CUDA-C Programming - Basics

Modern computing for physics

J.Pazzini
Padova University

Physics of Data
AA 2024-2025

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

Using the Jetson Nano

Our tiny GPU

First and foremost, thanks to NVIDIA for giving us these boards for free thanks to their Academic Hardware Grant process

Secondly, many thanks to Pietro Bernardi for having reworked inside-out the SD image, allowing us to use quite a set of functionalities

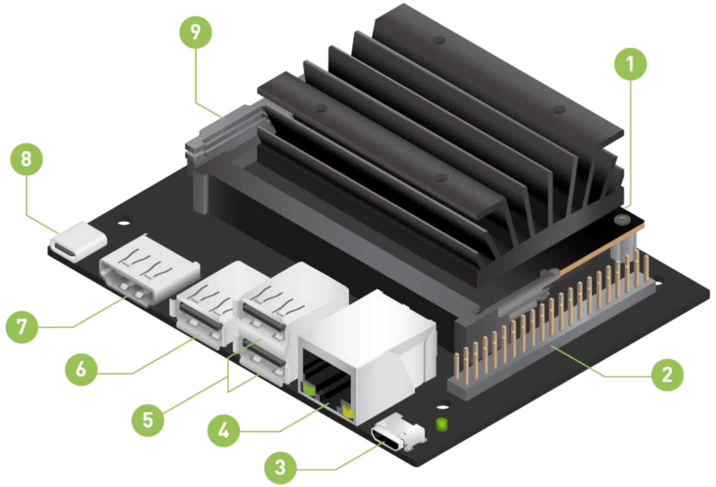
The Jetson Nano 2GB is a Developer Kit mostly used for teaching and small AI/robotics projects

It is powered via USB-C 5V (provided)

It can be used as a standalone “machine”, attaching it to video (via HDMI) and mouse/keyboard (via USB-A)

Or, as we mainly use it, as a “remote” integrated machine over network, either via RJ-45 or Micro-USB

→ we will connect to it via **`ssh`**



The board’s resources

This board integrates a CPU, some (very little) memory, and a GPU

CPU:	ARMv8 Processor (rev 1)
Memory:	2 GB LPDDR4 (LP stands for LowPower)
GPU:	Maxwell architecture, Compute Capability 5.3

4-core CPU (w/ 1 threads/core) @ max 1.48GHz

ARM ➔ Advanced RISC Machines ➔ RISC: Reduced Instruction Set Computer

64-bit Armv8-A architecture, also known as AArch64

Very simple ISA, with not much “intelligence”, and no Vector (SIMD-like) instructions

The board's resources

This board integrates a CPU, some (very little) memory, and a GPU

CPU:	ARMv8 Processor (rev 1)
Memory:	2 GB LPDDR4 (LP stands for LowPower)
GPU:	Maxwell architecture, Compute Capability 5.3

```
~$ lscpu
```

```
Architecture:        aarch64
CPU op-mode(s):      32-bit, 64-bit
Byte Order:          Little Endian
CPU(s):              4
On-line CPU(s) list: 0-3
Thread(s) per core:  1
Core(s) per socket:  4
Socket(s):           1
Vendor ID:           ARM
Model:               1
Model name:          Cortex-A57
Stepping:            r1p1
CPU max MHz:         1479.0000
CPU min MHz:         102.0000
BogoMIPS:            38.40
L1d cache:           128 KiB
L1i cache:           192 KiB
L2 cache:            2 MiB
Flags:               fp asimd evtstrm aes pmull sha1 sha2 crc32
```

The board's resources

This board integrates a CPU, some (very little) memory, and a GPU

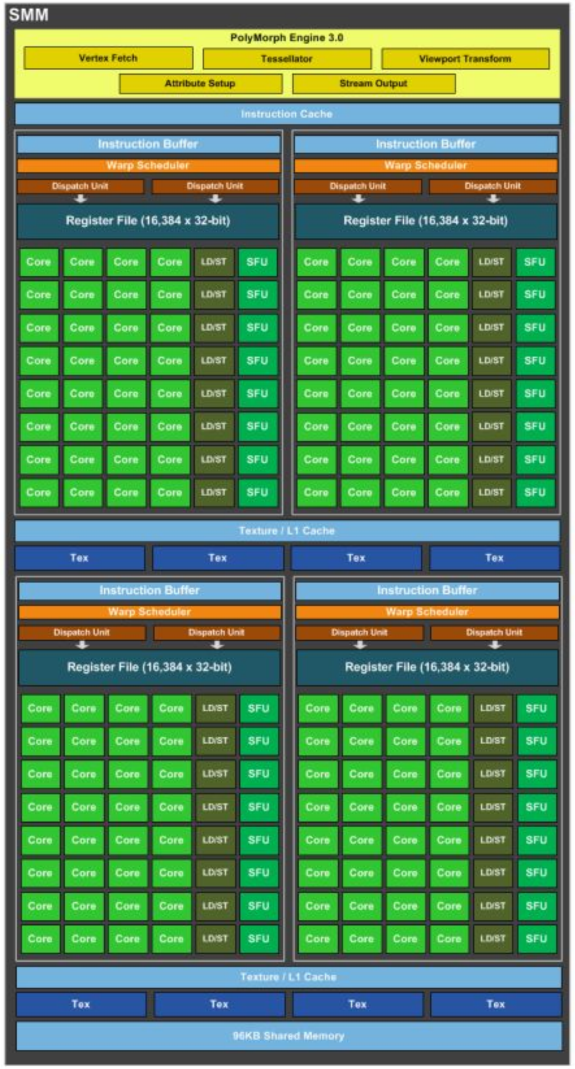
CPU: ARMv8 Processor (rev 1)
Memory: 2 GB LPDDR4 (LP stands for LowPower)
GPU: Maxwell architecture, Compute Capability 5.3

Maxwell architecture (same as the GeForce 980)

➔ Each SMP integrating 128 SPs (CUDA Cores)

The Jetson Nano only equips 1 SMP...

Definitely not enough for heavy computation,
but OK for testing and wetting our feet in CUDA

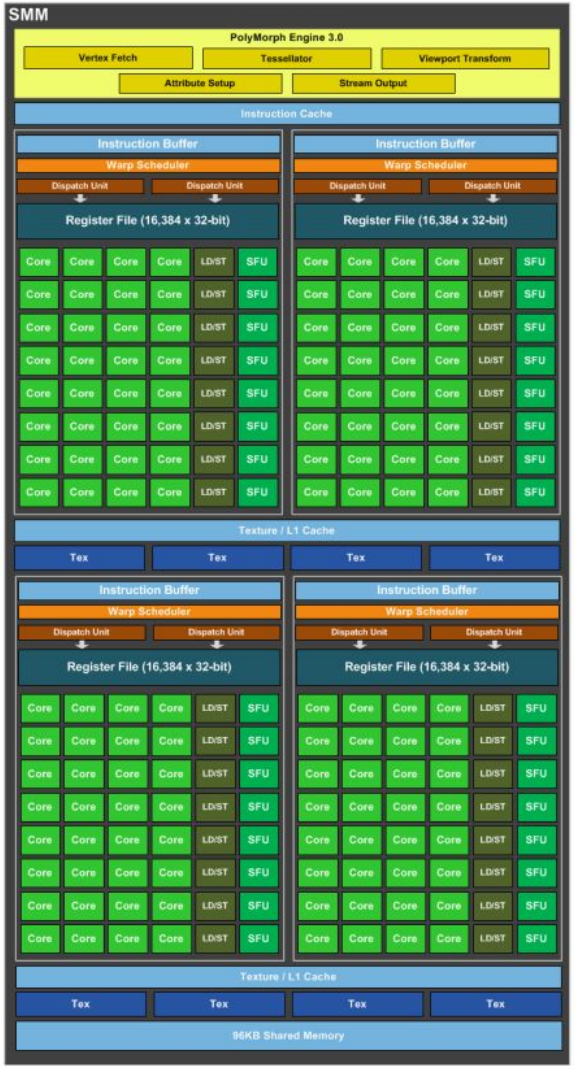


The board’s resources

This board integrates a CPU, some (very little) memory, and a GPU

CPU:	ARMv8 Processor (rev 1)
Memory:	2 GB LPDDR4 (LP stands for LowPower)
GPU:	Maxwell architecture, Compute Capability 5.3

```
Number of devices: 1
Device Number: 0
  Device name: NVIDIA Tegra X1
  n SMPs: 1
  n SPs: 128
  Clock rate (MHz): 900
  L2 Cache Size (KB): 262.1
  Memory Clock Rate (MHz): 12
  Memory Bus Width (bits): 64
  Peak Memory Bandwidth (GB/s): 0.2
  Total global memory (GB) 1.9
  Shared memory per block (Kbytes) 48.0
  Warp-size: 32
```



The board’s resources

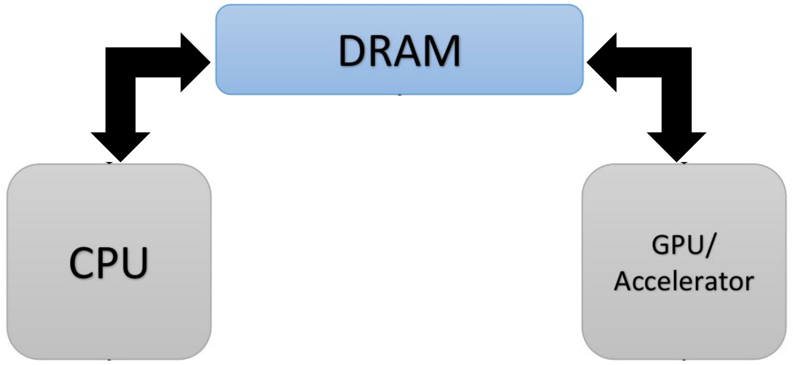
This board integrates a CPU, some (very little) memory, and a GPU

CPU:	ARMv8 Processor (rev 1)
Memory:	2 GB LPDDR4 (LP stands for LowPower)
GPU:	Maxwell architecture, Compute Capability 5.3

The Memory in the Jetson Nano is **shared** between the CPU and the GPU

➜ No real separation between the concept of “Host Memory” and “Device Memory”

However… we are going to do the same all the data transfers as if we were dealing with a standard Heterogeneous system



Using the Jetson from our laptop

Connecting the USB-A ➔ Micro-USB cable to the Jetson, we can log in from remote via ssh

In order to do that we must set up our laptop connection to allow Ethernet over USB

Once this is done, we can connect to the Jetson over ssh:

```
username : jetson
password : jetson
Jetson IP: 192.168.55.1
```

```
~$ ssh jetson@192.168.55.1
```

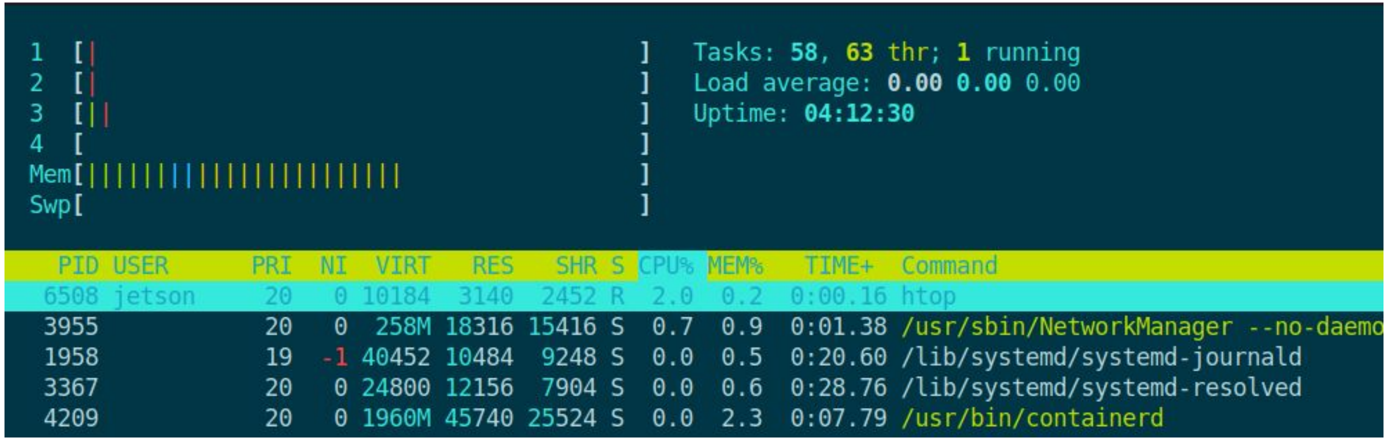
A conda environment contains all that required to run the lab activities (CUDA libraries, numba, cupy, etc):

```
(base) jetson@jetson:~$ conda activate mcp
```

Checking the status of the resource utilization

The usual `htop` process viewer shows the usage of the CPU and the memory (shared across CPU and GPU in this very specific circumstance)

```
(mcp) jetson@jetson:~$ htop
```



```
  1  [|                                              ]   Tasks: 58, 63 thr; 1 running
  2  [|                                              ]   Load average: 0.00 0.00 0.00
  3  [||                                             ]   Uptime: 04:12:30
  4  [                                               ]
  Mem[||||||||||||||||||||||||                       ]
  Swp[                                               ]

    PID USER       PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
   6508 jetson      20   0 10184  3140  2452 R  2.0  0.2  0:00.16 htop
   3955             20   0  258M 18316 15416 S  0.7  0.9  0:01.38 /usr/sbin/NetworkManager --no-daemo
   1958             19  -1 40452 10484  9248 S  0.0  0.5  0:20.60 /lib/systemd/systemd-journald
   3367             20   0 24800 12156  7904 S  0.0  0.6  0:28.76 /lib/systemd/systemd-resolved
   4209             20   0 1960M 45740 25524 S  0.0  2.3  0:07.79 /usr/bin/containerd
```

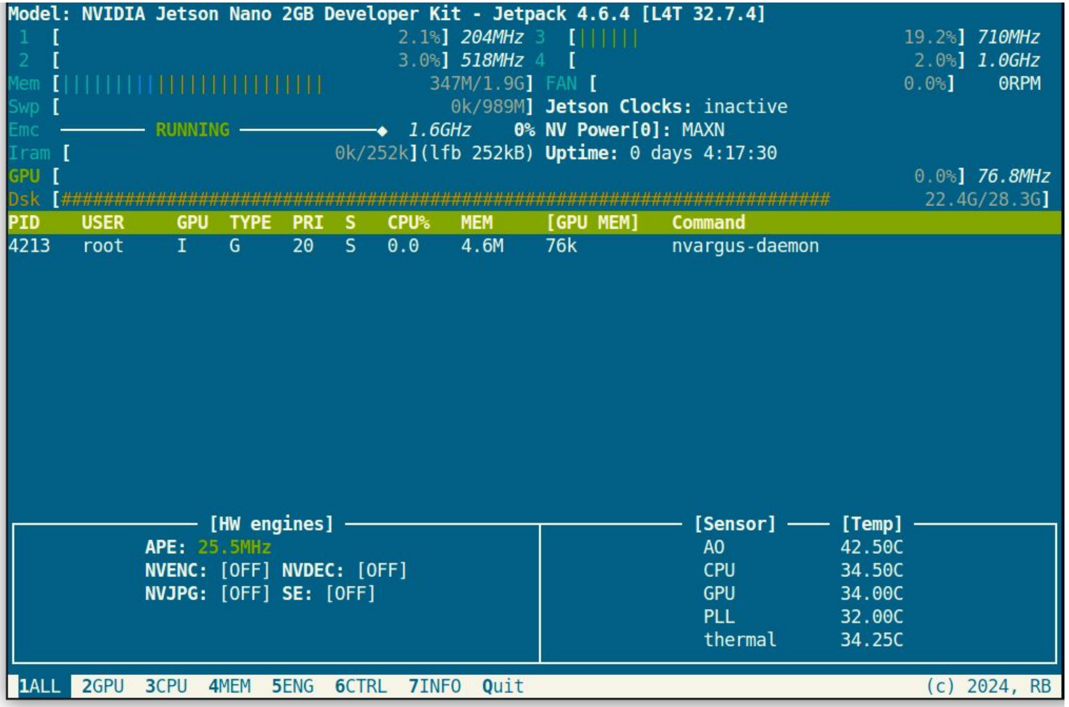
Checking the status of the resource utilization

A dedicated alternative process viewer, `jtop`, instead shows both the CPU and the GPU utilization

```
(mcp) jetson@jetson:~$ jtop
```

Navigate using the keyboard:

1 ➔ Overall stats
2 ➔ GPU stats
3 ➔ CPU stats
4 ➔ Memory stats
q ➔ exit



```
Model: NVIDIA Jetson Nano 2GB Developer Kit - Jetpack 4.6.4 [L4T 32.7.4]
 1  [                                   2.1%] 204MHz 3  [||||||                          19.2%] 710MHz
 2  [                                   3.0%] 518MHz 4  [                                 2.0%] 1.0GHz
Mem [|||||||||||||||||||||||||      347M/1.9G] FAN [                                  0.0%]    0RPM
Swp [                                0k/989M] Jetson Clocks: inactive
Emc ──────── RUNNING ─────────────◆ 1.6GHz      0% NV Power[0]: MAXN
Iram [                    0k/252k](lfb 252kB) Uptime: 0 days 4:17:30
GPU [                                                                               0.0%] 76.8MHz
Dsk [#########################################################################      22.4G/28.3G]
PID   USER      GPU  TYPE  PRI  S   CPU%   MEM     [GPU MEM]   Command
4213  root      I    G     20   S   0.0    4.6M    76k         nvargus-daemon


               [HW engines]                                [Sensor]      [Temp]
          APE: 25.5MHz                                      AO            42.50C
          NVENC: [OFF] NVDEC: [OFF]                         CPU           34.50C
          NVJPG: [OFF] SE: [OFF]                            GPU           34.00C
                                                            PLL           32.00C
                                                            thermal       34.25C
 1ALL  2GPU  3CPU  4MEM  5ENG  6CTRL  7INFO  Quit                                    (c) 2024, RB
```

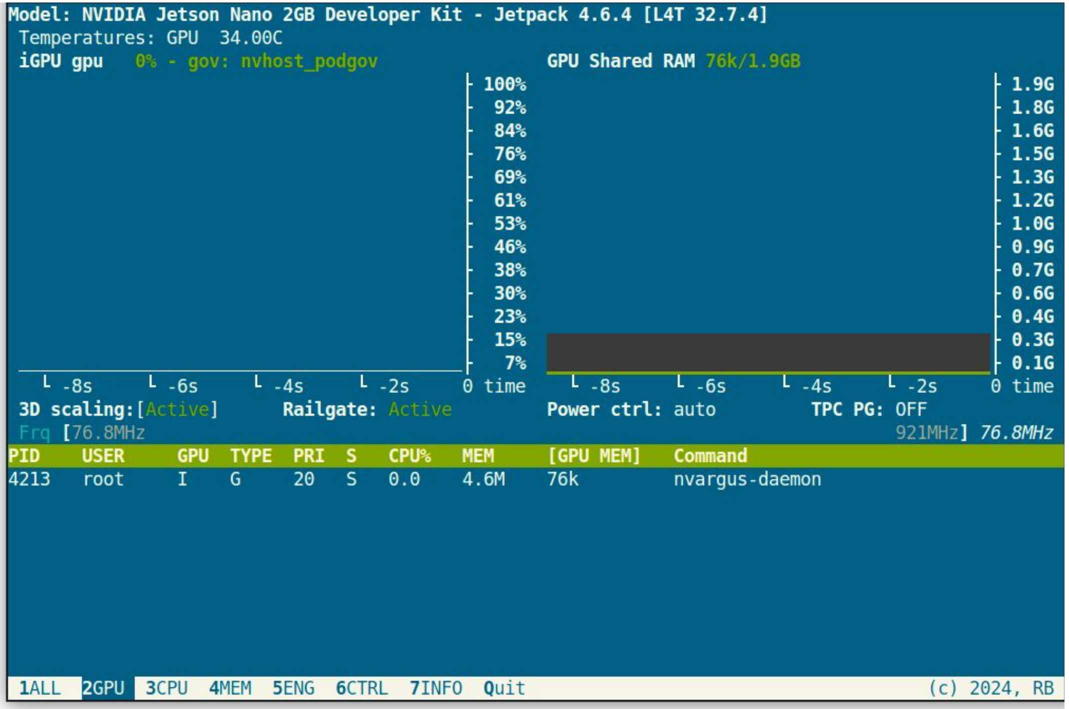
Checking the status of the resource utilization

A dedicated alternative process viewer, `jtop`, instead shows both the CPU and the GPU utilization

```
(mcp) jetson@jetson:~$ jtop
```

Navigate using the keyboard:

1 → Overall stats
2 → GPU stats
3 → CPU stats
4 → Memory stats
q → exit



```
Model: NVIDIA Jetson Nano 2GB Developer Kit - Jetpack 4.6.4 [L4T 32.7.4]
 Temperatures: GPU  34.00C
 iGPU gpu    0% - gov: nvhost_podgov                     GPU Shared RAM 76k/1.9GB
                                          ├ 100%                                         ├ 1.9G
                                          ├ 92%                                          ├ 1.8G
                                          ├ 84%                                          ├ 1.6G
                                          ├ 76%                                          ├ 1.5G
                                          ├ 69%                                          ├ 1.3G
                                          ├ 61%                                          ├ 1.2G
                                          ├ 53%                                          ├ 1.0G
                                          ├ 46%                                          ├ 0.9G
                                          ├ 38%                                          ├ 0.7G
                                          ├ 30%                                          ├ 0.6G
                                          ├ 23%                                          ├ 0.4G
                                          ├ 15%                                          ├ 0.3G
                                          ├ 7%                                           ├ 0.1G
   └ -8s     └ -6s     └ -4s     └ -2s    0 time     └ -8s     └ -6s     └ -4s     └ -2s    0 time
 3D scaling:[Active]        Railgate: Active         Power ctrl: auto          TPC PG: OFF
 Frq [76.8MHz                                                                  921MHz] 76.8MHz
PID    USER      GPU  TYPE  PRI  S   CPU%   MEM     [GPU MEM]    Command
4213   root      I    G     20   S   0.0    4.6M    76k          nvargus-daemon

 1ALL  2GPU  3CPU  4MEM  5ENG  6CTRL  7INFO  Quit                                    (c) 2024, RB
```

Text editors

The Jetson OS includes the basic shell-based editors: **`vi/vim`** and **`nano/pico`**

It would be useful to all of you to get a minimal expertise with at least one of them

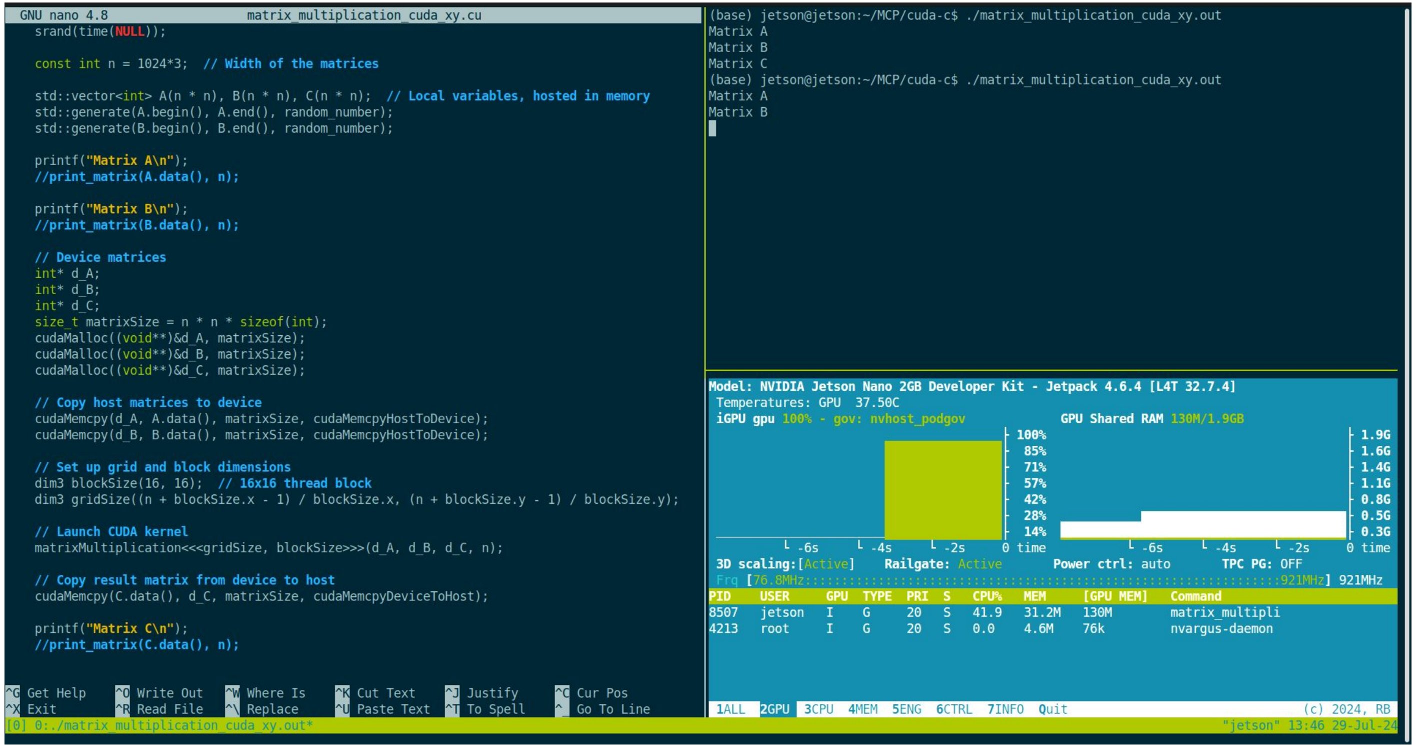
Text editors

The Jetson OS includes the basic shell-based editors: **`vi/vim`** and **`nano/pico`**

It would be useful to all of you to get a minimal expertise with at least one of them

If using the text editors from shell, it can be useful to use **`tmux`** a “terminal multiplexer” for handling on the same shell multiple terminals

(e.g. one for writing code, another to execute it, another to monitor the resources)



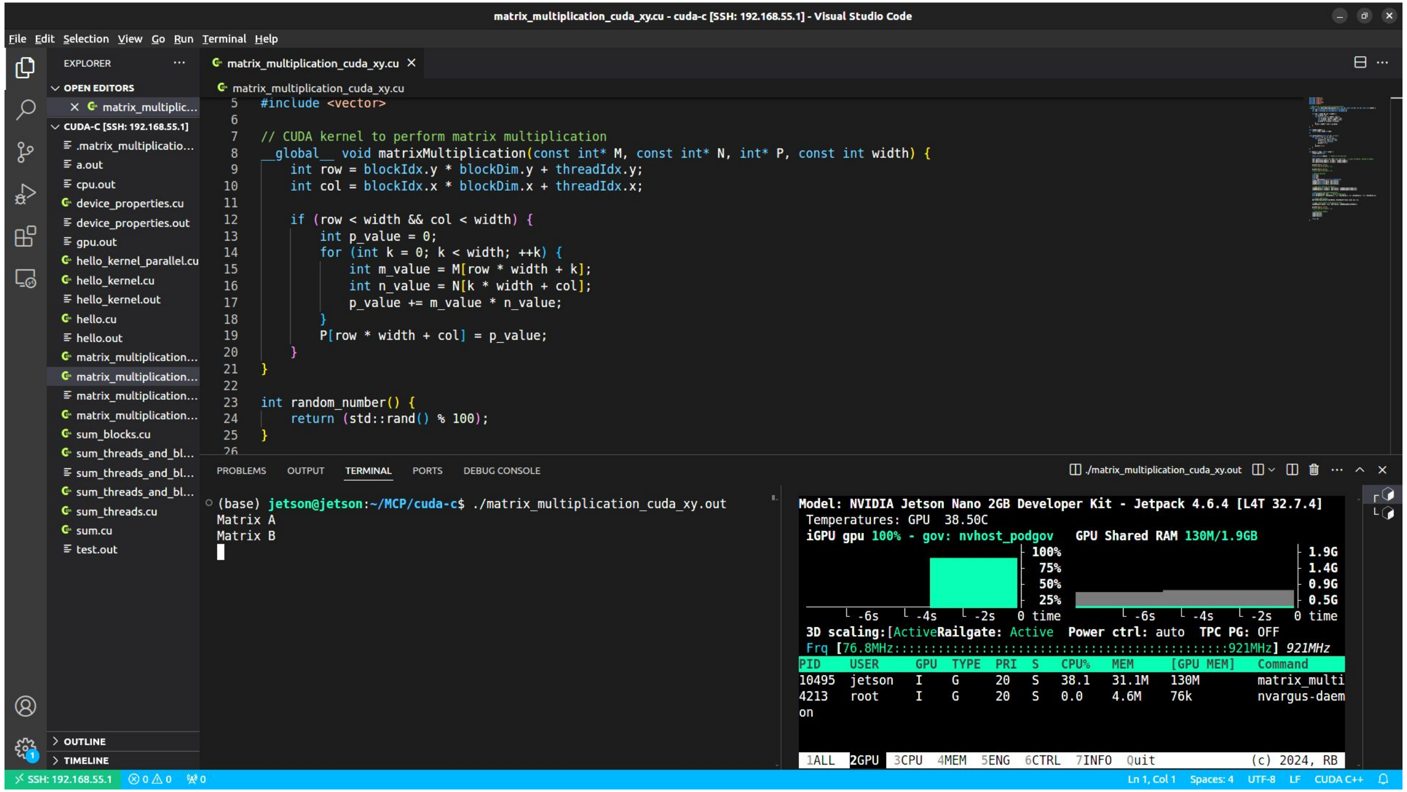
```
  GNU nano 4.8                    matrix_multiplication_cuda_xy.cu
    srand(time(NULL));

    const int n = 1024*3;  // Width of the matrices

    std::vector<int> A(n * n), B(n * n), C(n * n);  // Local variables, hosted in memory
    std::generate(A.begin(), A.end(), random_number);
    std::generate(B.begin(), B.end(), random_number);

    printf("Matrix A\n");
    //print_matrix(A.data(), n);

    printf("Matrix B\n");
    //print_matrix(B.data(), n);

    // Device matrices
    int* d_A;
    int* d_B;
    int* d_C;
    size_t matrixSize = n * n * sizeof(int);
    cudaMalloc((void**)&d_A, matrixSize);
    cudaMalloc((void**)&d_B, matrixSize);
    cudaMalloc((void**)&d_C, matrixSize);

    // Copy host matrices to device
    cudaMemcpy(d_A, A.data(), matrixSize, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, B.data(), matrixSize, cudaMemcpyHostToDevice);

    // Set up grid and block dimensions
    dim3 blockSize(16, 16);  // 16x16 thread block
    dim3 gridSize((n + blockSize.x - 1) / blockSize.x, (n + blockSize.y - 1) / blockSize.y);

    // Launch CUDA kernel
    matrixMultiplication<<<gridSize, blockSize>>>(d_A, d_B, d_C, n);

    // Copy result matrix from device to host
    cudaMemcpy(C.data(), d_C, matrixSize, cudaMemcpyDeviceToHost);

    printf("Matrix C\n");
    //print_matrix(C.data(), n);

^G Get Help    ^O Write Out   ^W Where Is    ^K Cut Text    ^J Justify     ^C Cur Pos
^X Exit        ^R Read File   ^\ Replace     ^U Paste Text  ^T To Spell    ^_ Go To Line
```

```
(base) jetson@jetson:~/MCP/cuda-c$ ./matrix_multiplication_cuda_xy.out
Matrix A
Matrix B
Matrix C
(base) jetson@jetson:~/MCP/cuda-c$ ./matrix_multiplication_cuda_xy.out
Matrix A
Matrix B
```

```
Model: NVIDIA Jetson Nano 2GB Developer Kit - Jetpack 4.6.4 [L4T 32.7.4]
 Temperatures: GPU  37.50C
 iGPU gpu 100% - gov: nvhost_podgov                  GPU Shared RAM 130M/1.9GB
                                    100%                                        1.9G
                                     85%                                        1.6G
                                     71%                                        1.4G
                                     57%                                        1.1G
                                     42%                                        0.8G
                                     28%                                        0.5G
                                     14%                                        0.3G
          └ -6s    └ -4s    └ -2s  0 time          └ -6s    └ -4s    └ -2s    0 time
 3D scaling:[Active]   Railgate: Active       Power ctrl: auto      TPC PG: OFF
 Frq [76.8MHz::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::921MHz] 921MHz
PID    USER     GPU  TYPE  PRI  S   CPU%   MEM    [GPU MEM]   Command
8507   jetson   I    G     20   S   41.9   31.2M  130M        matrix_multipli
4213   root     I    G     20   S   0.0    4.6M   76k         nvargus-daemon

 1ALL  2GPU  3CPU  4MEM  5ENG  6CTRL  7INFO  Quit                          (c) 2024, RB
[0] 0:./matrix_multiplication_cuda_xy.out*                     "jetson" 13:46 29-Jul-24
```

Don’t panic…

If you prefer the “safety” of a graphic editor or an IDE (integrated development environment), you can use for example VSCode to do all of this, by establishing a **remote session over ssh**



```cuda
#include <vector>

// CUDA kernel to perform matrix multiplication
__global__ void matrixMultiplication(const int* M, const int* N, int* P, const int width) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < width && col < width) {
        int p_value = 0;
        for (int k = 0; k < width; ++k) {
            int m_value = M[row * width + k];
            int n_value = N[k * width + col];
            p_value += m_value * n_value;
        }
        P[row * width + col] = p_value;
    }
}

int random_number() {
    return (std::rand() % 100);
}
```

```
(base) jetson@jetson:~/MCP/cuda-c$ ./matrix_multiplication_cuda_xy.out
Matrix A
Matrix B
```

```
Model: NVIDIA Jetson Nano 2GB Developer Kit - Jetpack 4.6.4 [L4T 32.7.4]
 Temperatures: GPU  38.50C
 iGPU gpu 100% - gov: nvhost_podgov   GPU Shared RAM 130M/1.9GB
                                 100%                              1.9G
                                  75%                              1.4G
                                  50%                              0.9G
                                  25%                              0.5G
      └ -6s   └ -4s   └ -2s   0 time      └ -6s   └ -4s   └ -2s   0 time
 3D scaling:[ActiveRailgate: Active  Power ctrl: auto  TPC PG: OFF
 Frq [76.8MHz:::::::::::::::::::::::::::::::::::::::::::::921MHz] 921MHz
PID    USER     GPU  TYPE  PRI  S   CPU%   MEM     [GPU MEM]   Command
10495  jetson   I    G     20   S   38.1   31.1M   130M        matrix_multi
4213   root     I    G     20   S   0.0    4.6M    76k         nvargus-daem
on

 1ALL  2GPU  3CPU  4MEM  5ENG  6CTRL  7INFO  Quit            (c) 2024, RB
```

Hello Jetson

```
you@yourpc:~$ scp local_file jetson@192.168.55.1:path/to/folder
```

- Turn on the Jetson by plugging it to the USB-C power adapter
- Connect the Jetson to your laptop via the micro-USB cable
- `ssh` to it and take confidence with `htop` and `jtop`
- Write (using a shell-based editor or VSCode in remote) or copy* the basic C hello world code

```c
// Include the standard input/output library
#include <stdio.h>

// Main function, entry point of the program
int main(int argc, char **argv){

   // Print "Hello world" to the console
   printf("Hello world\n");

   // Return 0 to indicate successful execution
   return 0;
}
```

Using `hello.cu`

- Compile it with **`nvcc`** and execute it
- ...congratulations!

CUDA-C/C++

CUDA Functions and Kernels

A function which runs on a GPU is called **kernel**

- When a kernel is **launched** on a GPU, a number (possibly thousands) of threads will execute its code on the SMPs
- Programmers choose the number of threads to run
    - Each thread acts on a different data element independently
      ➔ the GPU parallelism is very close to the SPMD paradigm

```
__global__ void a_kernel(void){

   // Code executed on the GPU

}
```

`__global__` indicates a **kernel function**
- launched by the host
- executed on the device

When launched from host and executed on device, functions must return `void`

- CUDA kernels are **asynchronous** ➔ control is returned immediately to the host code
    - If necessary to complete the execution of a kernel before advancing, an explicit synchronization is required

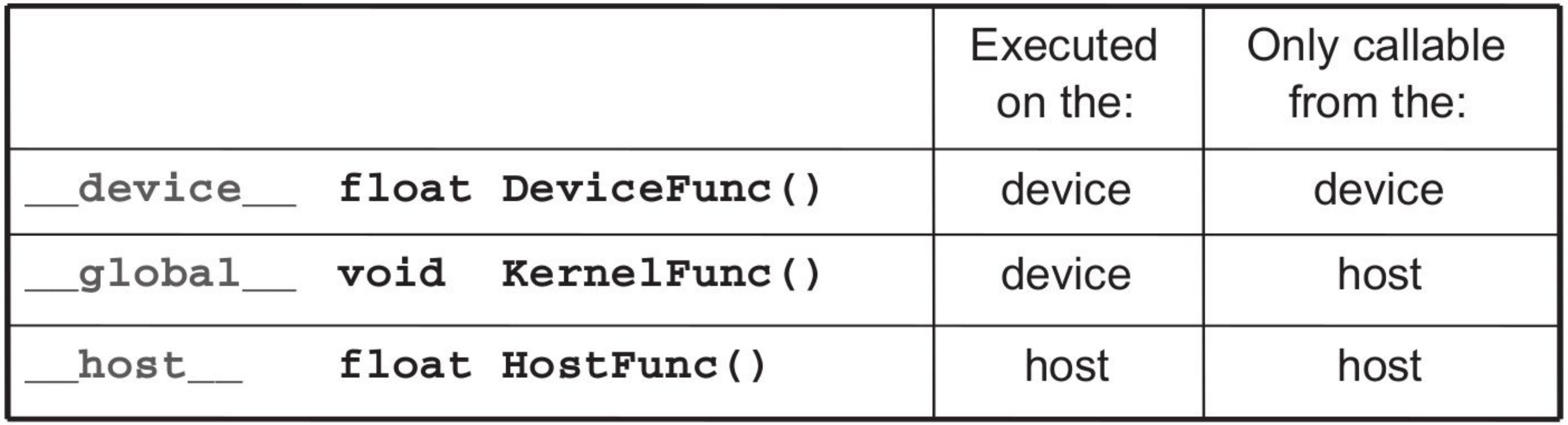
CUDA Functions and Kernels

	Executed on the:	Only callable from the:
`__device__ float DeviceFunc()`	device	device
`__global__ void KernelFunc()`	device	host
`__host__ float HostFunc()`	host	host

By default, all functions in a CUDA program are `__host__` functions if they do not have any of the CUDA keywords in their declaration
⇒ `__host__` functions are simple C/C++ functions (can return int/float,...)

`__device__` functions are CUDA functions that can be called only from an active kernel (similarly to `__host__` functions can return int/float/...)

CUDA Functions and Kernels



	Executed on the:	Only callable from the:
`__device__ float DeviceFunc()`	device	device
`__global__ void KernelFunc()`	device	host
`__host__ float HostFunc()`	host	host

`nvcc` separates source code into host and device components:

- Device functions processed by NVIDIA compiler (**`nvcc`**)
- Host functions processed by the C compiler (e.g. **`gcc`**)

One can use both **`__host__`** and **`__device__`** in a function declaration to generate (at compile time) two versions of the same function

Kernel launch

```
a_kernel<<<1,4>>>();
```

Triple angle brackets (*chevrons*) mark a call to device code
➔ mostly referred to as a **kernel launch**

The parameters inside the triple angle brackets are the CUDA kernel execution configuration

The parameters inside the “standard” parentheses are instead the kernel input data (if any)

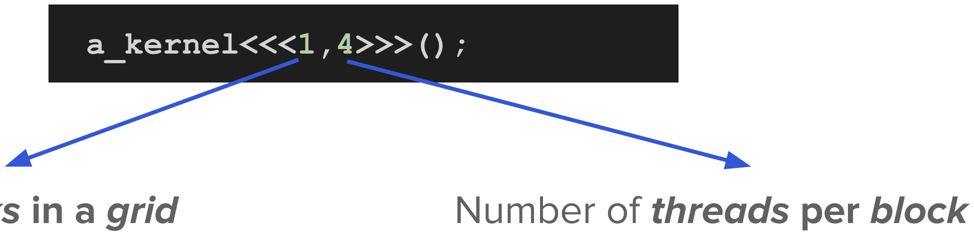
Kernel launch

```
a_kernel<<<1,4>>>();
```

Triple angle brackets (chevrons) mark a call to device code
➜ mostly referred to as a **kernel launch**

The parameters inside the triple angle brackets are the CUDA kernel execution configuration

The parameters inside the “standard” parentheses are instead the kernel input data (if any)



```
a_kernel<<<1,4>>>();
```

Number of ***blocks* in a *grid***
Number of ***threads* per *block***

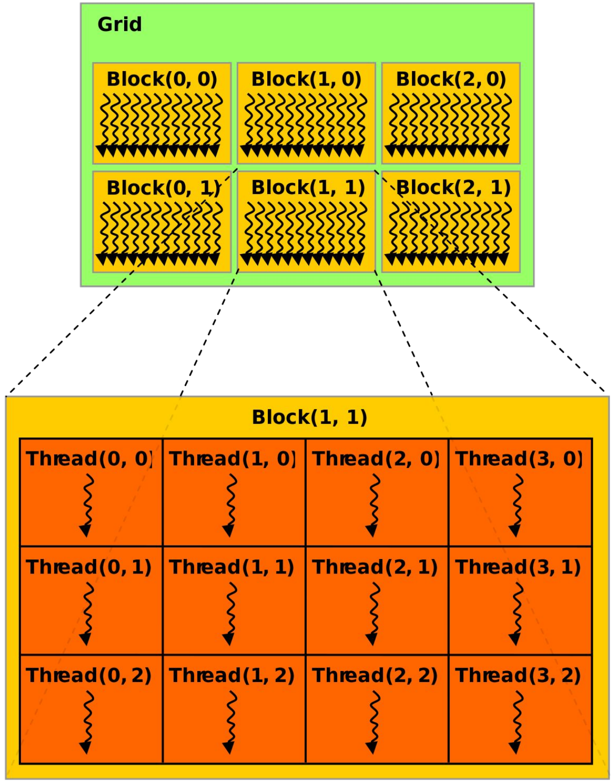
Number of times the kernel is executed = **n_blocks * n_threads_per_block**

Grid/blocks/threads

In order to compute N elements on the GPU in parallel, at least N concurrent threads must be created on the device

The CUDA abstraction to issue parallelization relies on the organization of the concurrent threads in a hierarchical 3-layer structure

1. **GRID** → The whole set of kernels generated in a kernel launch is collectively called a *grid*
2. **BLOCK** → Each grid is subdivided into smaller groups of threads called *blocks*
3. **THREAD** → Each block contains multiple threads

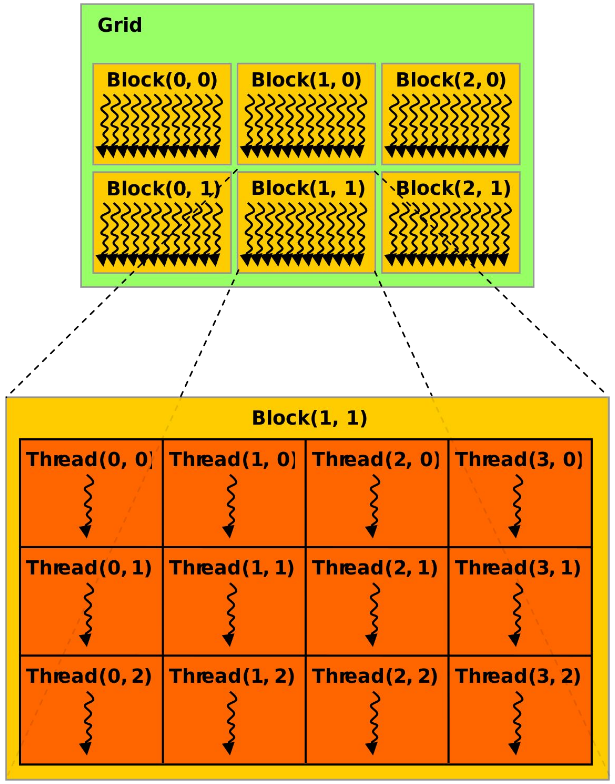


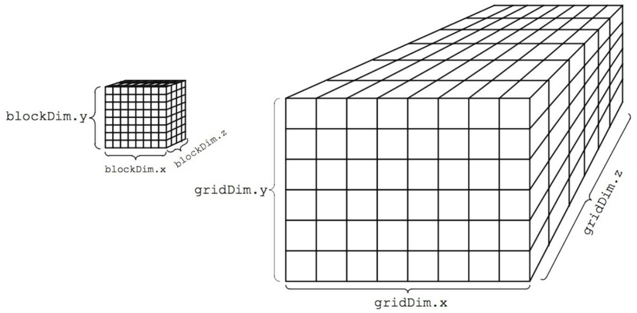
Grid/blocks/threads

In order to compute N elements on the GPU in parallel, at least N concurrent threads must be created on the device

The CUDA abstraction to issue parallelization relies on the organization of the concurrent threads in a hierarchical 3-layer structure

1. **GRID** ➔ The whole set of kernels generated in a kernel launch is collectively called a *grid*
2. **BLOCK** ➔ Each grid is subdivided into smaller groups of threads called *blocks*
3. **THREAD** ➔ Each block contains multiple threads



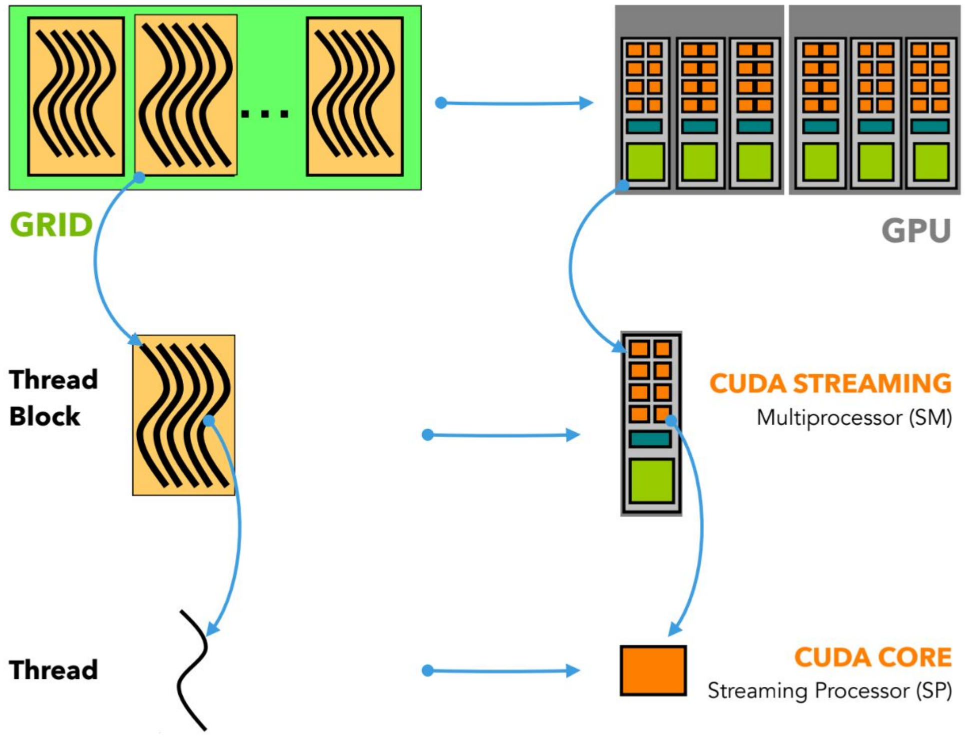


Mapping it to the HW

Kernel launched are organized in a grid of thread blocks running on the GPU

A block of threads is executed on one SMP and does not “migrate” across different SMPs

Each thread of a block is executed on a given SP of a SMP



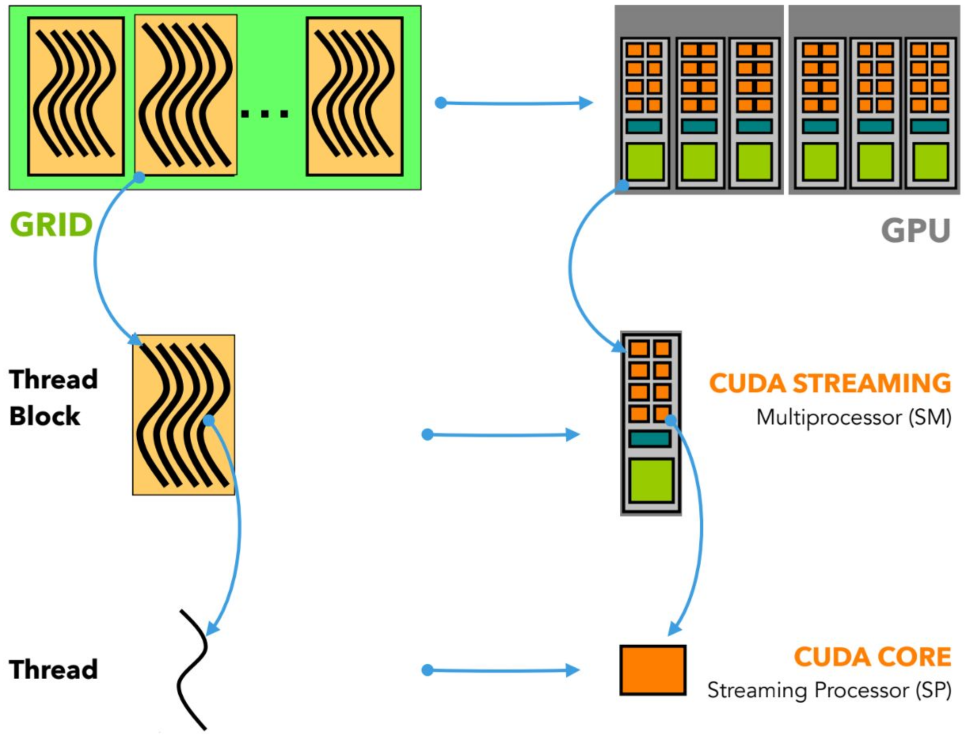
Mapping it to the HW

Kernel launched are organized in a grid of thread blocks running on the GPU

A block of threads is executed on one SMP and does not “migrate” across different SMPs

Each thread of a block is executed on a given SP of a SMP

➔ **Threads within a block can cooperate via shared memory (of a SMP)**



Mapping it to the HW

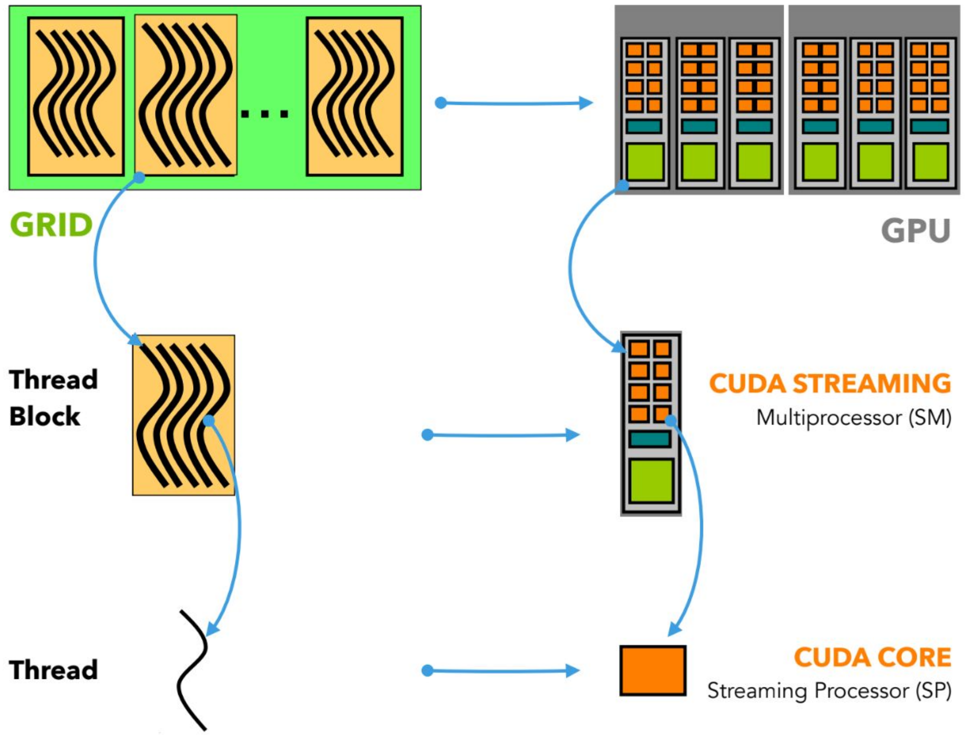
Kernel launched are organized in a grid of thread blocks running on the GPU

A block of threads is executed on one SMP and does not “migrate” across different SMPs

Each thread of a block is executed on a given SP of a SMP

→ Threads within a block can cooperate via shared memory (of a SMP)

→ Threads within a block can synchronize (wait for others to complete)



Mapping it to the HW

Kernel launched are organized in a grid of thread blocks running on the GPU

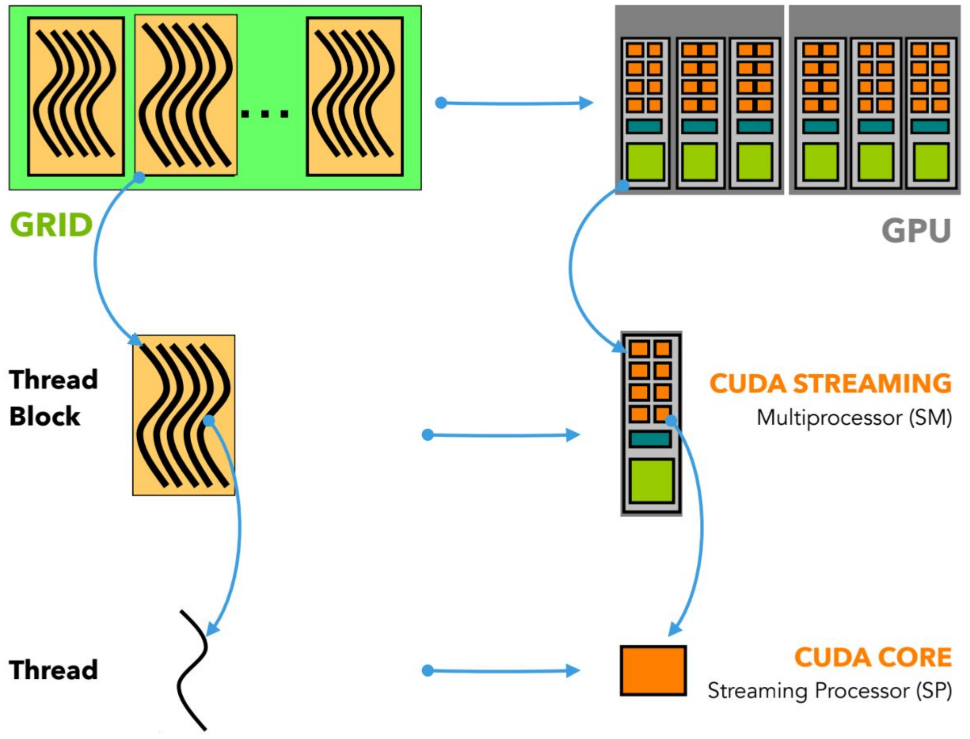
A block of threads is executed on one SMP and does not “migrate” across different SMPs

Each thread of a block is executed on a given SP of a SMP

→ Threads within a block can cooperate via shared memory (of a SMP)

→ Threads within a block can synchronize (wait for others to complete)

→ Threads in different blocks cannot cooperate (unless via global memory)



Mapping it to the HW

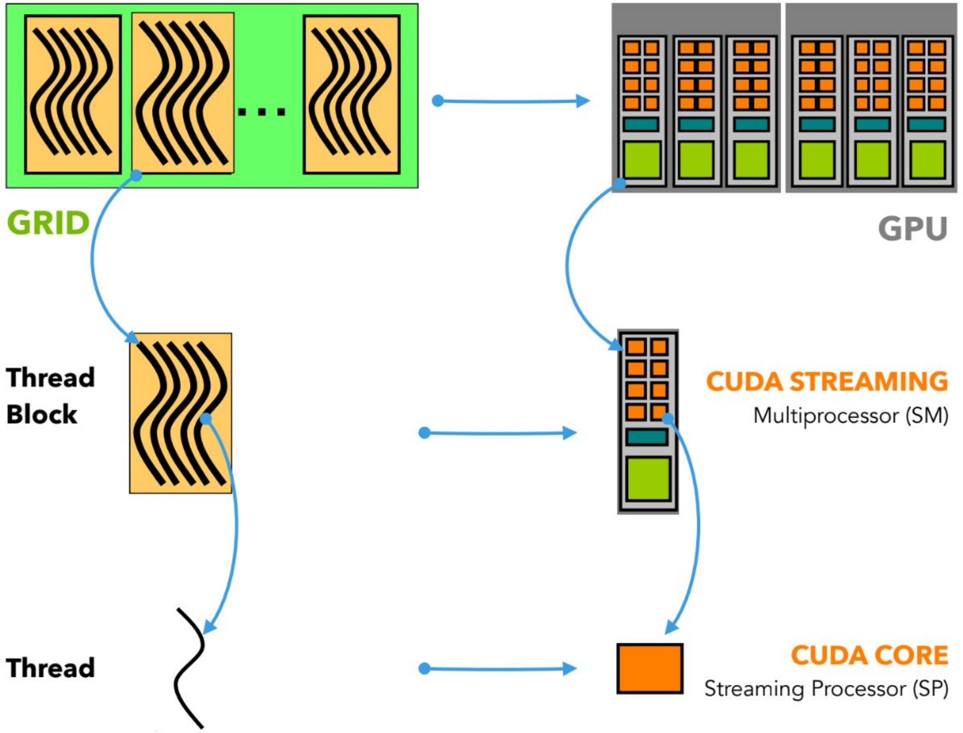
```
a_kernel<<<1,4>>>();
```

1 block ↔ 1 SMP

4 threads per block ↔ 4 SPs per SMP

We are launching a grid of 4 threads, organized in 1 block

⇒ all on the same SMP
⇒ they can “cooperate”



Mapping it to the HW

```
a_kernel<<<4,1>>>();
```

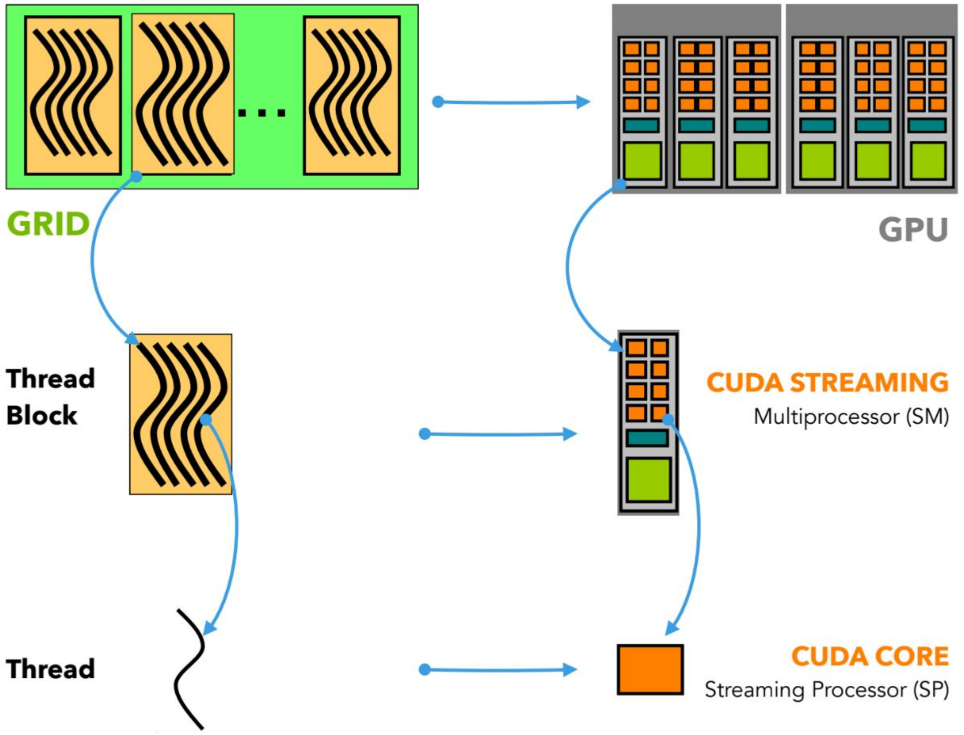
4 blocks ↔ 4 SMPs

1 thread per block ↔ 1 SP per SMP

We are launching a grid of 4 threads, organized in 4 blocks

⇒ all on different SMPs

⇒ they cannot “cooperate”



Our first kernel

Using `hello_kernel.cu`

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

int main(int argc, char **argv){

   ...
   a_kernel<<<1,1>>>();
   ...

}
```

Using `hello_kernel_parallel.cu`

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

int main(int argc, char **argv){

   ...
   a_kernel<<<1,4>>>();
   ...

}
```

Coordinating Host and Device

Kernel launches are *asynchronous*:

- control is returned to the host thread before the device has completed the requested task
- CPU needs to synchronize before consuming the results

  ➔ This is typically done via Data transfer between Device and Host

  ➔ Alternatively, one can call the following function to force the host application to wait for all kernels to complete: `cudaDeviceSynchronize(void);`

```
int main(int argc, char **argv){
   // CPU can do something here
   ...
   // Asynchronous kernel launch (non-blocking)
   my_kernel<<<N_BLOCKS,N_THREADS_PER_BLOCK>>>();

   // CPU can continue to do something here
   ...
   // When necessary to force the synchronization
   cudaDeviceSynchronize();
   // CPU execution is blocked until all threads are done
   ...
}
```

Coordinating Host and Device

Kernel launches are *asynchronous*:

- control is returned to the host thread before the device has completed the requested task
- CPU needs to synchronize before consuming the results

  ➜ This is typically done via Data transfer between Device and Host

  ➜ Alternatively, one can call the following function to force the host application to wait for all kernels to complete: `cudaDeviceSynchronize(void);`

Data transfers between Host and Device are *synchronous*:

- Kernels operate out of device memory. Device pointers point to GPU memory
  1. Host memory allocation and declaration (input and output data)
  2. Device memory allocation
  3. Host➜Device data transfer for kernel execution
  4. Device➜Host data transfer for result utilization on Host
  5. Free up memory

⇒ CUDA offers APIs for handling device memory:	↕	**`cudaMalloc()`**	**`cudaFree()`**	**`cudaMemcpy()`**
similar to the C equivalents:		**`malloc()`**	**`free()`**	**`memcpy()`**

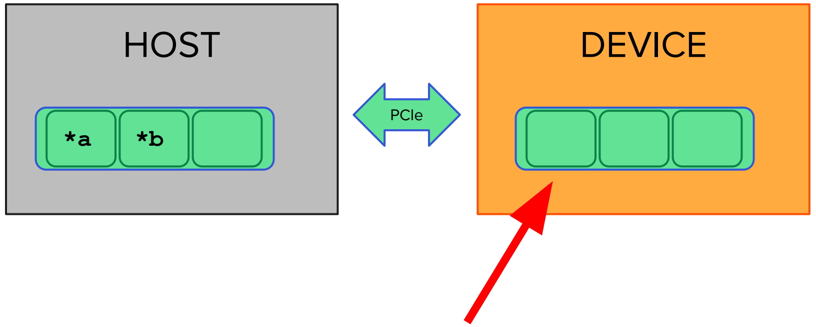
Memory handling

```
int main(int argc, char **argv){

 // Local variables, hosted in the host memory
 int a, b, c;

 // Pointers to device (GPU) memory
 int *d_a, *d_b, *d_c;

 // Size of the memory required for each integer
 int size = sizeof(int);

 // Allocate space on the device for the copies
 // of all the variables (both inputs and output)
 cudaMalloc((void **)&d_a, size);
 cudaMalloc((void **)&d_b, size);
 cudaMalloc((void **)&d_c, size);

 ...
```

Using `scalar_sum.cu`

```
cudaMalloc(void **p, size_t size);
```

- allocates `size` bytes of GPU **global** memory
- `p` is a valid device memory address



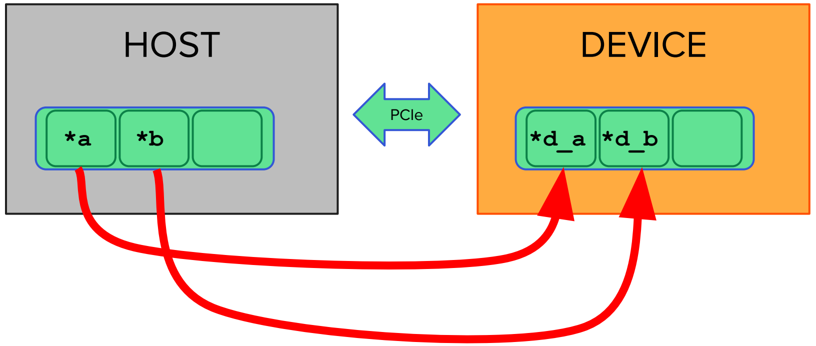
Memory handling

- `cudaMemcpyHostToHost`
- **`cudaMemcpyHostToDevice`**
- `cudaMemcpyDeviceToHost`
- `cudaMemcpyDeviceToDevice`

```
cudaMemcpy(void *dst, void *src, size_t size, cudaMemcpyKind size);
```

- copy bytes from source to destination buffers
- `dst` is a valid destination memory address
- `src` is a valid source memory address

```
...

 // Copy the input variables from host to device
 cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
 cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);

...
```

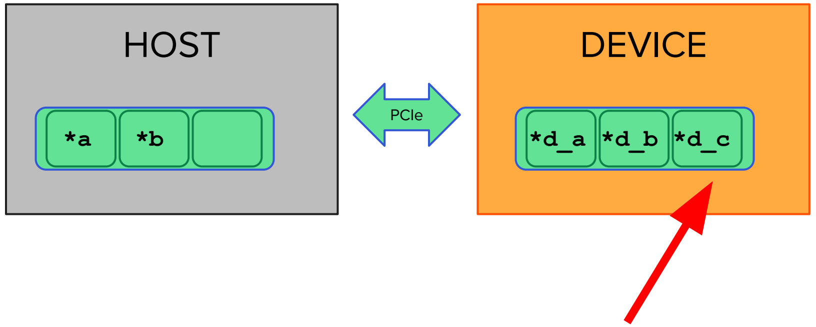


Memory handling

```
// Define a CUDA kernel that adds two integers
__global__ void sum_kernel(int *x, int *y, int *res){
   // All operands are passed by reference
   //
   // The operation is executed on the device,
   // so the variables x, y, res must point to GPU memory

   *res = *x + *y;  // Perform addition on the GPU
}
```

- Launch kernel on inputs `d_a, d_b`
- Store output on the device global memory as `d_c`

```
...


// Launch kernel with custom dimensions
sum_kernel<<<1, 1>>>(d_a, d_b, d_c);


...
```



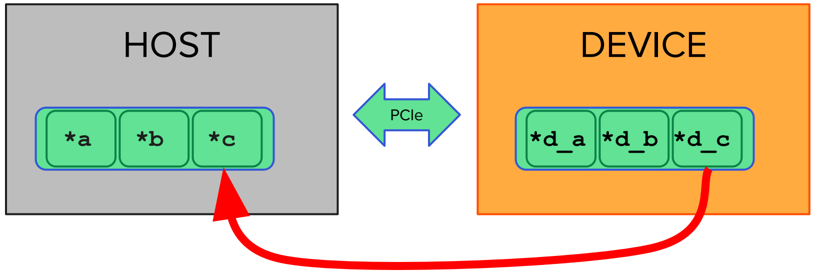
Memory handling

- `cudaMemcpyHostToHost`
- `cudaMemcpyHostToDevice`
- **`cudaMemcpyDeviceToHost`**
- `cudaMemcpyDeviceToDevice`

```
cudaMemcpy(void *dst, void *src, size_t size, cudaMemcpyKind size);
```

- copy bytes from source to destination buffers
- `dst` is a valid destination memory address
- `src` is a valid source memory address

```
...

// Copy output result from device to host
cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);

...
```

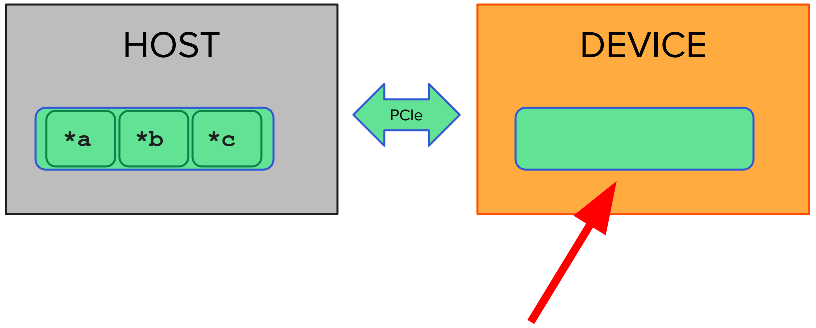


Memory handling

```
cudaFree(void *p);
```

- p is a valid device memory address

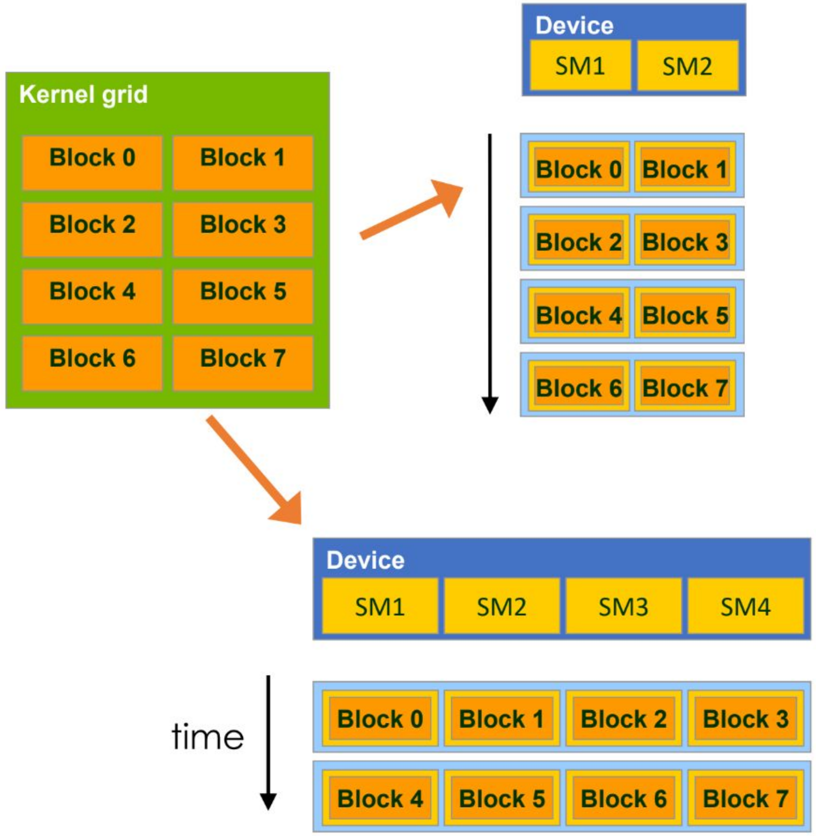
```
...

   // Free up all used device memory
   cudaFree(d_a);
   cudaFree(d_b);
   cudaFree(d_c);

...
```



How CUDA handles this

When a CUDA kernel is invoked

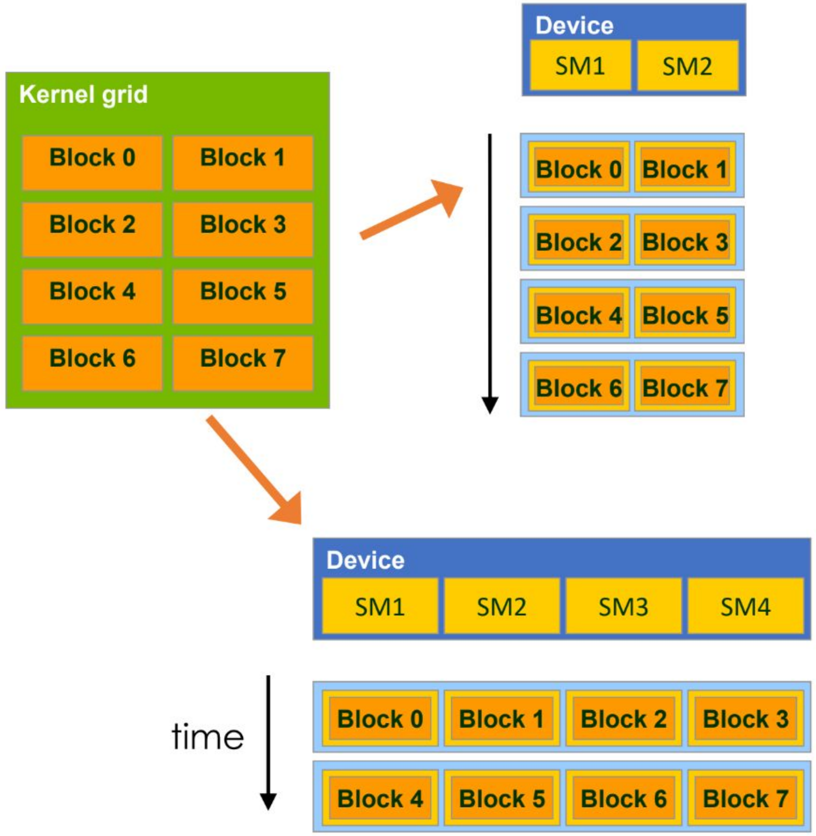
1. The CUDA runtime system maintains a list of active blocks
   - ➜ a max number of blocks can be assigned to each SMP
   - ➜ CUDA imposes a max 1024 threads in a block
   - ➜ the thread block assigns new blocks to SMPs as they complete



How CUDA handles this

When a CUDA kernel is invoked

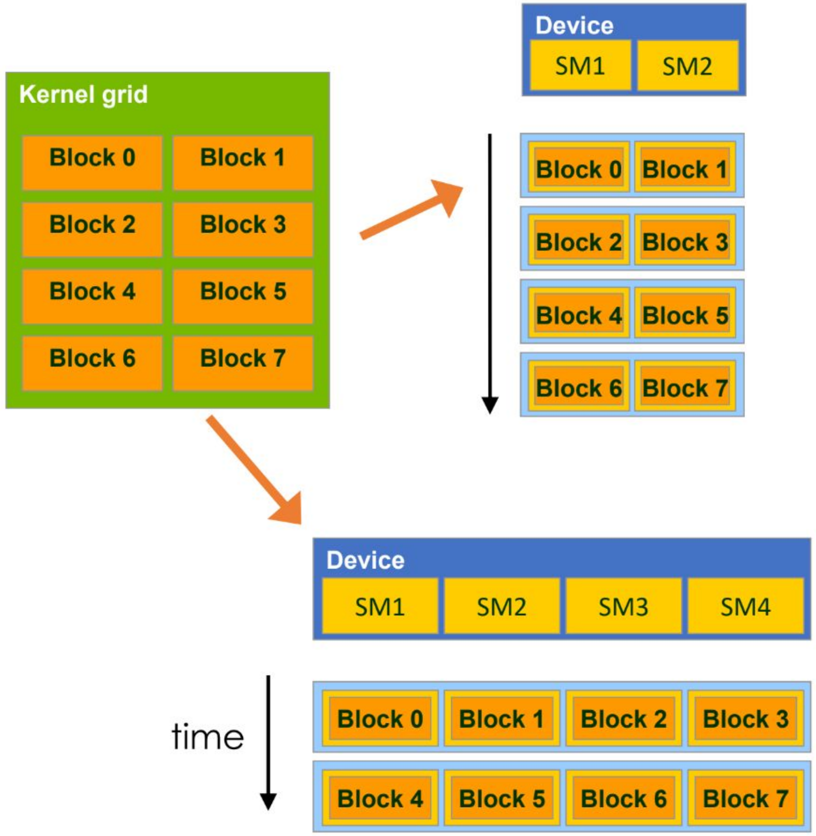
1. The CUDA runtime system maintains a list of active blocks
   - a max number of blocks can be assigned to each SMP
   - CUDA imposes a max 1024 threads in a block
   - the thread block assigns new blocks to SMPs as they complete
2. Each thread blocks is assigned to SMPs in a round robin mode
   - the thread block remains on the SMP until all its threads have finished



How CUDA handles this

When a CUDA kernel is invoked

1. The CUDA runtime system maintains a list of active blocks
   - a max number of blocks can be assigned to each SMP
   - CUDA imposes a max 1024 threads in a block
   - the thread block assigns new blocks to SMPs as they complete
2. Each thread blocks is assigned to SMPs in a round robin mode
   - the thread block remains on the SMP until all its threads have finished
3. The GPU runtime system can execute blocks in any order
   - blocks do not communicate
   - kernel code can't rely on sequential execution across blocks



Thread indexing

Grids and Blocks are organized as “geometrical” structures in arrays/matrices/tensors of threads:

- ➔ a grid is generally a 3D array of blocks
- ➔ a block is generally a 3D array of threads

Both grid and blocks can be of lower dimensions, if preferred

The choice is up to the programmer to optimize the execution of threads in the case of hierarchical data structures (e.g. arrays/matrices/tensors)

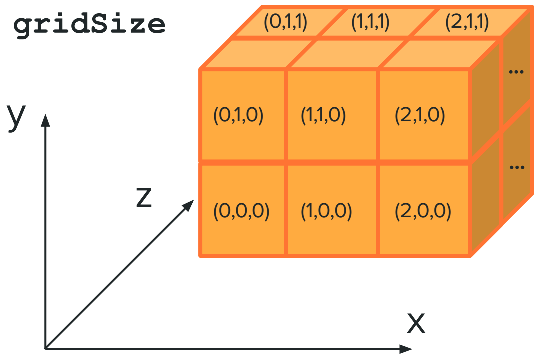
CUDA C offers a built-in **`dim3`** data type for setting the size grids and blocks, and **`uint3`** built-in variables to describe the current index of thread/block

Variable	Type	Description
`gridDim`	`dim3`	Dimensions of grid
`blockIdx`	`uint3`	Index of current block within grid
`blockDim`	`dim3`	Dimensions of block
`threadIdx`	`uint3`	Index of current thread within block

Thread indexing

```
// Set up grid and block dimensions
dim3 blockSize(4, 4);     // 4x4   threads in a block
dim3 gridSize(3, 2, 2);   // 3x2x2 blocks in a grid

// Launch kernel with custom dimensions
my_kernel<<<gridSize, blockSize>>>(input_data,
                                   output_data);
```

12 blocks in the grid



Thread indexing

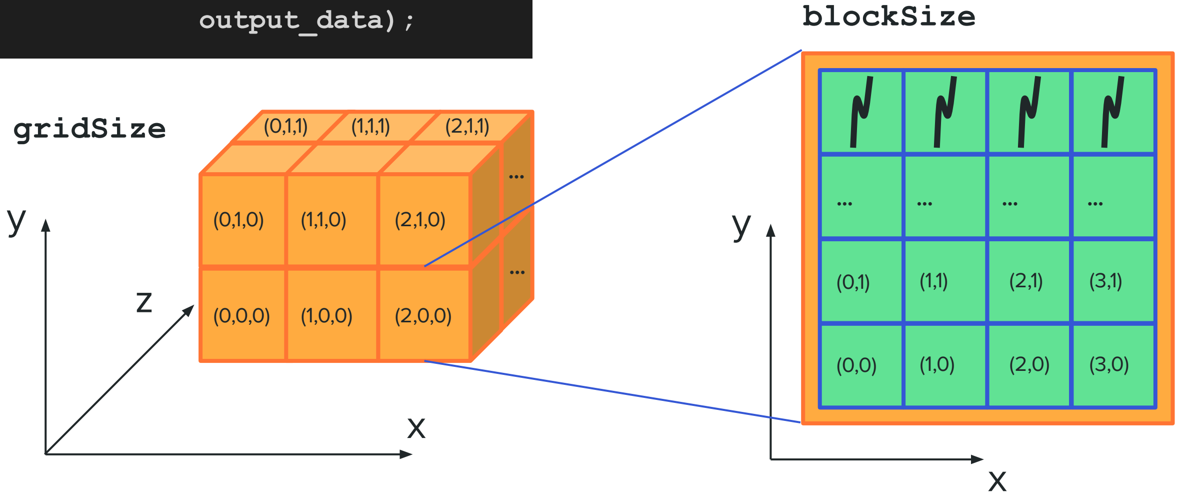
```
// Set up grid and block dimensions
dim3 blockSize(4, 4);    // 4x4   threads in a block
dim3 gridSize(3, 2, 2);  // 3x2x2 blocks in a grid

// Launch kernel with custom dimensions
my_kernel<<<gridSize, blockSize>>>(input_data,
                                   output_data);
```

16 threads per block

12 blocks in the grid

16x12 =

192 total threads in grid



Thread indexing

```
__global__ void my_kernel(const int* input_data,
                          const int* output_data){
   int id_x = blockIdx.x * blockDim.x + threadIdx.x;
   int id_y = blockIdx.y * blockDim.y + threadIdx.y;
   int id_z = blockIdx.z * blockDim.z + threadIdx.z;
   ...
}
```

Remember, every thread is identical

➡ a call of the same kernel (function)

➡ but operating on different data inputs

Map each CUDA thread onto a unique index to access data

Indexing is extremely helpful when dealing with multidimensional data, for example, 2D matrices, tensors, etc

Thread indexing

```
__global__ void my_kernel(const int* input_data,
                          const int* output_data){
   int id_x = blockIdx.x * blockDim.x + threadIdx.x;
   int id_y = blockIdx.y * blockDim.y + threadIdx.y;
   int id_z = blockIdx.z * blockDim.z + threadIdx.z;
   ...
}
```

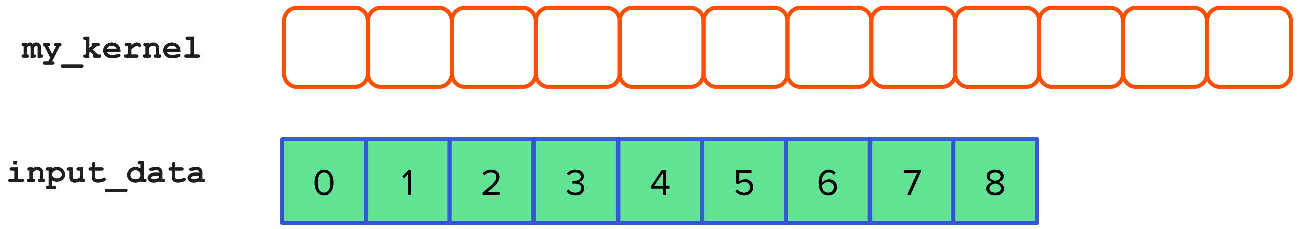
Remember, every thread is identical
➔ a call of the same kernel (function)
➔ but operating on different data inputs

Map each CUDA thread onto a unique index to access data

Indexing is extremely helpful when dealing with multidimensional data, for example, 2D matrices, tensors, etc

BUT we must prevent out-of-border access to data if data is not an exact multiple of thread block size

```
my_kernel<<<4, 3>>>(input_data,output_data);
```



Thread indexing

```
__global__ void my_kernel(const int* input_data,
                          const int* output_data){
   int id_x = blockIdx.x * blockDim.x + threadIdx.x;
   int id_y = blockIdx.y * blockDim.y + threadIdx.y;
   int id_z = blockIdx.z * blockDim.z + threadIdx.z;
   ...
}
```

Remember, every thread is identical

➜ a call of the same kernel (function)

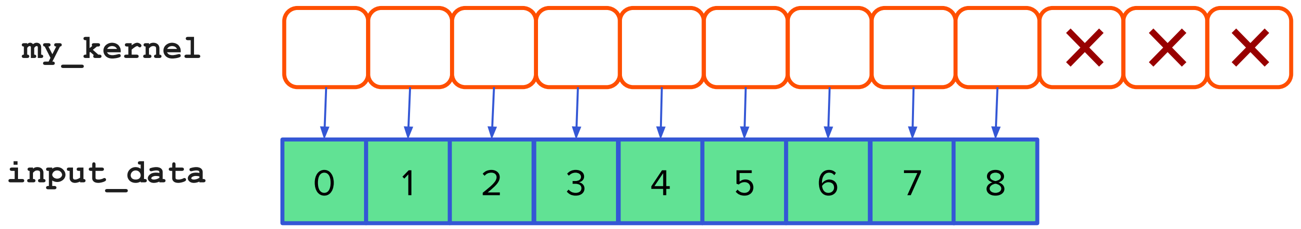
➜ but operating on different data inputs

Map each CUDA thread onto a unique index to access data

Indexing is extremely helpful when dealing with multidimensional data, for example, 2D matrices, tensors, etc

BUT we must prevent out-of-border access to data if data is not an exact multiple of thread block size

```
my_kernel<<<4, 3>>>(input_data,output_data);
```



Vector sum with threads

Using `vector_sum_threads.cu`

```
// Define a CUDA kernel that adds two vectors element-wise
__global__ void sum_kernel(const int *x, const int *y, int *res){
   // Calculate the thread ID
   int tid = threadIdx.x;
   printf("Thread number %d\n",tid);

   // Each thread computes one element of the result vector
   res[tid] = x[tid] + y[tid];
}
```

Vector sum with threads

```
#include <stdio.h>
#include <vector>      // For std::vector
#include <algorithm>   // For std::generate
#include <stdlib.h>    // For srand and rand
#include <time.h>      // For time

#define N 4  // Define the size of the vector

// Function to generate a random number between 0 and 99
int random_number() {
   return (std::rand() % 100);
}

int main(int argc, char **argv){

   // Seed the random number generator with the current time
   srand(time(NULL));  // Ensure that rand() produces different sequences each run

   // Local vectors hosted in memory, each with N elements
   std::vector<int> a(N), b(N), c(N);

   // Initialize vectors 'a' and 'b' with random numbers
   std::generate(a.begin(), a.end(), random_number);  // Fill vector 'a' with random numbers
   std::generate(b.begin(), b.end(), random_number);  // Fill vector 'b' with random numbers

   [...]
```

Vector sum with threads

```
[...]

// Pointers to device (GPU) memory for the vectors
int *dev_a, *dev_b, *dev_c;

// Determine the size of the memory required for each vector
int size = N * sizeof(int);

// Allocate space on the GPU for the copies of the vectors
cudaMalloc((void **)&dev_a, size);
cudaMalloc((void **)&dev_b, size);
cudaMalloc((void **)&dev_c, size);

// Print the result of vector addition on the CPU
printf("CPU result:\n");
for (int i = 0; i < N; i++) {
    printf("[el. %d] %d + %d = %d (on CPU) \n",i,a[i],b[i],a[i]+b[i]);
}

// Copy the input vectors from the CPU to the GPU
cudaMemcpy(dev_a, a.data(), size, cudaMemcpyHostToDevice);
cudaMemcpy(dev_b, b.data(), size, cudaMemcpyHostToDevice);

// Launch the sum kernel on the GPU with 1 block and N threads
sum_kernel<<<1, N>>>(dev_a, dev_b, dev_c);

[...]
```

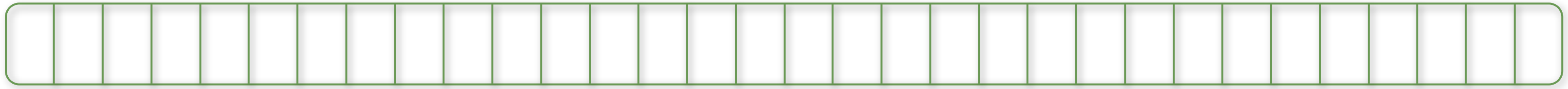
Vector sum with threads

```
    [...]

    // Copy the result vector from the GPU back to the CPU
    cudaMemcpy(c.data(), dev_c, size, cudaMemcpyDeviceToHost);

    // Print the result of the vector addition performed on the GPU
    printf("GPU result:\n");
    for (int i = 0; i < N; i++) {
        printf("[el. %d] %d + %d = %d (on GPU) \n",i,a[i],b[i],c[i]);
    }

    // Cleanup by freeing the allocated GPU memory
    cudaFree(dev_a);
    cudaFree(dev_b);
    cudaFree(dev_c);

    // Return 0 to indicate successful execution
    return 0;
}
```

Vector sum with blocks and threads

To assign kernel execution on an array structure over multiple blocks and threads, the indexing is important

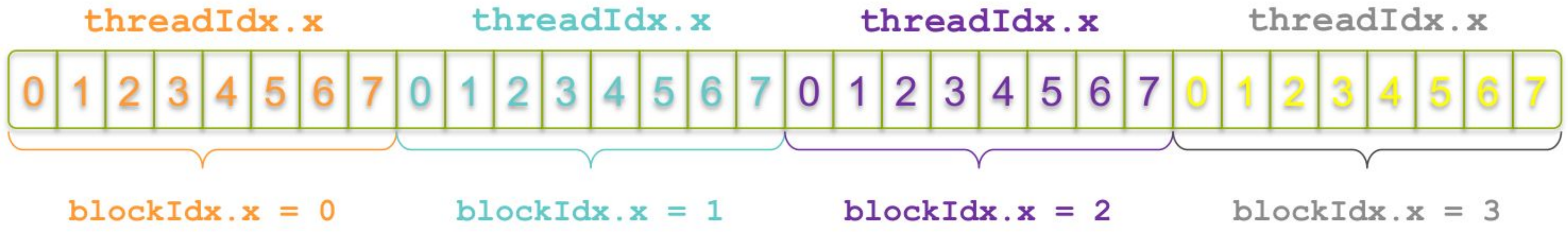
Consider indexing an array with one element per thread (8 threads/block in the figure):



Vector sum with blocks and threads

To assign kernel execution on an array structure over multiple blocks and threads, the indexing is important

Consider indexing an array with one element per thread (8 threads/block in the figure):

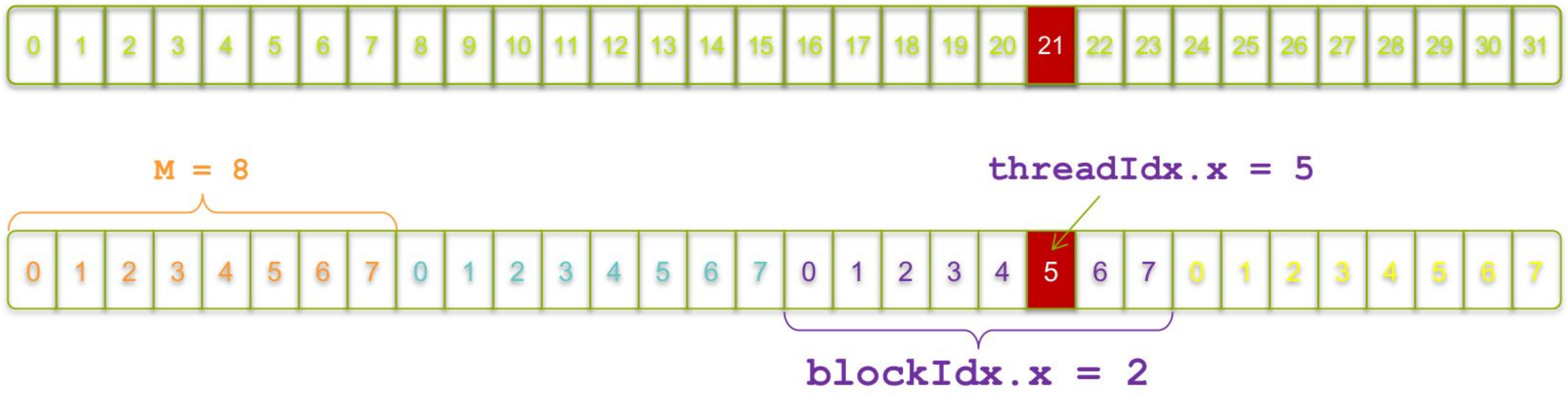


A unique index for each thread is given by:

```
int index = threadIdx.x + blockIdx.x * blockDim.x;
```

Vector sum with blocks and threads

This way each element of the vector is going to be indexed uniquely and processed by one given block-thread combination



Vector sum with blocks and threads

Using `vector_sum_blocks_and_threads.cu`

```
[...]

#define N 1024                    // Define the size of the vector
#define THREADS_PER_BLOCK 128     // Define the number of threads in a block

// Define a CUDA kernel that adds two vectors element-wise
__global__ void sum_kernel(const int *x, const int *y, int *res){
   // Calculate the thread ID of the overall grid
   auto idx = threadIdx.x + blockIdx.x * blockDim.x;

   printf("Block - Thread number %d-%d\n",blockIdx.x,threadIdx.x);

   // Each thread computes one element of the result vector
   res[idx] = x[idx] + y[idx];
}

[...]
```

Vector sum with blocks and threads

Using `vector_sum_blocks_and_threads.cu`

```
[...]

   // Compute the number of blocks and threads per block
   int N_b   = N/THREADS_PER_BLOCK;
   int N_tpb = THREADS_PER_BLOCK;

   // Launch the sum kernel on the GPU with multiple blocks and threads
   sum_kernel<<<N_b, N_tpb>>>(dev_a, dev_b, dev_c);

[...]
```

Try printing out and inspecting the Block-Thread number during the GPU execution...

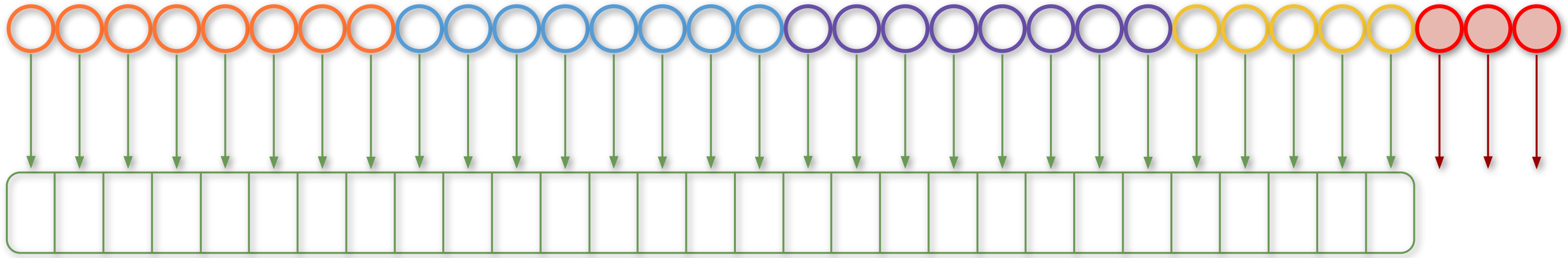
Notice any pattern in the numbering?

Arbitrary vector sum with blocks and threads

What if the size of the vector is not neatly divisible in blocks and threads?

Naively, this may not seem an issue, but we must remember that:

- scheduling of the kernels happens by assigning thread blocks to SMPs
  ➡ 1 single block with 1000s of threads won't be efficient at all in the case of devices with several SMPs
- all kernels are identical, but must act on valid data locations
  ➡ a kernel attempting to access a non-existent data location is going to fail and crash the application



Arbitrary vector sum with blocks and threads

Using `vector_sum_blocks_and_threads_arbitrary.cu`

```
[...]

#define N 1021                        // Define the size of the vector
#define THREADS_PER_BLOCK 128         // Define the number of threads in a block

// Define a CUDA kernel that adds two vectors element-wise
__global__ void sum_kernel(const int *x, const int *y, int *res, int max_index){
   // Calculate the thread ID of the overall grid
   auto idx = threadIdx.x + blockIdx.x * blockDim.x;

   printf("Block - Thread number %d-%d\n",blockIdx.x,threadIdx.x);

   // Each thread computes one element of the result vector
   // Protect the thread against accessing non-existing memory locations
   if (idx < max_index){
       res[idx] = x[idx] + y[idx];
   }
}

[...]
```

Arbitrary vector sum with blocks and threads

```
[...]

   // Compute the number of blocks and threads per block
   int N_b    = ceil(N/THREADS_PER_BLOCK);
   int N_tpb = THREADS_PER_BLOCK;

   // Launch the sum kernel on the GPU with multiple blocks and threads
   // and protect against accessing non-existing data locations
   sum_kernel<<<N_b, N_tpb>>>(dev_a, dev_b, dev_c, N);

[...]
```

We must have enough threads to “cover” the entire vector, although some of them won’t actually compute anything

Thus we use the ceiling function to N/THREADS_PER_BLOCK, e.g.

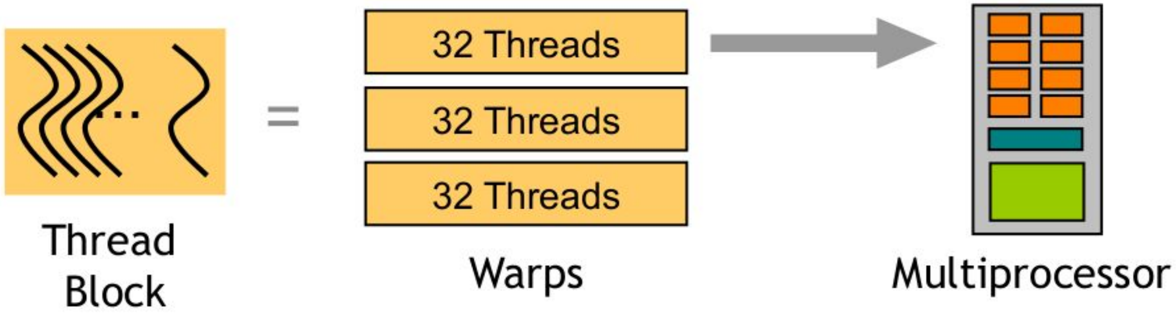
```
N = 1021
THREADS_PER_BLOCK = 128
```

⇒ `N/THREADS_PER_BLOCK = 7.976… → 8 blocks`

What about *warps* then?

We anticipated that a SMP actually schedules threads in groups of a given number (normally 32), called **warps**

Threads of each block are in fact partitioned into warps of consecutive threads



A warp execute **one common set of instruction at a time (SIMD)**, mapping onto the multiple core ALU of the SMP units

- ➔ fully efficient when all threads have the same execution path

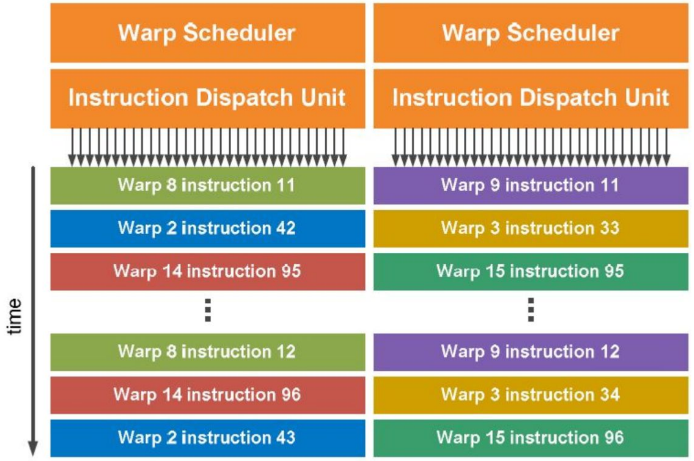
What about *warps* then?

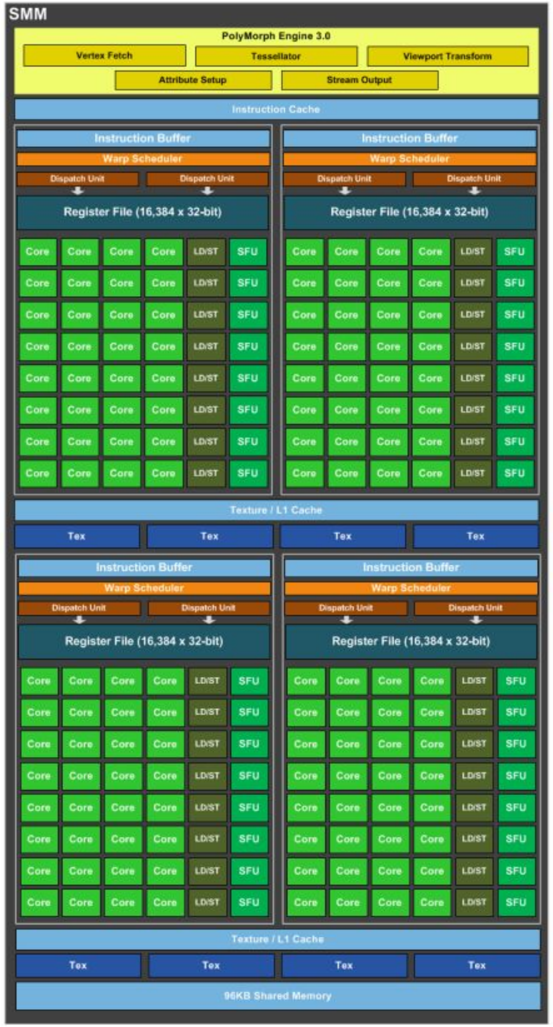
A SMP can host multiple warp schedulers, allowing for concurrent execution of multiple warps at the same time

- ➜ Maxwell architecture (e.g. Jetson) 1 SMP hosts 4 warp schedulers

The scheduler select for execution an eligible warp from one of the residing blocks in each SMP

- ➜ next instruction was fetched and all its arguments are ready
- ➜ resources (ALUs) are available (e.g. fp32 cores)



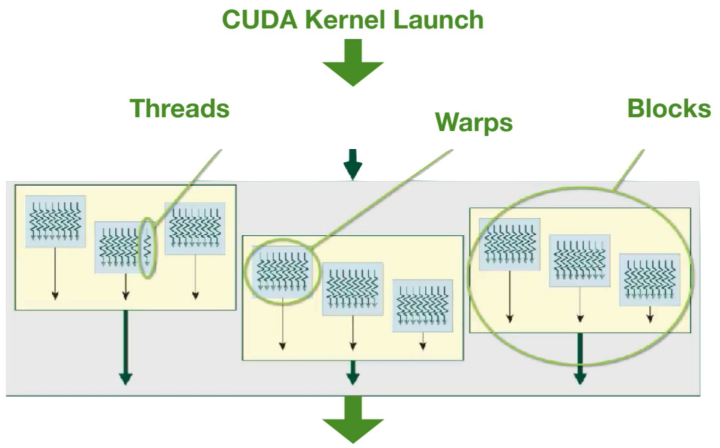


Mapping SW-HW parallelism

Re-interpreting SIMT: Single Instruction Multiple Threads

- All (possibly thousands of threads) are concurrent, as in “standard” computing
- Within a warp though, all threads are synchronized at the SIMD level
- Warps however don’t have to be running simultaneously or be synchronized

CUDA	Mapping to NVIDIA GPU	Meaning
Kernel		Function
Grid	Whole GPU	“Non cooperating” threads
Block	SMP	“Cooperating” threads
Thread	SP	Thread
Warp	32-SP	SIMD

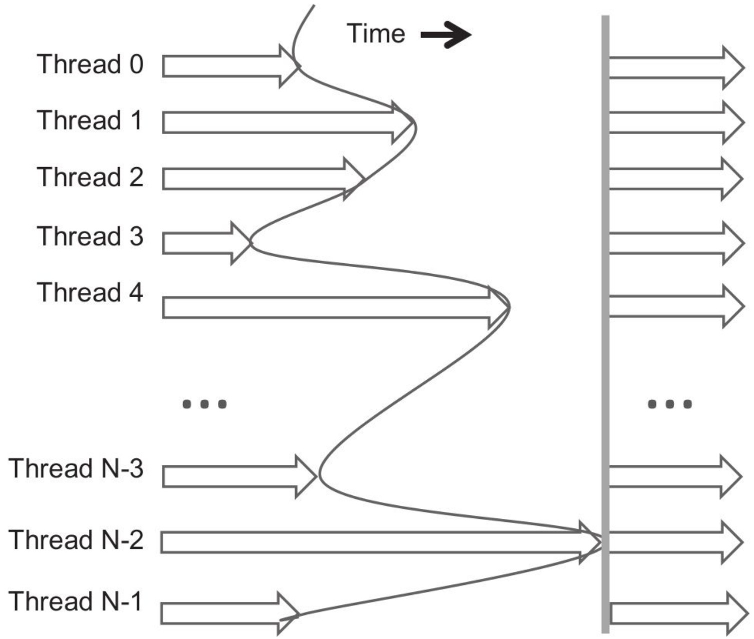


Thread synchronization

In CUDA, all threads of a grid run the same code, but we can take branches by using `if` conditions, for instance based on `threadIdx.x` and `blockIdx.x`

REMEMBER ➔ GPU SPs are not as “smart” as CPU Cores... No OoO execution or Branch Prediction

Branches can induce different threads in the same block to have different execution times



Thread synchronization

Threads **in the same block** can be synchronized using, in the kernel definition a specific call:

```
__global__ void a_kernel(void){

   // Kernel code
   ...

   // Wait here for all threads to synch
   __syncthreads()

   // Continue on with kernel code
   ...
}
```

This will be extremely useful when needing to access/update a shared memory location for all threads in a block

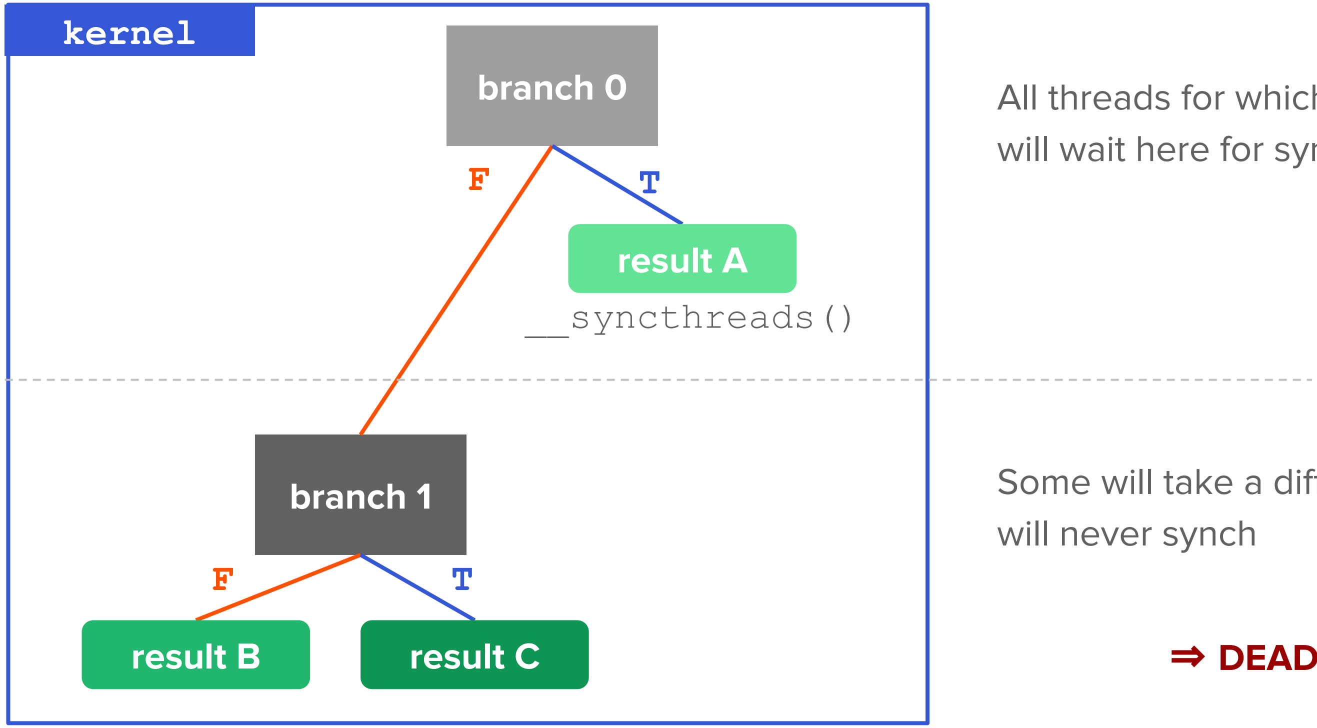
➜ All threads of a block will halt there until the last thread (of the same block) has reached this point

➜ On the other hand, threads in different blocks are independent and cannot be synchronized!

Thread synchronization

A sidenote...

- What happens if threads in the same block can take different branches containing different `__syncthreads()` statements?



All threads for which branch 0 is T will wait here for synchronization

Some will take a different route, and will never synch

⇒ DEADLOCK

Transparent scalability and thread scheduling

Let's review one important concept:

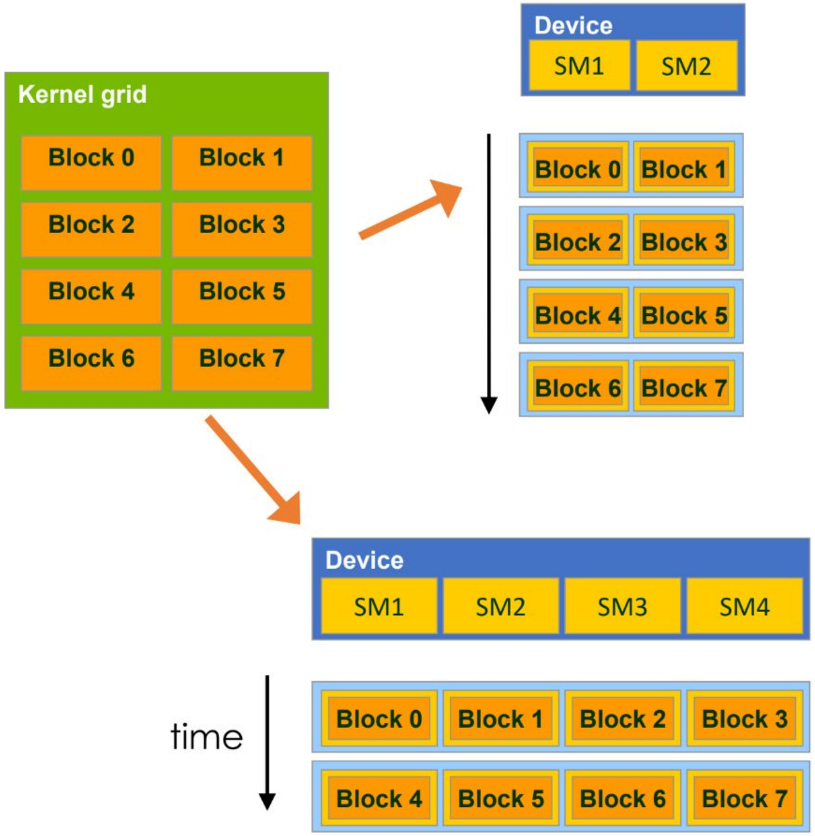
Each blocks is assigned to SMPs and remains active on the SMP until all its threads have finished

Overall, the **GPU can execute blocks in any order**

This is often referred to as **Transparent scalability**

➜ No by-hand assignment of blocks to SMPs
➜ Every block execution is scheduled independently from the others
➜ Different devices will schedule blocks differently

However, this also comes with the aforementioned price of complete independence across blocks



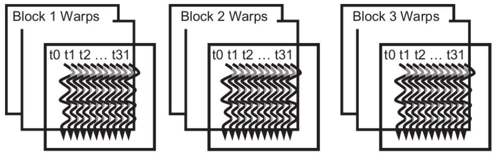
Warp scheduling allows to hide latency

It's important to keep the SMPs busy with lots of warps! … but why?

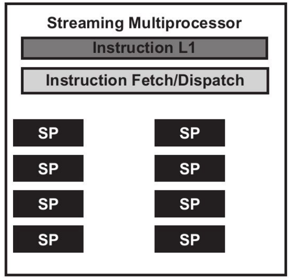
Warps are the actual "unit of scheduling"
➡ All threads within a warp proceed as SIMD ⇒ All threads in a warp will have the same execution timing.

Let's imagine that 3 blocks of 256 threads are scheduled to run on the same SMP
⇒ 256 threads / 32 threads per warp = 8 warps per block
⇒ 3 blocks per SMP * 8 warps per block = 24 warps per SMP

For simplicity, let's also assume the SMPs have hardware resources (SPs) to execute 1 warp



In this example there are fewer SPs (32) than threads (24*32 = 768) assigned to the SMP



Only 1 warp can proceed at the same time, so why bother?

Because of long-latency operations such as memory access

Warp scheduling allows to hide latency

When an instruction in a warp is executed by all the threads, the latency of it kicks in
➜ if it's a computation it's usually a low-latency (~1-10s of clock cycles)
➜ if it's a data load requiring to access the global GPU memory it's a very-high-latency one (~100s of clock cycles)

Warp schedulers can switch between different warps without introducing idle time
⇒ zero-overhead thread scheduling

The warp scheduler decides to switch to a different warp while a long-latency instruction is completed, thus resulting in a shorted overall execution time

The latency is still all there, but it's "hidden"

This is possibly only if there are enough warps on the SMP to be able to switch among them based on which one has a "ready-to-perform" next instruction

What we got on optimization so far

Latency is our enemy and memory bandwidth is one of the main culprit

Warp scheduling seems unnecessary at first, but it’s actually where GPU computing gains substantially

To optimize the execution we should be careful to:

1. Avoid unnecessary branching and the related synchronization
2. Have the kernel to perform few memory access and lots of computation
3. Overcommit the SMPs with warps, to let the warp schedulers hide latency

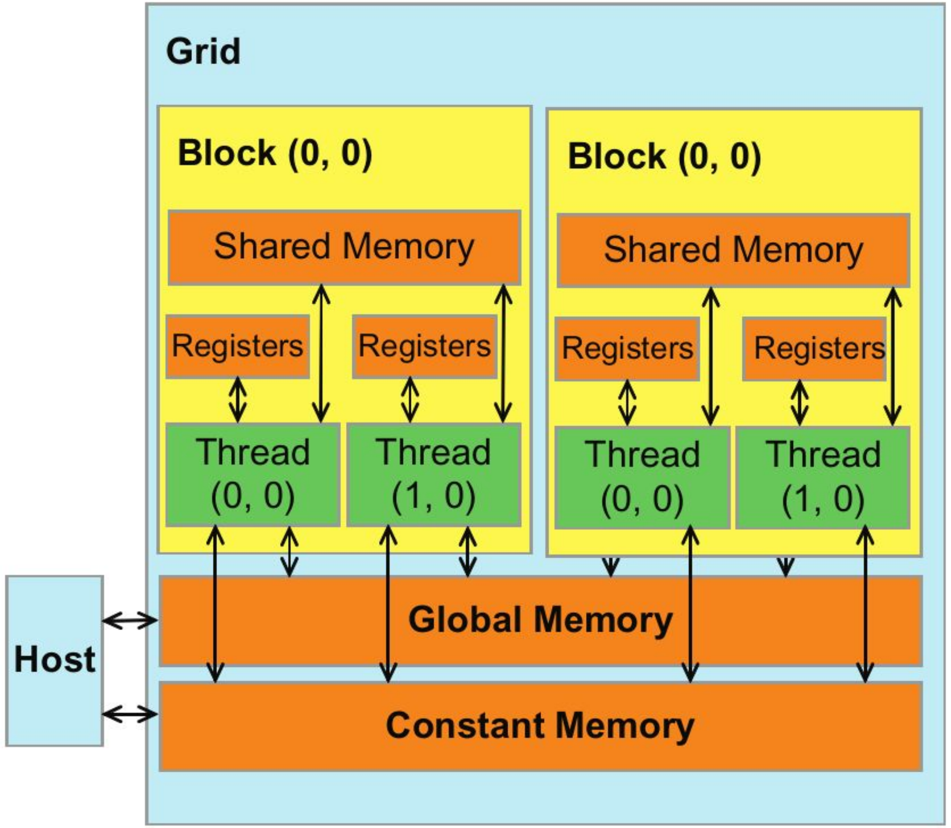
CUDA device memory model

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



Memory and performance

Memory accesses are most often the performance limiting factor in GPU applications

Let’s work it out with an example:

```
for (int k = 0; k < width; ++k) {
    p_tmp_val += dev_M[row * width + k] * dev_N[k * width + col];
}
```

Memory and performance

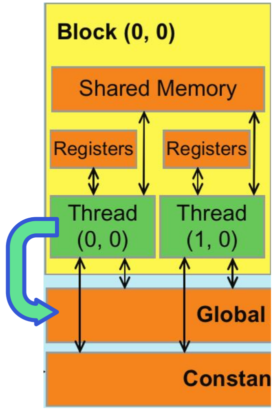
Memory accesses are most often the performance limiting factor in GPU applications

Let’s work it out with an example:

```
for (int k = 0; k < width; ++k) {
    p_tmp_val += dev_M[row * width + k] * dev_N[k * width + col];
}
```

floating point operations

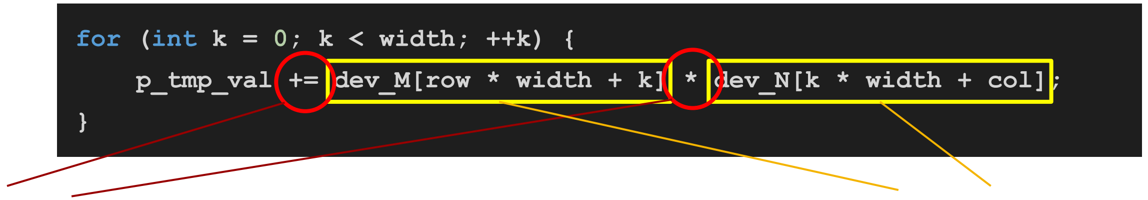
load from global device memory



Memory and performance

Memory accesses are most often the performance limiting factor in GPU applications

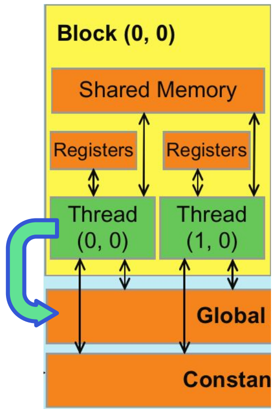
Let's work it out with an example:



```
for (int k = 0; k < width; ++k) {
    p_tmp_val += dev_M[row * width + k] * dev_N[k * width + col];
}
```

floating point operations

load from global device memory



In this loop there are 2 global memory accesses, 1 FP multiplication, and 1 FP addition

The ratio of **C**ompute operations to **G**lobal **M**emory **A**ccesses (CGMA) is 2/2 = 1.0

Memory and performance

Let’s assume we are using a GPU whose performances are:

- peak Floating Point performance 1500 GFLOPS
- memory bandwidth 200 GB/s

Memory and performance

Let’s assume we are using a GPU whose performances are:

- peak Floating Point performance 1500 GFLOPS
- memory bandwidth 200 GB/s

Given our CGMA of 1.0, and given the single-precision FP size of 4 Bytes, the maximum number of Floating point OPerations Per Second we can achieve during execution of this thread is

max FLOPS = 200 GB/s / 4 B * 1.0 = 50 GFLOP

Memory and performance

Let’s assume we are using a GPU whose performances are:

- peak Floating Point performance 1500 GFLOPS
- memory bandwidth 200 GB/s

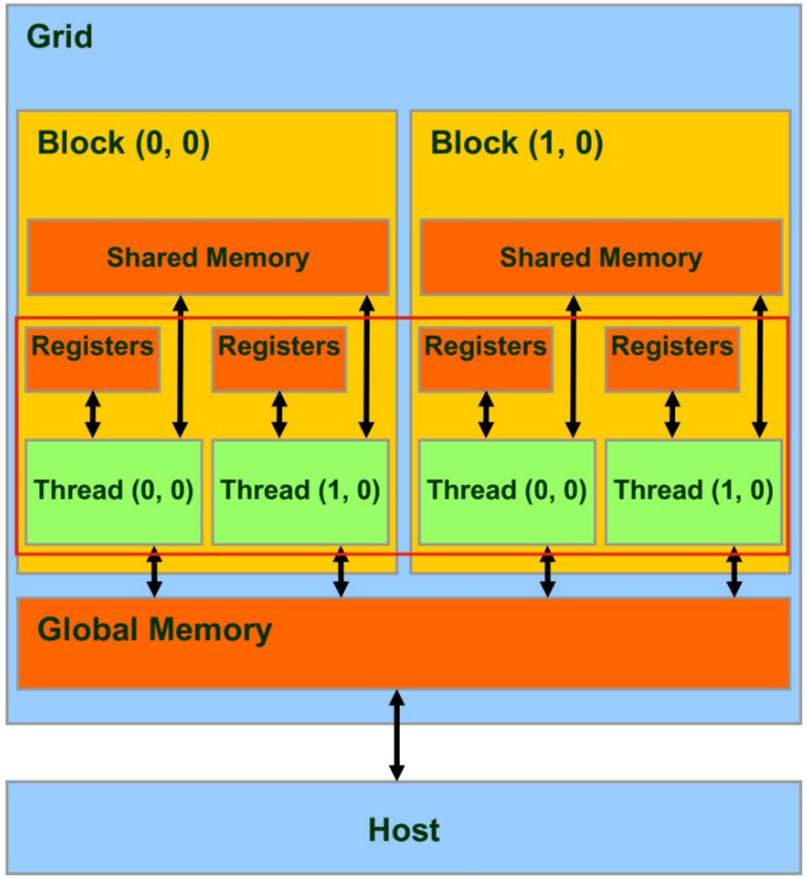
Given our CGMA of 1.0, and given the single-precision FP size of 4 Bytes, the maximum number of Floating point OPerations Per Second we can achieve during execution of this thread is

max FLOPS = 200 GB/s / 4 B * 1.0 = 50 GFLOP

➡ this is a direct result of the limitation of the global memory bandwidth

To reach peak performance, we would need a CGMA of 30 (30 FP operations per Global Memory Access)

Memory hierarchy - Thread



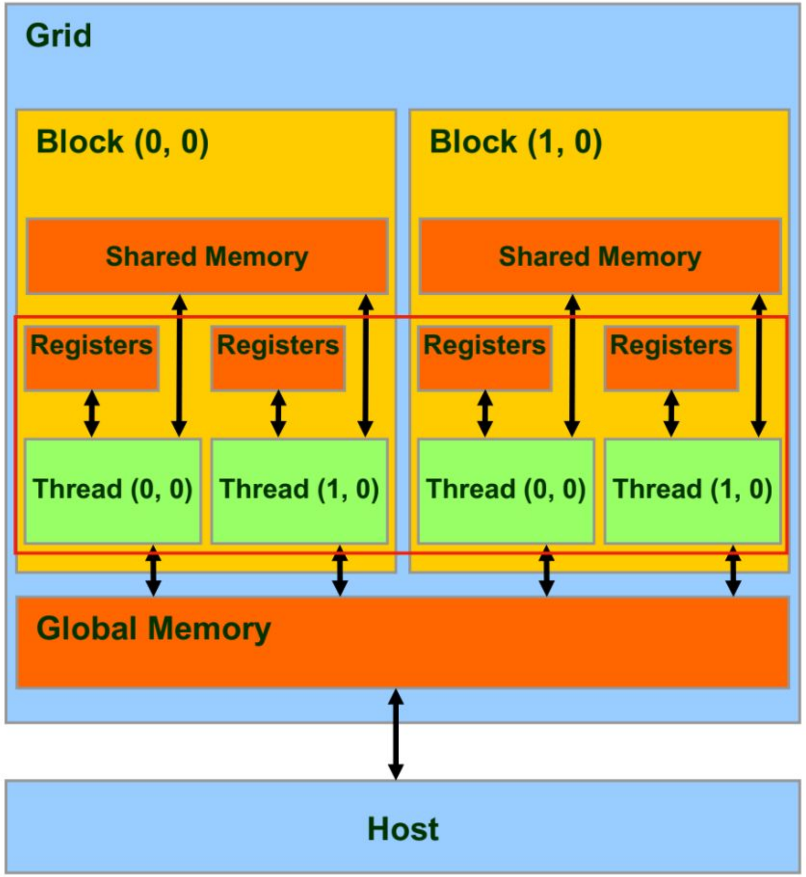
Per Thread

- Scope: Private to each thread
- Lifetime: Thread

Registers

- The fastest memory on the GPU (~zero latency)
- Scalar variables declared in a kernel use registers
- Typical size: ~48 KB

Memory hierarchy - Thread



Per Thread

- Scope: Private to each thread
- Lifetime: Thread

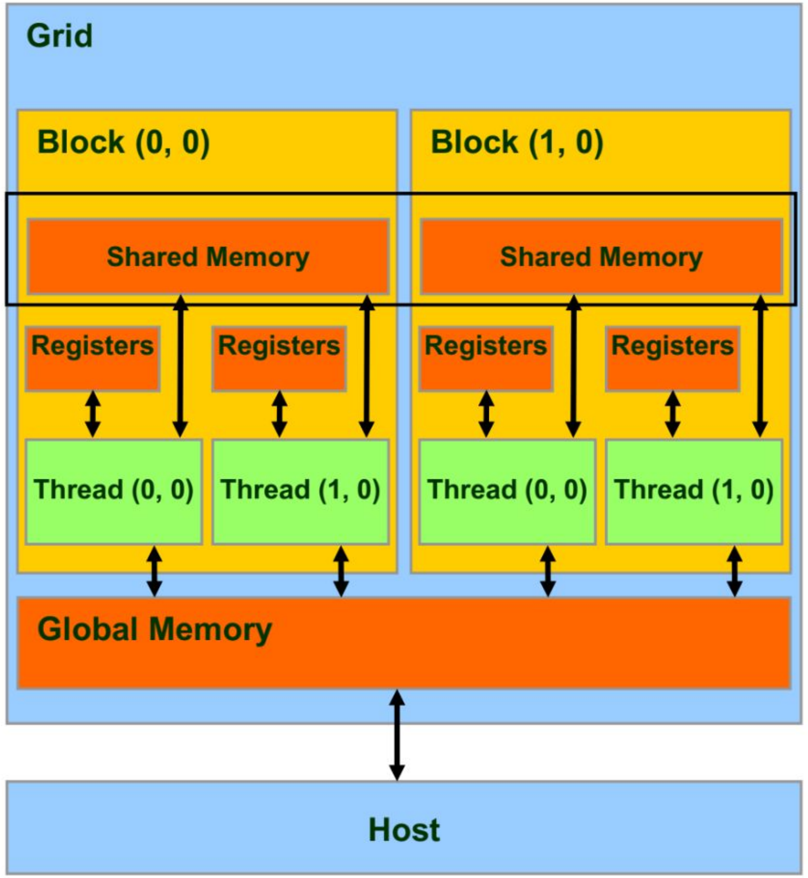
Registers

- The fastest memory on the GPU (~zero latency)
- Scalar variables declared in a kernel use registers
- Typical size: ~48 KB

Local memory

- Part of main memory of the GPU (same as the global memory)
- Generally slow, but can be partially cached
- Used automatically by threads when run out of registers

Memory hierarchy - Block



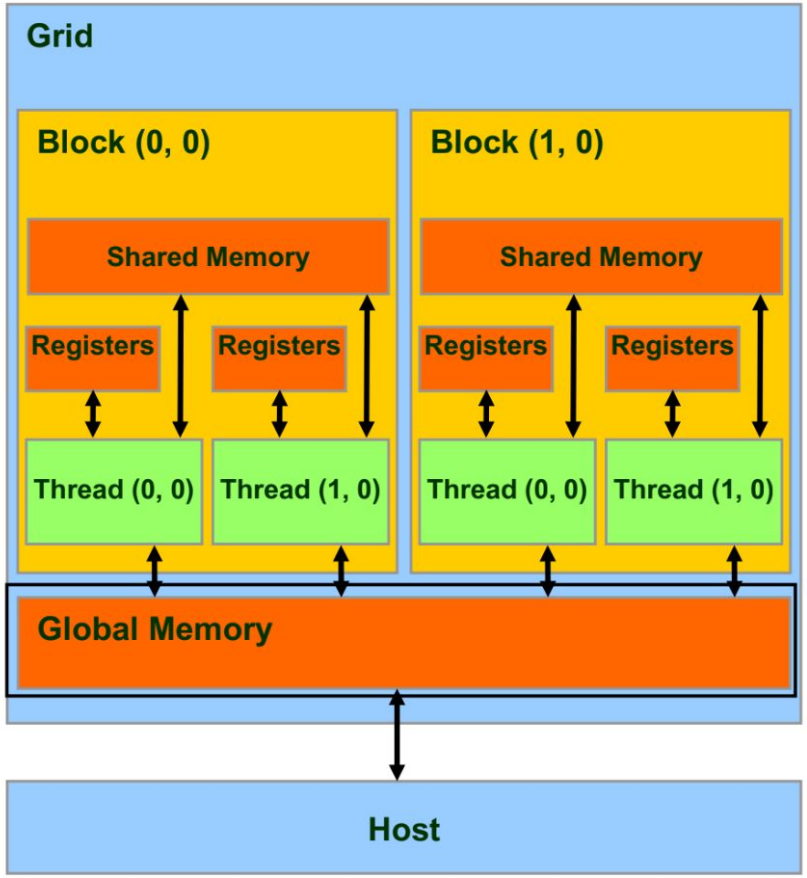
Per Thread Block

- Scope: Every thread in the block has access
- Lifetime: Block

Shared memory

- Small but quite fast memory mounted on each SMP
- Low latency: ~2-10 clock cycles
- Typical size: ~48 KB
- Read/write access for threads of blocks residing on the same SM
- It is used to enable fast communication between threads in a block

Memory hierarchy - Grid



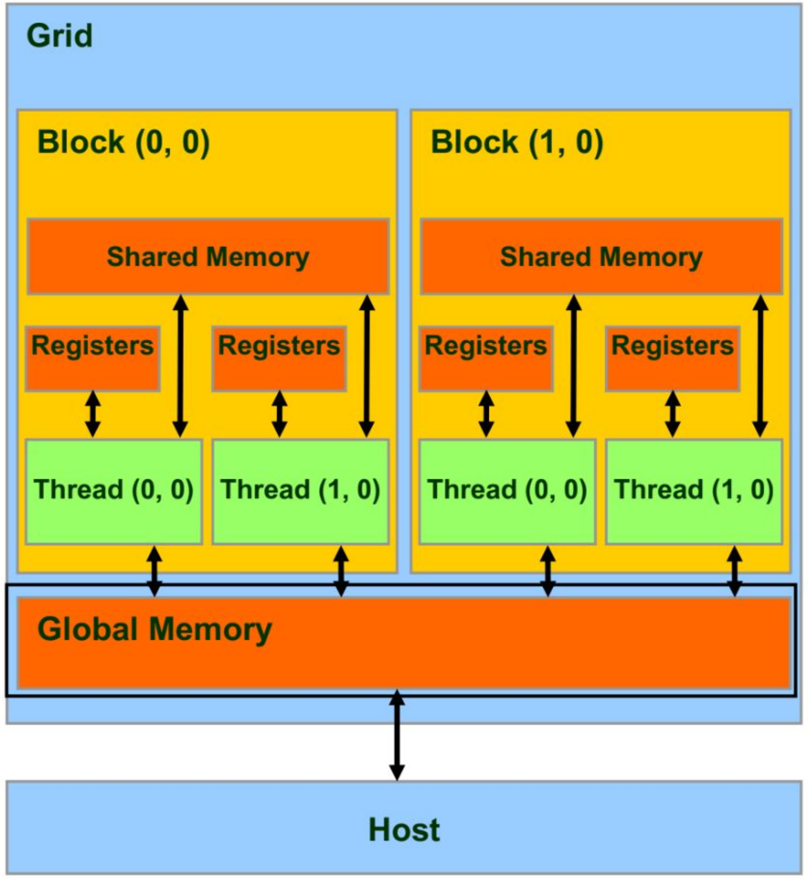
Grids (all threads)

- Scope: Every thread in all grids have access
- Lifetime: Entire program in host code - `main()`

Global memory

- On the on-board device memory
- Large memory (~10s of GB)
- Bandwidth of ~100 GB/s
- Very high latency: 400-800 clock cycles
- All running threads can read and write

Memory hierarchy - Grid



Grids (all threads)

- Scope: Every thread in all grids have access
- Lifetime: Entire program in host code - `main()`

Global memory

- On the on-board device memory
- Large memory (~10s of GB)
- Bandwidth of ~100 GB/s
- Very high latency: 400-800 clock cycles
- All running threads can read and write

Constant Memory

- Part of GPU's main memory
- Supports ~64k of memory
- Read-only access by all threads

Variable definition and memory usage

Variable Declaration	Memory	Scope	Lifetime
`int var;`	Register	Thread	Kernel
`int var[len];`	Local	Thread	Kernel
`__device__ __shared__ int SharedVar;`	Shared	Block	Kernel
`__device__ int GlobalVar;`	Global	Grid	Application
`__device__ __constant__ int ConstVar;`	Constant	Grid	Application

All automatic **scalar variables declared within a kernel** are assigned to **registers** (unless full)

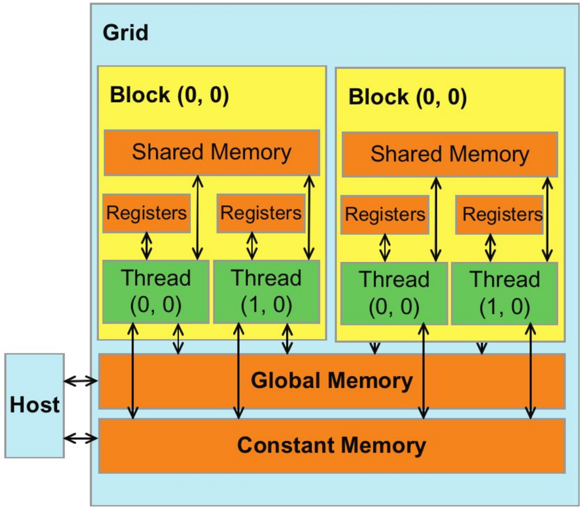
All automatic array variables are **not** stored in registers, but are stored in local memory

Understanding and exploiting the memory model

Moving data in and out of global memory is expensive

Limit calls to global data in favor of:

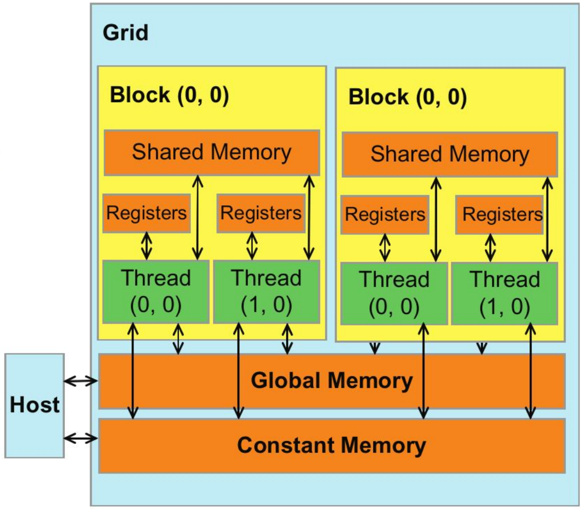
- If small amount of data that can be re-computed on the fly
  ⇒ re-instantiate variables as local per-thread entities
  (registers ➔ higher CGMA)
- If larger amount, required by several threads
  ⇒ partition data and allocate it as shared memory and
  reuse it across threads
  (memory tiling)



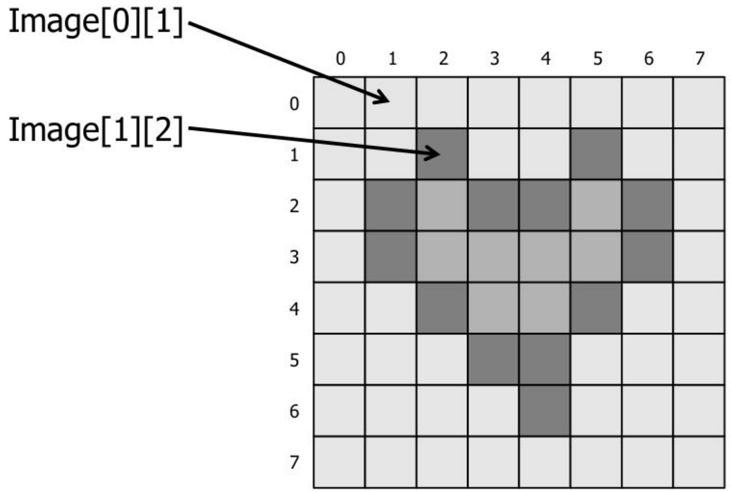
Understanding and exploiting the memory model

However:

- Be aware not to exceed the capacity of registers/shared memories
  ⇒ risk of spilling data to global memory
- When using shared memory, be careful to include a synchronization stage to avoid data race an corruptions
  ⇒ `__synchthreads()`



Scalar Matrix multiplication



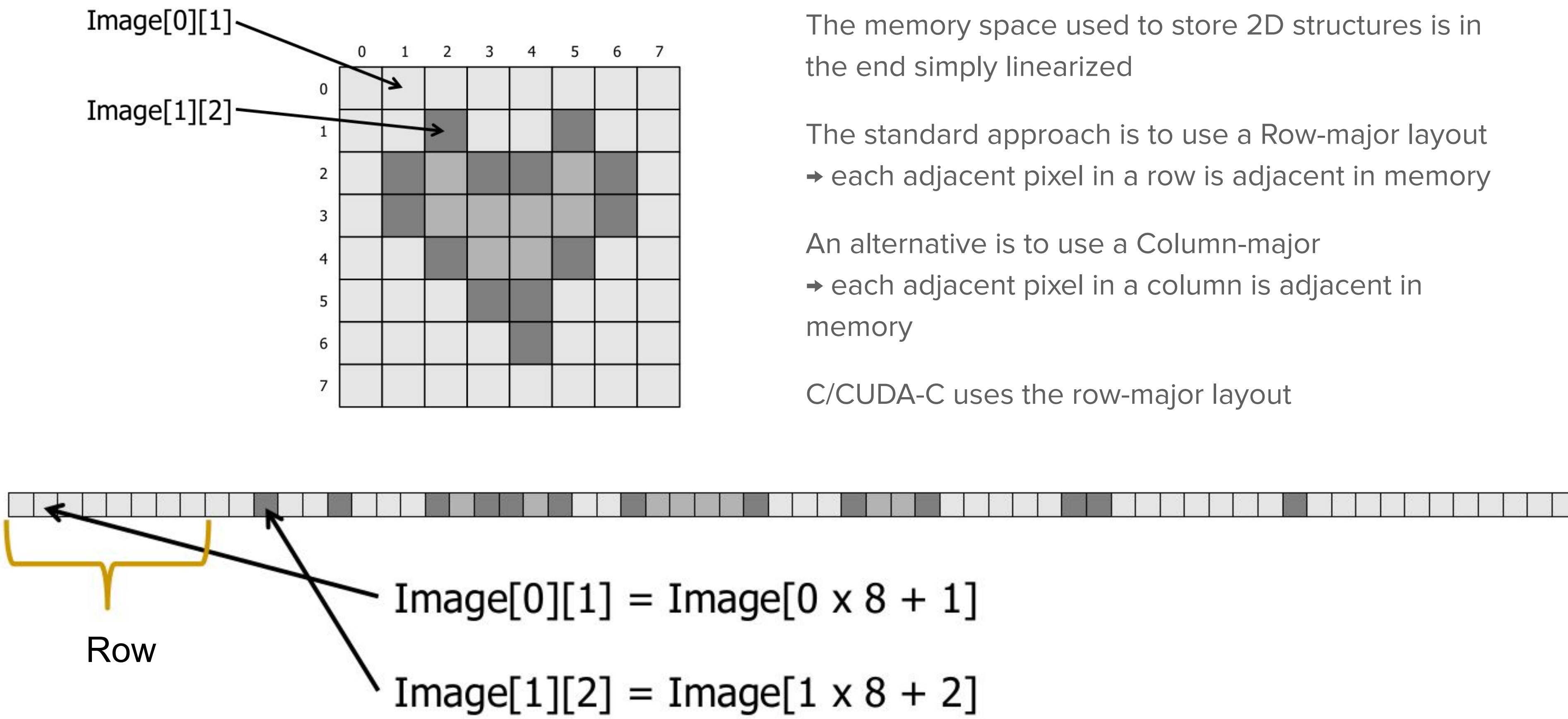
The memory space used to store 2D structures is in the end simply linearized

The standard approach is to use a Row-major layout
➡ each adjacent pixel in a row is adjacent in memory

An alternative is to use a Column-major
➡ each adjacent pixel in a column is adjacent in memory

C/CUDA-C uses the row-major layout

Scalar Matrix multiplication



Image[0][1]
Image[1][2]
0 1 2 3 4 5 6 7
0 1 2 3 4 5 6 7
Row
Image[0][1] = Image[0 x 8 + 1]
Image[1][2] = Image[1 x 8 + 2]

The memory space used to store 2D structures is in the end simply linearized

The standard approach is to use a Row-major layout
➡ each adjacent pixel in a row is adjacent in memory

An alternative is to use a Column-major
➡ each adjacent pixel in a column is adjacent in memory

C/CUDA-C uses the row-major layout

Scalar Matrix multiplication

Using `matrix_elementwise_1d.cu`

```
[...]

#define ALPHA 0.25                // Define the scalar
#define ROWS 1080                 // Define the matrix row number
#define COLS 1920                 // Define the matrix column number
#define THREADS_PER_BLOCK 256     // Define the number of threads in a block

// CUDA kernel to perform elementwise multiplication
__global__ void matrix_elementwise(const float* M, float* P, const float alpha, const int rows, const int cols)
{
   // Calculate the thread ID of the overall grid
   int idx = blockIdx.x * blockDim.x + threadIdx.x;

   // Each thread computes one element of the result matrix
   if (idx < rows * cols) {
       P[idx] = alpha*M[idx];
   }
}

[...]
```

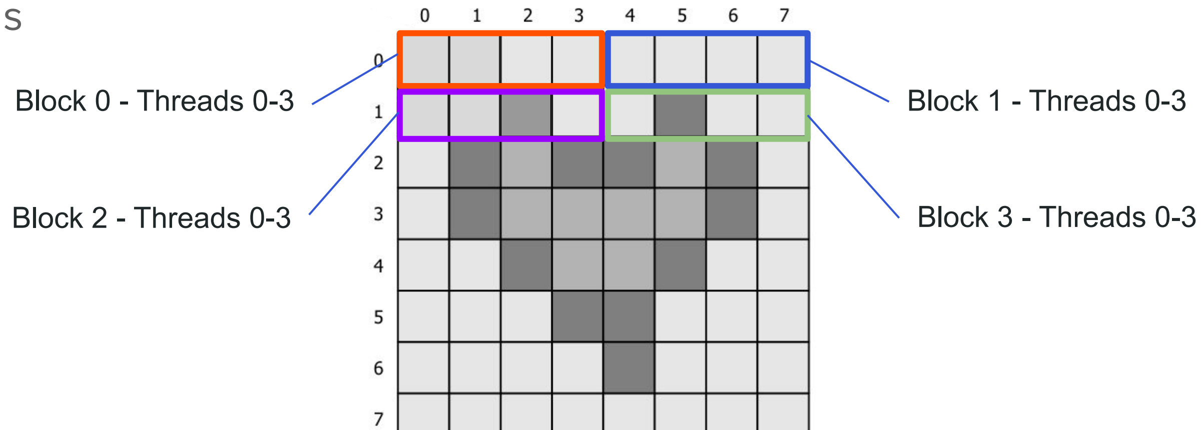
Scalar Matrix multiplication

```
[...]

   // Compute the number of blocks and threads per block
   // Blocks are 1-dimensional
   int N_b   = ceil(ROWS*COLS/THREADS_PER_BLOCK);
   int N_tpb = THREADS_PER_BLOCK;

   // Launch the CUDA kernel
   matrix_elementwise<<<N_b, N_tpb>>>(d_M, d_P, ALPHA, ROWS, COLS);

[...]
```

Scalar Matrix multiplication

```
[...]

    // Compute the number of blocks and threads per block
    // Blocks are 1-dimensional
    int N_b   = ceil(ROWS*COLS/THREADS_PER_BLOCK);
    int N_tpb = THREADS_PER_BLOCK;

    // Launch the CUDA kernel
    matrix_elementwise<<<N_b, N_tpb>>>(d_M, d_P, ALPHA, ROWS, COLS);

[...]
```

This way we are treating the matrix as 1D array, and covering it with a grid of 1D blocks and threads



Scalar Matrix multiplication - 2D Grid

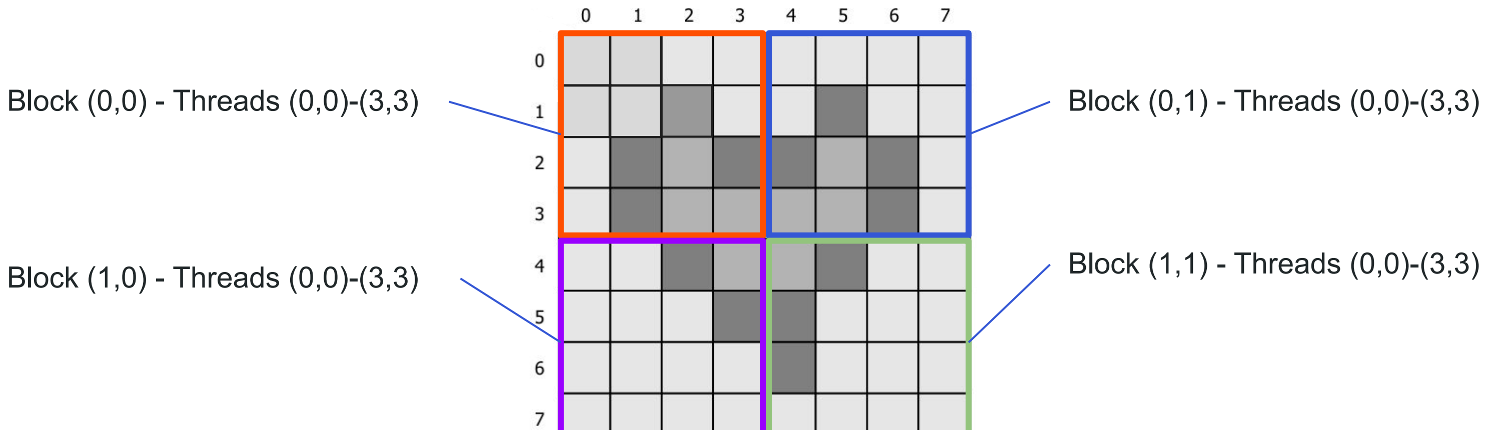
Using `matrix_elementwise_2d.cu`

```
[...]

#define ALPHA 0.25                  // Define the scalar
#define ROWS 2160                   // Define the matrix row number
#define COLS 4096                   // Define the matrix column number
#define THREADS_PER_BLOCK_X 32  // Define the number of threads in a block in x
#define THREADS_PER_BLOCK_Y 32  // Define the number of threads in a block in y

// CUDA kernel to perform elementwise multiplication
__global__ void matrix_elementwise(const float* M, float* P, const float alpha, const int rows, const int cols)
{
   // Calculate the thread ID of the overall grid
   int idrow = blockIdx.y * blockDim.y + threadIdx.y;
   int idcol = blockIdx.x * blockDim.x + threadIdx.x;
   int idx = idrow * rows + idcol;

   // Each thread computes one element of the result matrix
   // if (idrow < rows && idcol < cols) {
   if (idx < rows * cols) {
       P[idx] = alpha*M[idx];
   }
}

[...]
```

Scalar Matrix multiplication

```
[...]

    // Compute the dimensions of blocks and grid
    // Blocks are now 2-dimensional
    dim3 blockSize(THREADS_PER_BLOCK_X,THREADS_PER_BLOCK_Y);
    dim3 gridSize(ceil(float(ROWS)/blockSize.x),ceil(float(COLS)/blockSize.y));

    // Launch the CUDA kernel
    matrix_elementwise<<<gridSize, blockSize>>>(d_M, d_P, ALPHA, ROWS, COLS);

[...]
```

This way we are covering the matrix with a grid of 2D blocks and threads



0 1 2 3 4 5 6 7
0 1 2 3 4 5 6 7
Block (0,0) - Threads (0,0)-(3,3)
Block (0,1) - Threads (0,0)-(3,3)
Block (1,0) - Threads (0,0)-(3,3)
Block (1,1) - Threads (0,0)-(3,3)

Benchmarking CUDA

Timing on the GPU

How to time and benchmark the execution on GPU?

1. Use the standard `time` Unix function to measure the overall time of the executable
   - Including all code, including printout to terminal…
   - Measuring the time host-side, but GPU calls may be asynchronous!
2. Use the standard `time` Unix function to measure the overall time of the executable
   - A different timer, more complex to use…
   - Need to identify ***what*** to measure exactly

CUDA provides the `cudaEvents` facility ➜ special objects that can be used to mark points in your code and grant you access to the GPU timer.

CUDA Events

At its very basic (see the API documentation for more advanced usecases):

- Create and initialize `cudaEvent` objects (timers)
- Record the “start” Event before the kernel launch (or before the memory allocation, etc)
- Record the “stop” Event after the kernel launch
- Synchronize the host and device, waiting for the GPU to be done executing the kernel
- Extract the elapsed time (in ms) from the difference between the two events
- Destroy the two events

CUDA Events

At its very basic (see the API documentation for more advanced usecases):

- Create and initialize `cudaEvent` objects (timers)
- Record the “start” Event before the kernel launch (or before the memory allocation, etc)
- Record the “stop” Event after the kernel launch
- Synchronize the host and device, waiting for the GPU to be done executing the kernel
- Extract the elapsed time (in ms) from the difference between the two events
- Destroy the two events

See the example of `scalar_sum_with_timers.cu`

```
// Create the cudaEvent timers
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);

// Assign the start cudaEvent timer
cudaEventRecord(start);

// Launch the kernel
sum_kernel<<<grid,block>>>(...);

// Assign the stop cudaEvent timer and synchronize
cudaEventRecord(stop);
cudaEventSynchronize(stop);

// Print the time taken (in ms) between two events
float elapsed;
cudaEventElapsedTime(&elapsed,start,stop);
printf("Elapsed time (kernel): %.1f us\n",
        elapsed*1000);

// Destroy cudaEvents
cudaEventDestroy(start);
cudaEventDestroy(stop);
```

What to benchmark?

Benchmarking execution on GPU is not trivial ➔ execution can be compute-bound or memory-bound

FLOPS

Number of FP operations in a kernel per second

$$\text{FLOPS} = \frac{\text{n. FP operations in kernel}}{\text{elapsed kernel time (s)}}$$

Common metric for measuring performance in compute-bound tasks

Memory bandwidth

Amount of data transferred (read+write) per second

$$\text{B (GB/s)} = \frac{\text{size of memory transferred (GB)}}{\text{elapsed kernel time (s)}}$$

Common metric for measuring performance in memory-bound tasks

What to benchmark?

```
// Define a CUDA kernel that adds two floats
__global__ void sum_kernel(float *x,
                           float *y,
                           float *res){
   *res = *x + *y;
}

[...]

// Launch the sum kernel on the GPU
// with 1 block and 1 thread
   sum_kernel<<<1,1>>>(dev_a, dev_b, dev_c);
```

- 1 FP operation in kernel
- 3 FP transfer to/from global memory
  ➜ 3 x 4 B = 12 B

```
Elapsed time (kernel):  87.9 us
```

FLOPS

$$\text{FLOPS} = \frac{\text{n. FP operations in kernel}}{\text{elapsed kernel time (s)}}$$

= 11.4 MFLOPS

Memory bandwidth

$$\text{B (GB/s)} = \frac{\text{size of memory transferred (GB)}}{\text{elapsed kernel time (s)}}$$

= 136 kB/s (0.00014 GB/s)

What to benchmark?

FLOPS

$$\text{FLOPS} = \frac{\text{n. FP operations in kernel}}{\text{elapsed kernel time (s)}}$$

= 11.4 MFLOPS

Memory bandwidth

$$\text{B (GB/s)} = \frac{\text{size of memory transferred (GB)}}{\text{elapsed kernel time (s)}}$$

= 136 kB/s (0.00014 GB/s)

Compared with the spec. sheet of the GPU for the reference max values to estimate the GPU utilization

For the Jetson Nano 2GB Dev. Kit:

- **Max FLOPS: 472 GFLOPS (FP16)**
- **Max Bandwidth : 25.6 GB/s**

What to benchmark?

FLOPS

$$\text{FLOPS} = \frac{\text{n. FP operations in kernel}}{\text{elapsed kernel time (s)}}$$

$= 11.4$ MFLOPS

Memory bandwidth

$$\text{B (GB/s)} = \frac{\text{size of memory transferred (GB)}}{\text{elapsed kernel time (s)}}$$

$= 136$ kB/s (0.00014 GB/s)

Compared with the spec. sheet of the GPU for the reference max values to estimate the GPU utilization

For the Jetson Nano 2GB Dev. Kit:

- **Max FLOPS: 472 GFLOPS (FP16)**
- **Max Bandwidth : 25.6 GB/s**

0.0024 % $\Rightarrow$ **SEVERE UNDERUTILIZATION OF THE GPU!** $\Leftarrow$ **0.00053 %**

Work at home/lab

- matrix-matrix addition
- vector-matrix multiplication
- boost grayscale image luminosity by 20%