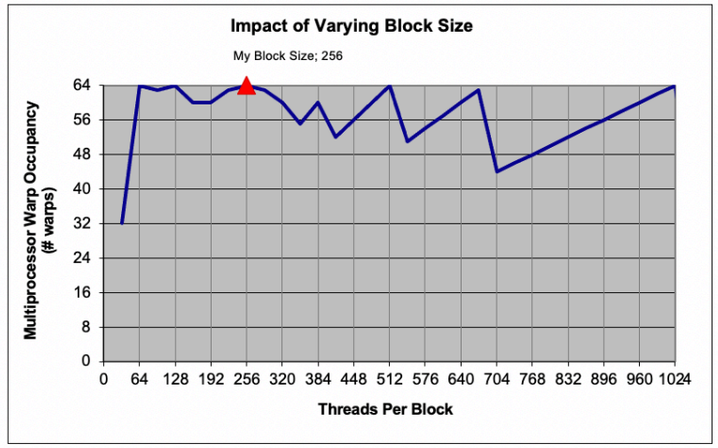
Note: Occupancy limiter is shown in orange

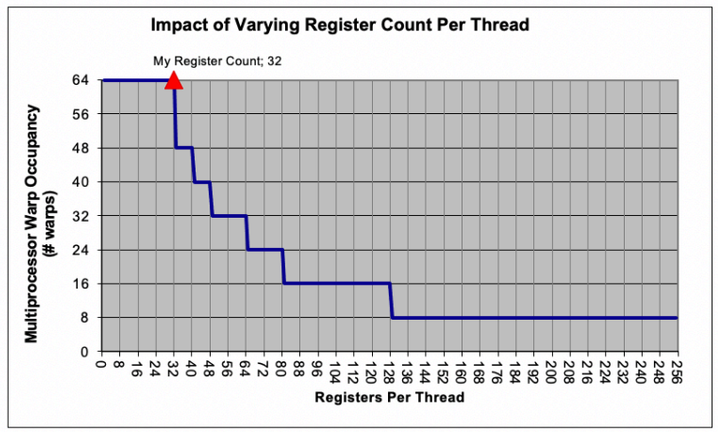
Physical Max Warps/SM = 64
Occupancy = 64 / 64 = 100%

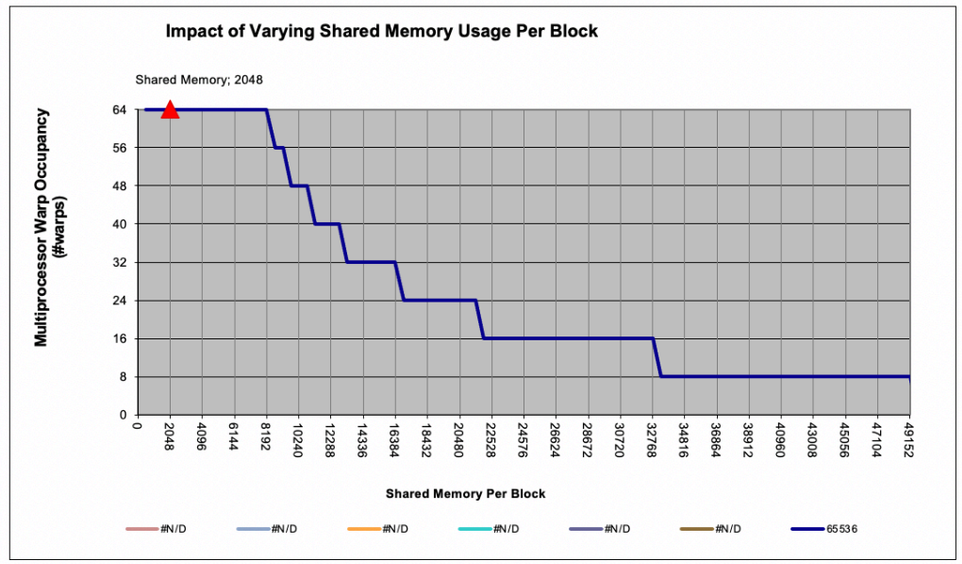
CUDA Occupancy Calculator	
Version:	11,1
Copyright and License	

Click Here for detailed instructions on how to use this occupancy calculator.
For more information on NVIDIA CUDA, visit http://developer.nvidia.com/cuda

Your chosen resource usage is indicated by the red triangle on the graphs. The other data points represent the range of possible block sizes, register counts, and shared memory allocation.







Calculator | Help | GPU Data | Copyright & License | +

Resources usage

- Two parameters in resource usage panel can be easily set up:
  - # threads per block
  - Shared memory per block (described in a future lecture)
- To get registers per block: add `-Xptxas="-v"` flag to `nvcc` compiler. You'll get an output similar to:

```
ptxas info: Used 14 registers, 4096 bytes smem, 380 bytes cmem[0]
ptxas info: Compiling entry function
'_Z12some_kernel_nameIfEvPT_PKS0_S3_' for 'sm_70'
ptxas info: Function properties for _Z12some_kernel_nameIfEvPT_PKS0_S3_
 0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
```

  - In this example pick 14 as the number of registers

CUDA Occupancy Calculator

Just follow steps 1, 2, and 3 below! (or click here for help)

1.) Select Compute Capability (click):	**6,2**	(Help)
1.b) Select Shared Memory Size Config (bytes)	**65536**	
1.d) Select Global Load Caching Mode	**L1+L2 (ca)**	
2.) Enter your resource usage:		
Threads Per Block	256	(Help)
Registers Per Thread	32	
User Shared Memory Per Block (bytes)	2048	

(Don't edit anything below this line)

3.) GPU Occupancy Data is displayed here and in the graphs:		
Active Threads per Multiprocessor	**2048**	(Help)
Active Warps per Multiprocessor	**64**	
Active Thread Blocks per Multiprocessor	**8**	
Occupancy of each Multiprocessor	**100%**	

Physical Limits for GPU Compute Capability:	**6,2**
Threads per Warp	32
Max Warps per Multiprocessor	64
Max Thread Blocks per Multiprocessor	32
Max Threads per Multiprocessor	2048
Maximum Thread Block Size	1024
Registers per Multiprocessor	65536
Max Registers per Thread Block	65536
Max Registers per Thread	255
Shared Memory per Multiprocessor (bytes)	65536
Max Shared Memory per Block	49152
Register allocation unit size	256
Register allocation granularity	warp
Shared Memory allocation unit size	256
Warp allocation granularity	4
Shared Memory Per Block (bytes) (CUDA runtime use)	0

Allocated Resources	Per Block	Limit Per SM	= Allocatable Blocks Per SM
Warps (Threads Per Block / Threads Per Warp)	8	64	8
Registers (Warp limit per SM due to per-warp reg count)	8	64	8
Shared Memory (Bytes)	2048	49152	32

Note: SM is an abbreviation for (Streaming) Multiprocessor

Maximum Thread Blocks Per Multiprocessor	Blocks/SM	* Warps/Block	= Warps/SM
Limited by Max Warps or Max Blocks per Multiprocessor	**8**	**8**	**64**
Limited by Registers per Multiprocessor	**8**	**8**	**64**
Limited by Shared Memory per Multiprocessor	32		

Note: Occupancy limiter is shown in orange

Physical Max Warps/SM = 64

- The blue table shows computed occupancy
- The maximum threads block per SMM shows what are the limitations (highlighted in orange)
- e.g. reducing registers # we don’t have anymore a limitation based on them.

CUDA Occupancy Calculator

Just follow steps 1, 2, and 3 below! (or click here for help)

1.) Select Compute Capability (click):	**6,2**	(Help)
1.b) Select Shared Memory Size Config (bytes)	**65536**	
1.d) Select Global Load Caching Mode	**L1+L2 (ca)**	
2.) Enter your resource usage:		
Threads Per Block	256	(Help)
Registers Per Thread	14	
User Shared Memory Per Block (bytes)	2048	

(Don't edit anything below this line)

3.) GPU Occupancy Data is displayed here and in the graphs:		
Active Threads per Multiprocessor	**2048**	(Help)
Active Warps per Multiprocessor	**64**	
Active Thread Blocks per Multiprocessor	**8**	
Occupancy of each Multiprocessor	**100%**	

Physical Limits for GPU Compute Capability:	**6,2**
Threads per Warp	32
Max Warps per Multiprocessor	64
Max Thread Blocks per Multiprocessor	32
Max Threads per Multiprocessor	2048
Maximum Thread Block Size	1024
Registers per Multiprocessor	65536
Max Registers per Thread Block	65536
Max Registers per Thread	255
Shared Memory per Multiprocessor (bytes)	65536
Max Shared Memory per Block	49152
Register allocation unit size	256
Register allocation granularity	warp
Shared Memory allocation unit size	256
Warp allocation granularity	4
Shared Memory Per Block (bytes) (CUDA runtime use)	0

Allocated Resources	Per Block	Limit Per SM	= Allocatable Blocks Per SM
Warps (Threads Per Block / Threads Per Warp)	8	64	8
Registers (Warp limit per SM due to per-warp reg count)	8	128	16
Shared Memory (Bytes)	2048	49152	32

Note: SM is an abbreviation for (Streaming) Multiprocessor

Maximum Thread Blocks Per Multiprocessor	Blocks/SM	* Warps/Block	= Warps/SM
Limited by Max Warps or Max Blocks per Multiprocessor	**8**	**8**	**64**
Limited by Registers per Multiprocessor	16		
Limited by Shared Memory per Multiprocessor	32		

Note: Occupancy limiter is shown in orange

Physical Max Warps/SM = 64
Occupancy = 64 / 64 = 100%

- The blue table shows computed occupancy
- The maximum threads block per SMM shows what are the limitations (highlighted in orange)
- e.g. reducing registers # we don’t have anymore a limitation based on them.

CUDA occupancy

	Result	Size	Time	Cycles	GPU	SM Frequency	Process	Attributes
Current	594 - mergeRanksAndIndicesKernel	(64, 1, 1)x(256, 1, 1)	2.75 us	5,947	0 - NVIDIA GeForce RTX 4080	2.16 Ghz	[12660] CuMergeSort.exe	

Summary | Details | Source | Context | Comments | Raw | Session

Compare | Tools | View | Export

Occupancy Calculator
Metric Details Window
Launch Details Window

GPU Speed Of Light Throughput

Compute (SM) Throughput [%]	5.73	Duration [us]	2.75
Memory Throughput [%]	7.26	Elapsed Cycles [cycle]	5,947

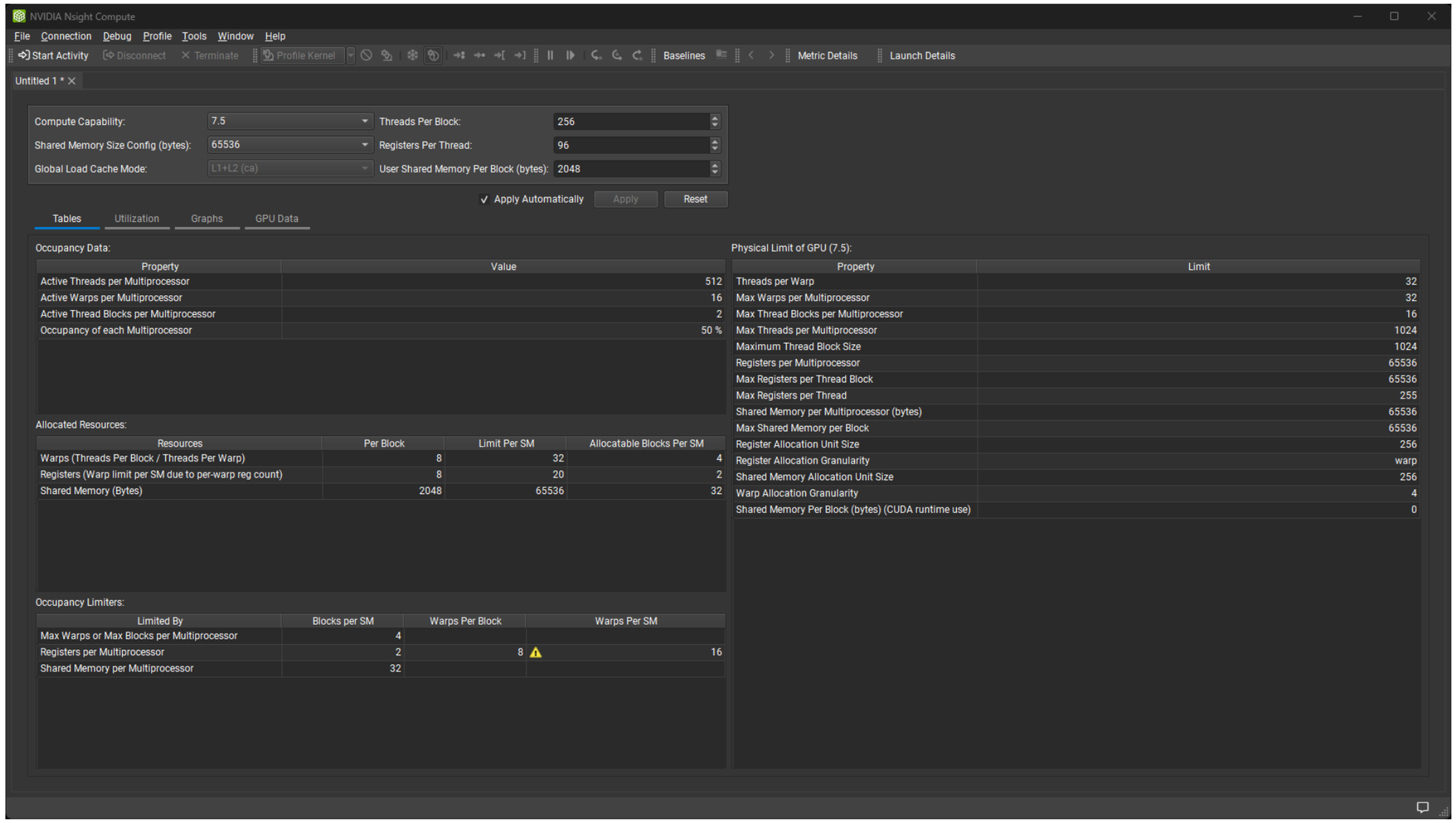
Occupancy

Occupancy is the ratio of the number of active warps per multiprocessor to the maximum number of possible active warps. Another way to view occupancy is the percentage of the hardware's ability to process warps that is actively in use. Higher occupancy does not always result in higher performance, however, low occupancy always reduces the ability to hide latencies, resulting in overall performance degradation. Large discrepancies between the theoretical and the achieved occupancy during execution typically indicates highly imbalanced workloads.

Theoretical Occupancy [%]	50	Block Limit Registers [block]	84
Theoretical Active Warps per SM [warp]	32	Block Limit Shared Mem [block]	32
Achieved Occupancy [%]	1.56	Block Limit Warps [block]	64
Achieved Active Warps Per SM [warp]	1.00	Block Limit SM [block]	32

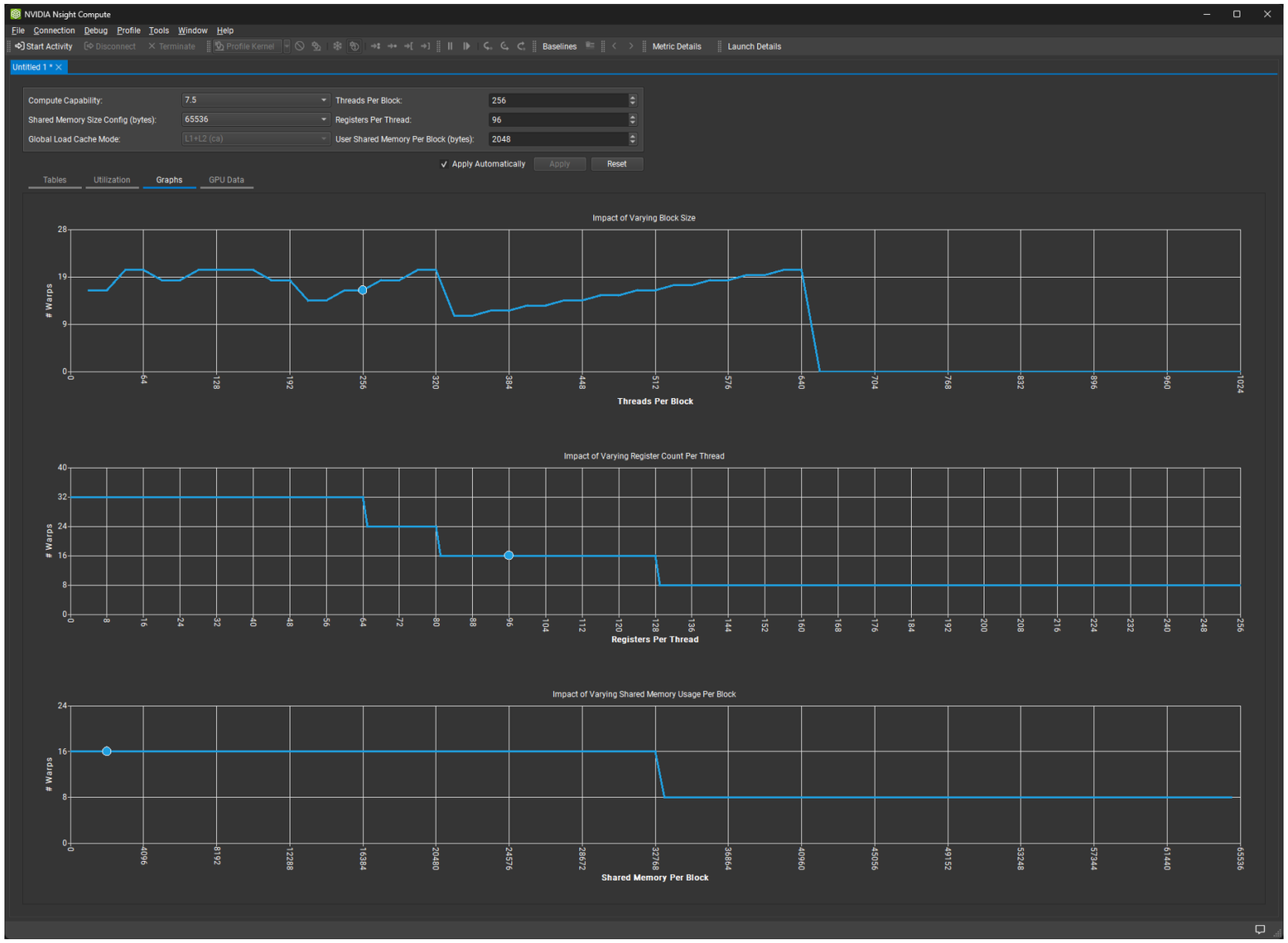
- The Occupancy Calculator can be opened from the Profiler Report in NVIDIA NSight Compute

CUDA occupancy



- The tables show the occupancy, as well as the number of active threads, warps, and thread blocks per multiprocessor, and the maximum number of active blocks on the GPU.

CUDA occupancy



- The graphs show the occupancy for your chosen block size as a blue circle, and for all other possible block sizes as a line graph.

Credits

- These slides report material from:
  - Prof. Jan Lemeire (Vrjie Universiteit Brussel)
  - Prof. Dan Negrut (Univ. Wisconsin - Madison)
  - NVIDIA GPU Teaching Kit

Books

- Parallel and High Performance Computing, R. Robey, Y. Zamora, Manning - Chapt. 9 and 10

- Programming Massively Parallel Processors: A Hands-on Approach, D. B. Kirk and W-M. W. Hwu, Morgan Kaufman - 2nd edition - Chapt. 1 and 3

  or

  Programming Massively Parallel Processors: A Hands-on Approach, D. B. Kirk and W-M. W. Hwu, Morgan Kaufman - 2nd edition - Chapt. 1-2

- Professional CUDA C Programming, J. Cheng, M. Grossman and T. McKercher, Wrox - Chapt. 1-2