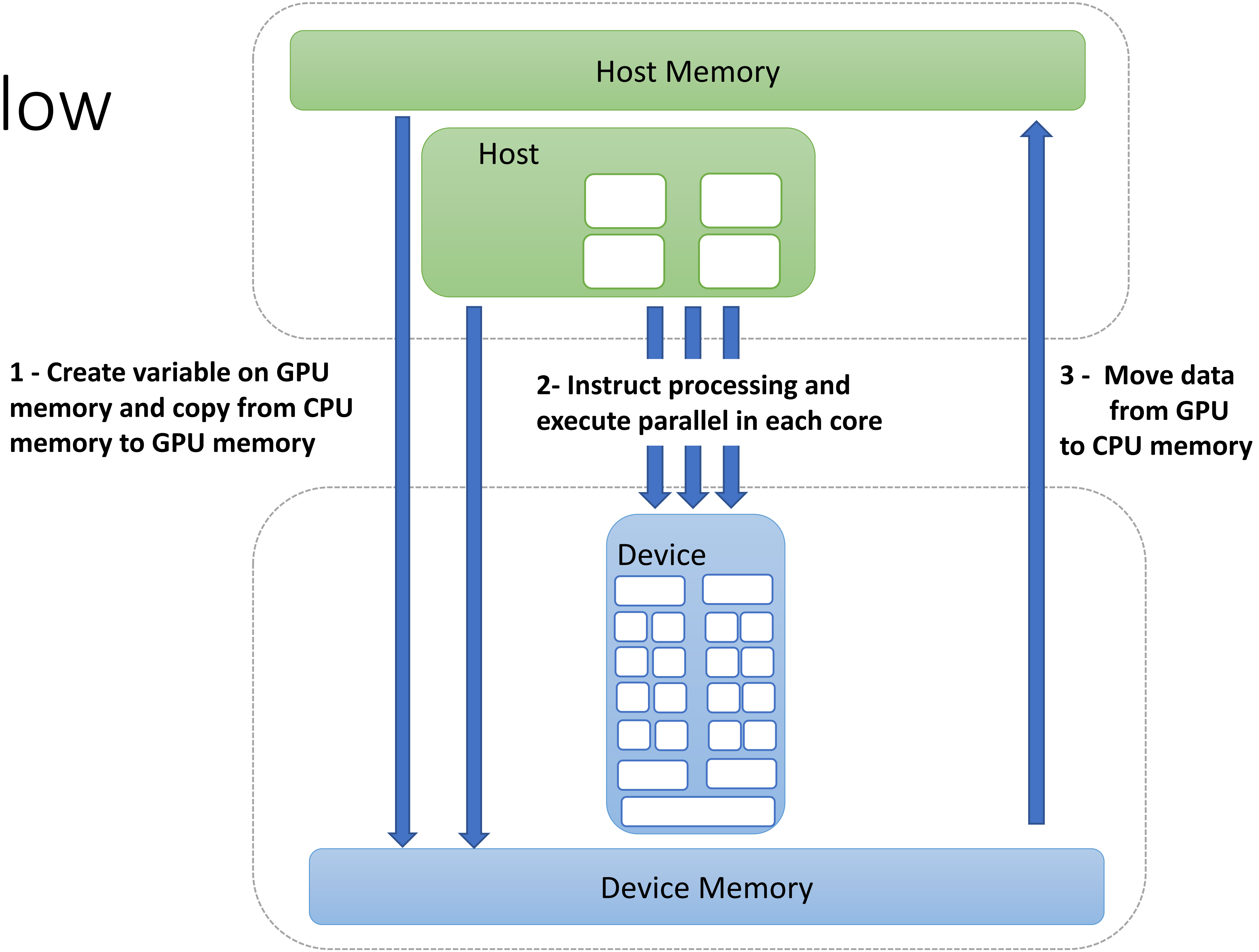
CUDA Essentials
Kernel Launching

Stefano Markidis and Sergio Rivas-Gomez

Four Key-Points

- Kernel launching is the invocation of a function to be run on the GPUs
- Second, CUDA follows the SIMT model in which each thread on the GPU executes the kernel function
- In CUDA, threads are organized in thread blocks so you need to decide the number of thread blocks and the number of threads per block (= execution configuration).
- When you define a kernel in your code, you need to the `__global__` or `__device__` qualifiers prepend before the function name when you launch the kernel from the CPU and GPU respectively.

CUDA Workflow

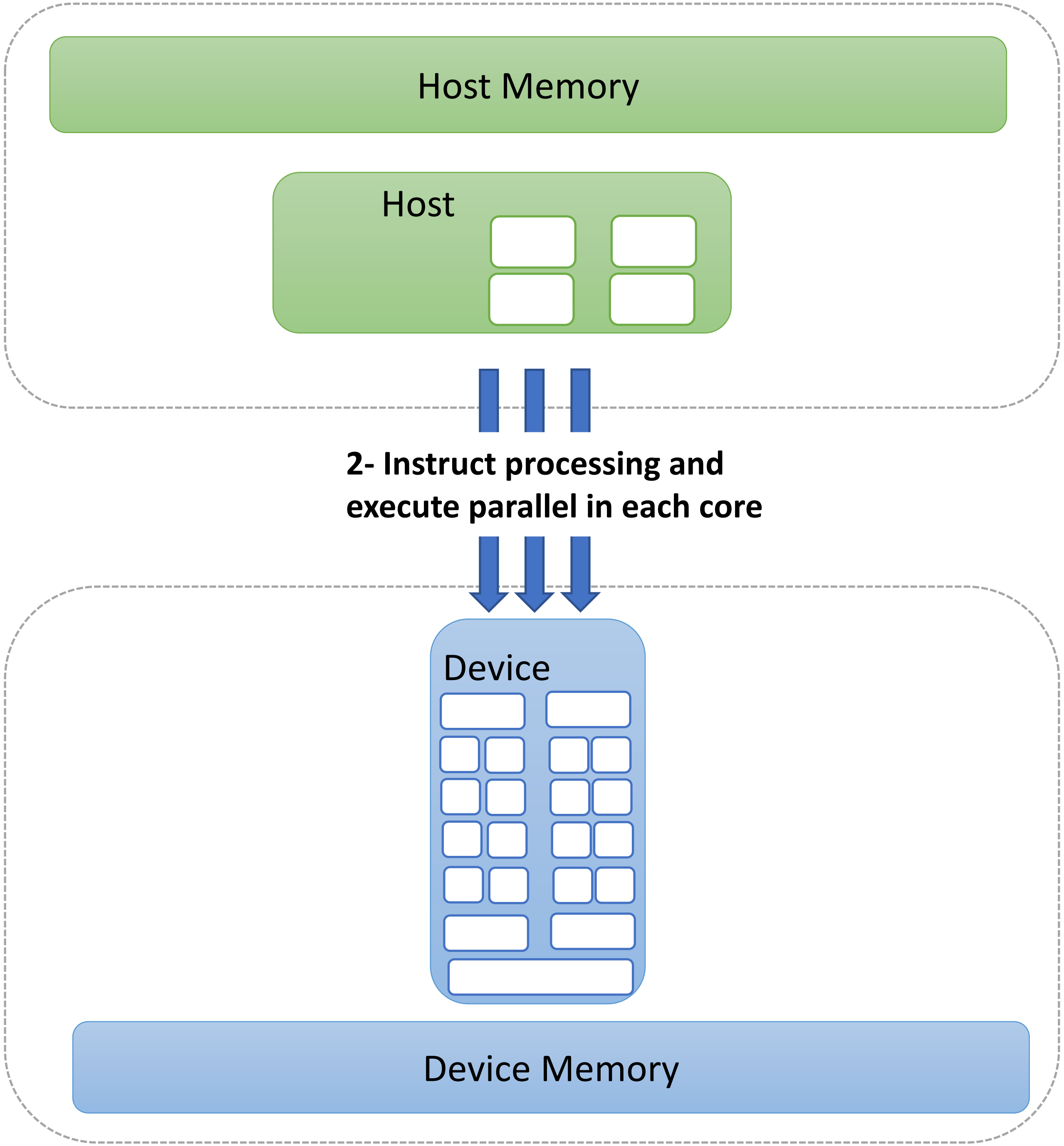


CUDA Code Execution on GPU

CUDA follows the SIMT parallelism model where each CUDA thread executes the same function (kernel).

To run a kernel we need:

1. Call the function to be executed by each thread on the GPU (so called **Kernel Launch**) providing the number of threads and their grouping
2. Define the function to be executed on the GPU



Kernel Launching

To call a function, called `my_kernel`, and run it on the GPU threads:

```
my_kernel <<<Dg, Db>>>(arg1, arg2, …)
```

When you launch a kernel you need to provide two arguments `Dg` and `Db` in the triple angle brackets to basically define how many threads you want to use on the GPU:

`Dg` = number of thread blocks

`Db` = number of threads per block

The total number of threads is `Dg*Db`.

`<<<Dg, Db>>>` is called **Execution Configuration**.

Thread Organization in CUDA

- Threads on a GPU are grouped into **thread blocks**:
  - The number of threads per block has maximum value of 1024
  - You can specify number of threads in 1, 2 or 3 dimensions
- Thread blocks are organized in **grids** that can be 1D (`x`), 2D (`x` and `y`) or 3D (`x`, `y` and `z`)
  - No maximum number of blocks per grid direction.
  - In the assignment and project, we will only use 1D grids: we need to specify only how many blocks in the `x` direction.
  - 2D grids are convenient for processing images
  - 3D grids are convenient for processing stacks of images

1D Grid with 7 Thread Blocks

256 T	**256 T**	**256 T**	**256 T**	**256 T**	**256 T**	**256 T**

Dg = 7
Db = 256
Tot. No. Threads = ?

Thread Organization in CUDA II

- 1D: `Dg` and `Db` can be simply two scalar values
  - **Example:** `Dg = 7; Db = 256; my_kernel <<<Dg, Db>>>(arg1, arg2, …);`
  - **Equivalent:** `my_kernel <<< dim3(Dg,1,1), dim3(Db,1,1) >>>(arg1, arg2, …;)`
- 2D: `Dg` and `Db` specifies number of blocks and number of threads per block in the `x` and `y` directions:
  - **Example:** `my_kernel <<< dim3(Dgx,Dgy), dim3(Dbx,Dby) >>>(arg1, arg2, …);`
  - **Equivalent:** `my_kernel <<< dim3(Dgx,Dgy,1), dim3(Dbx,Dby,1) >>>(arg1, arg2, …);`
- 3D: Execution configuration specifies number of blocks and number of threads per block in the `x`, `y` and `z` directions:
  - **Example:** `my_kernel <<< dim3(Dgx,Dgy,Dgz), dim3(Dbx,Dby,Dbz) >>>(arg1, arg2, …);`

Work ≠ Resources

- Thread blocks and grids are abstractions to express the application work. When you launch a kernel, you specify the **work** you want to do.
- The GPU then uses its resources (SMs, SPs, ..) to perform the work. The more resources a GPU has, the more work it can do in parallel.
- Basic strategy in CUDA is to **oversubscribe** to hide latency

- No. of Block ≠ No. of SMs
- No. of Threads per Block ≠ No. of SPs per SM

How do you Define Kernel Function?

CUDA makes distinction between function depending who is calling and where they should run. By prepending one of the following function type qualifiers:

- `__global__` is the **qualifier for kernels** (which can be called by the host and executed on device)
- `__device__` functions are called from **the device and execute on the device** (a function that is called from a kernel needs the `__device__` qualifier)
- `__host__` functions called from the host and executed on the host (default qualifier, often omitted)

Question: which qualifier do you have before the function you call from the CPU and you want to run on GPU:

- `__global__`
- `__host__`
- `__device__`

?

Kernel Limitations

- Kernels execute on the GPU and do not, **in general**, have access to data stored on the host side.
- Kernels **cannot return a value**, so the return type is always `void`, and kernel declarations starts as

```
__global__ void my_kernel(…)
```

  - How do I get the results from my kernel ?

To Summarize

- Kernel launching is the invocation of a function to be run on the GPUs
- Each thread on the GPU executes the kernel function.
- In CUDA, threads are organized in thread blocks so to set-up the number of threads you need to decide the number of thread blocks and the number of threads per block (execution configuration).
- When you define a kernel in your code, you need to prepend before the function name the `__global__` or `__device__` qualifiers when you launch the kernel from the CPU and GPU respectively.

How do you Choose the Execution Configuration?

- The problem dimensionality determines how many dimensions you want to use:
  - 1D array like an array of particle structures maps to 1D grids
  - Image Processing problem maps to 2D grids
- The input size determines the total number of threads you want to use so each thread process one input element
- The number of threads per block is typically fixed to a multiple of 32
- **Example:** array a has `len = 512` elements you want to 512 threads so each thread processes one `a` element. We fixed `TPB = 256`

Execution configuration will be `<<< len/TPB, TPB >>>`