CUDA Essentials

Kernel Execution, Indexing and Vector Types

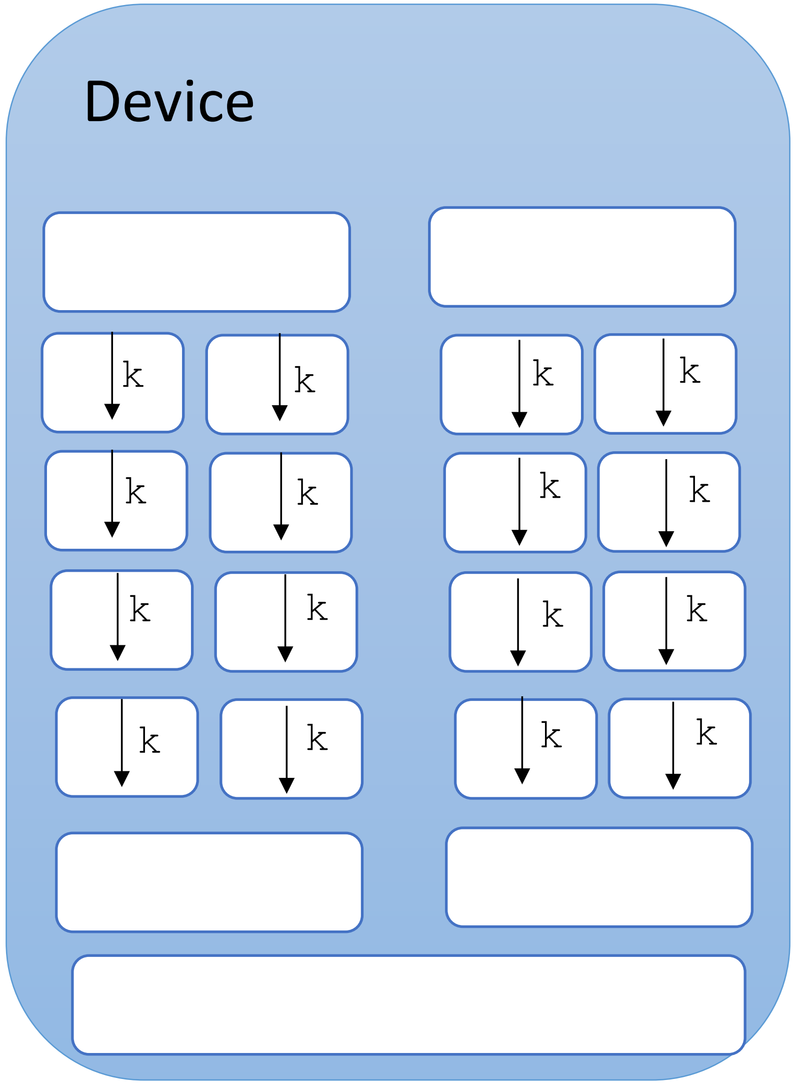
Stefano Markidis and Sergio Rivas-Gomez

Three Key-Points

- In a kernel, to determine which part of the array work on or different execution path we will need to obtain the thread ID. For that, we need to obtain the ID of the block, the ID of the thread in the block and do a simple math that is called thread indexing.
- Kernel execution, except for CUDA data movement functions, are asynchronous so we might use synchronization to ensure correct execution.
- CUDA provides vector types, similar to C structures, with 2, 3 and 4 members, to improve memory bandwidth

Kernel Execution

- SIMT execution: each thread executes the same kernel `k`.
- How do I know which thread I am, i.e. what is the my thread ID? The problem of finding the thread ID is called indexing.



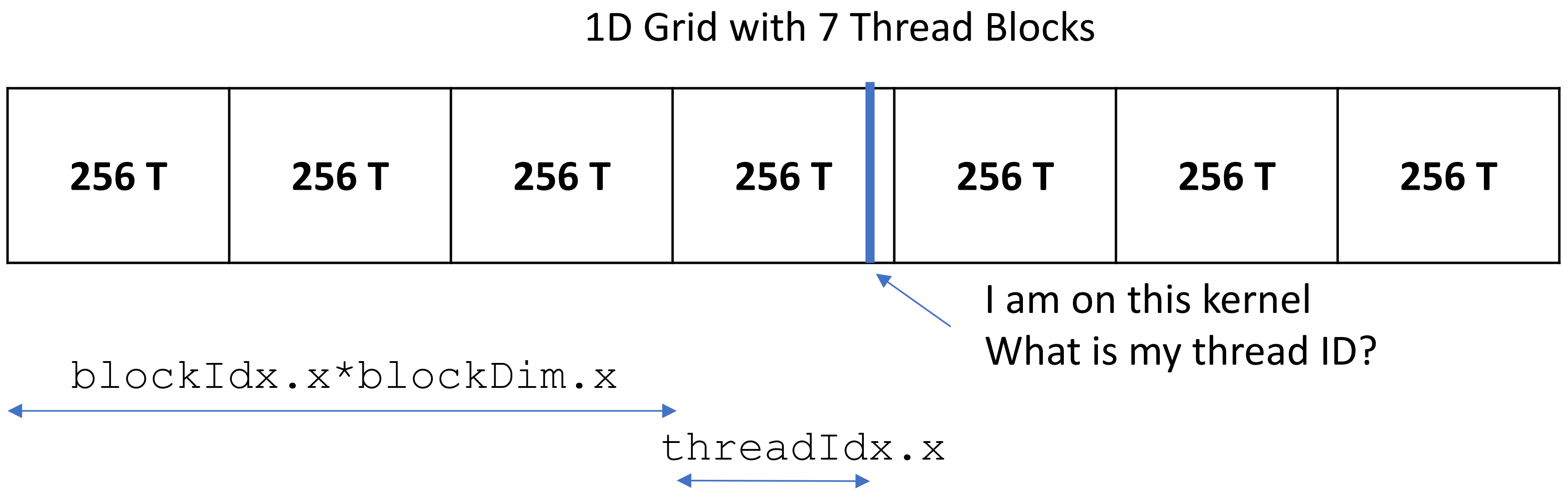
CUDA Built-in Variables: Dimension and Index

Kernel provides 4 built variables that can be accessed in during the kernel execution to determine your ID:

- **Index variables**
  - `blockIdx` = index of the block in the grid
    - `blockIdx.x` is index of the block in the `x` direction, `blockIdx.y` in the `y` direction and `blockIdx.z` in the `z` direction.
  - `threadIdx` = index of the thread within the block, `threadIDX.x` is the thread index in the x direction within the block …
- **Dimension variables**
  - `blockDim` = number of threads in each block: `blockDim.x`, …
  - `gridDim` = number of blocks in the grid, `gridDim.x`, …

What is my thread ID ?

Launched this kernel on `<<<7,256>>>`



```
__global__ k(…){
        …
        int myID = blockIdx.x*blockDim.x + threadIdx.x;
        …
```

Asynchronous Execution

A kernel launch is asynchronous = it returns control to the CPU immediately after starting up the GPU process, before the kernel has finished executing.

To enforce synchronization, CUDA provides functions to synchronize :

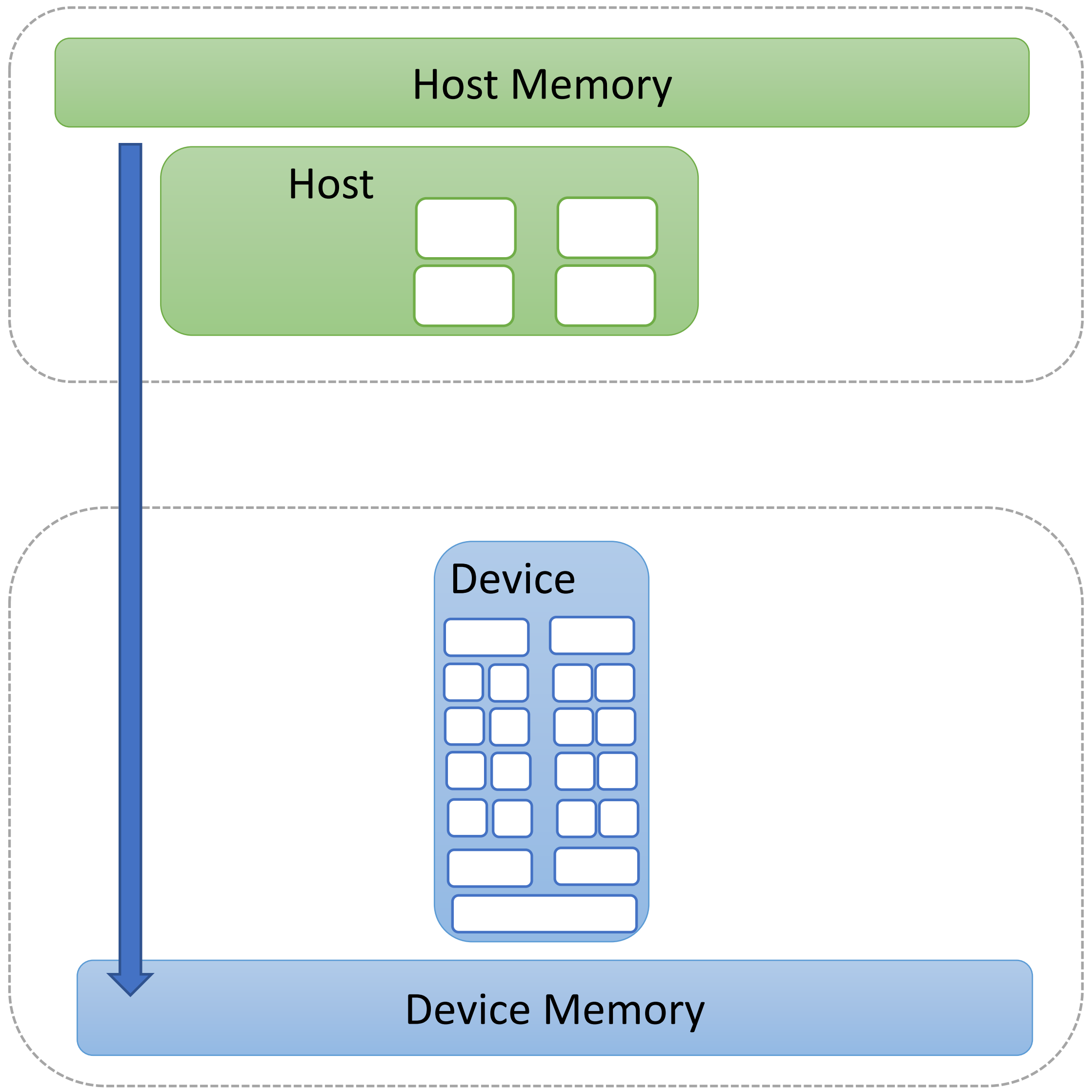
- `cudaDeviceSynchronize()` effectively synchronizes all threads in a grid; waits for all the threads in the kernel to complete before proceed.
- `__synchThreads()` synchronizes threads within a block

Classical example that requires `cudaDeviceSynchronize()`:

`printf` from the device

CUDA Functions for Data Movement are Synchronous

By default, data transfers are synchronous (the function does not return until the data transfer is complete), so `cudaMemcpy()` finishes execution before the CPU can move to other operations.



CUDA Vector Types

Vector types CUDA extends the standard C data types of length up to 4:

```
float4 f = (float4)(1.0f, 2.0f, 3.0f, 4.0f);
```

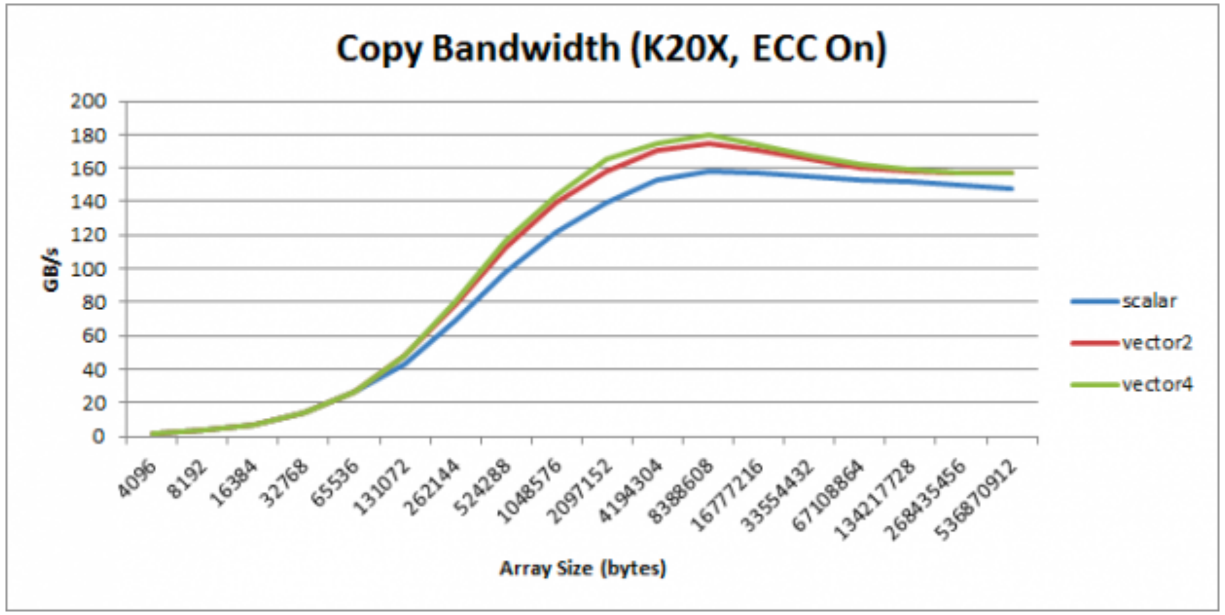
Individual components are accessed with the **suffixes** **`.x`, `.y`, `.z`, and `.w`.**
Accessing components beyond those declared for the vector type is an error:

```
float3 pos;
pos.z = 1.0f; // is legal
pos.w = 1.0f; // is illegal
```

Why to use CUDA vector types is good?

The use of vector types allows for vector loads and stores in CUDA C/C++ and helps increase bandwidth utilization

- Vectorized loads are a fundamental CUDA optimization that you should use when possible, because they increase bandwidth, reduce instruction count, and reduce latency.



https://devblogs.nvidia.com/parallelforall/cuda-pro-tip-increase-performance-with-vectorized-memory-access/

CUDA Data types for Index and Dimension Var.

CUDA uses the vector type `uint3` for the index variables, `blockIdx` and `threadIdx`:

- A `uint3` variable is a vector with three unsigned integer components.

CUDA uses the vector type `dim3` for the dimension variables, `gridDim` and `blockDim`:

- The `dim3` type is equivalent to `uint3` with unspecified entries set to 1. We will use `dim3` variables for specifying execution configuration.

To Summarize

- In a kernel, to identify what my thread ID is I need to obtain the ID of the block, the ID of the thread in the block and do a simple math (indexing)
- Kernel execution, except for data movement operations, are asynchronous so we might use synchronization to ensure correct execution
- CUDA provides vector types, similar to C structures, with maximum size 4, to improve memory bandwidth