Programming Massively Parallel Processors

Chapter03 Multidimensional grids and data

&

Chapter04 Compute architecture and scheduling

1944023 JiYeong Yi

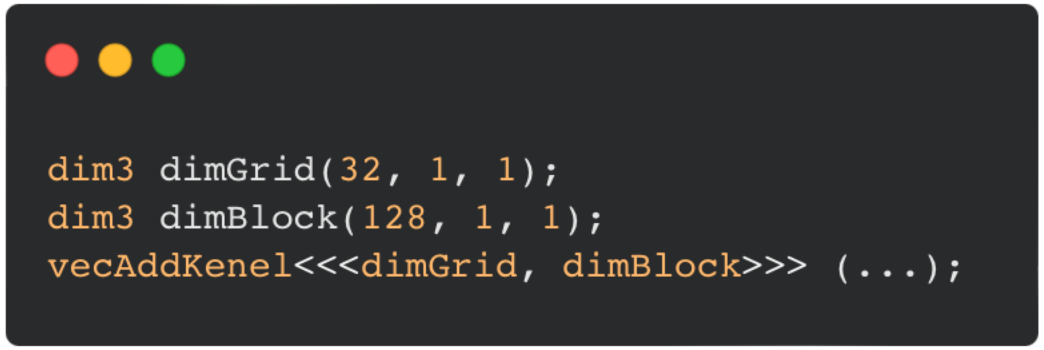
Chapter03

Multidimensional grids and data

Table of contents

3.1 Multidimensional grid organization

- The execution configuration parameters in a kernel call statement specify the dimensions of the grid and the dimensions of each block.
    - The first execution configuration parameter specifies the dimensions of the grid in number of blocks.
    - The second specifies the dimensions of each block in number of threads.
    - Each such parameter has the type dim3, which is an integer vector type of three elements x, y, and z.



```
dim3 dimGrid(32, 1, 1);
dim3 dimBlock(128, 1, 1);
vecAddKenel<<<dimGrid, dimBlock>>> (...);
```

- The programmer can use fewer than three dimensions by setting the size of the unused dimensions to 1.

3.1 Multidimensional grid organization

- gridDim and blockDim are built-in variables in a kernel and always reflect the dimensions of the grid and blocks, respectively.

- grid dimension value range
  - gridDim.x : 1 ~ (2^{31} - 1)
  - gridDim.y : 1 ~ (2^{16} - 1)
  - gridDim.z : 1 ~ (2^{16} - 1)

- The total size of a block in current CUDA systems is limited to 1024 threads.
  - (512, 1, 1), (8, 16, 4), and (32, 16, 2) are allowed.
  - (32, 32, 2) are not allowed. (32 x 32 x 2 = 2048)

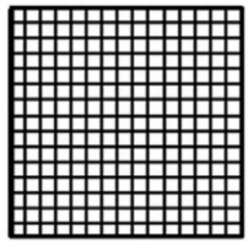
3.1 Multidimensional grid organization

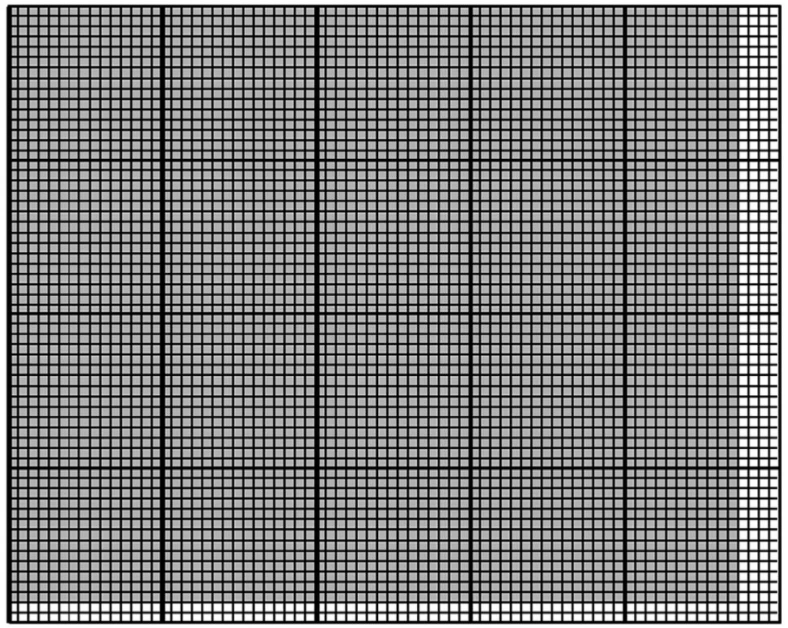
- The highest dimension takes precedence in the order of block and thread labels.
  This notation uses an ordering that is the reverse of that used in the C statements for setting configuration parameters.

Block (0, 1)	(blockIdx.y, blockIdx.x) blockIdx.y = 0, blockIdx.x = 1
Thread (4, 2, 1)	(threadIdx.z, threadIdx.y, threadIdx.x) threadIdx.z = 4, threadIdx.y = 2, threadIdx.x = 1

3.2 Mapping threads to multidimensional data

- The choice of 1D, 2D, and 3D thread organization is usually based on the nature of the data.
- For example, pictures are a 2D array of pixels. Using a 2D grid that consist of 2D blocks is often convenient for processing the pixels in a picture.


16×16 blocks



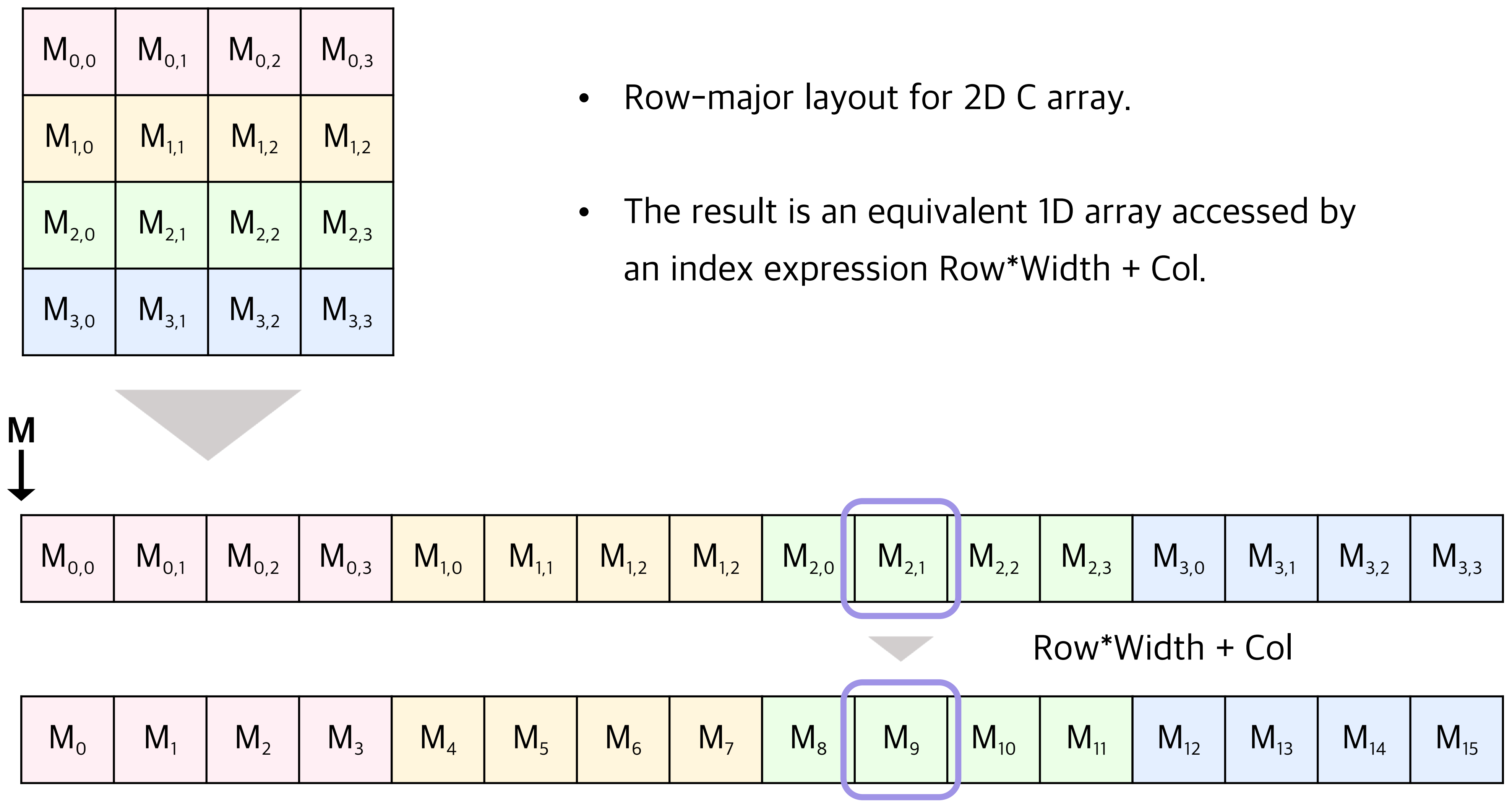
- Using a 2D thread grid to process a 62 X 76 picture.

y direction : 4 blocks
x direction : 5 blocks
Total : 20 blocks

3.2 Mapping threads to multidimensional data

- Memory space has a “flat” organization, where there is only one address for every location.
- As a result, all multidimensional arrays are ultimately “**flattened**” into equivalent one-dimensional array.
- There are at least two ways in which a 2D array can be linearized.
  1. row-major layout: place all elements of the same row into consecutive location.
  2. column-major layout: place all elements of the same column into consecutive location.
- CUDA C uses the **row-major layout** rather than column-major layout.

3.2 Mapping threads to multidimensional data



M
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$
$M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,2}$
$M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$
$M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$ $M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,2}$ $M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$ $M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$
Row*Width + Col
M_0 M_1 M_2 M_3 M_4 M_5 M_6 M_7 M_8 M_9 M_{10} M_{11} M_{12} M_{13} M_{14} M_{15}

- Row-major layout for 2D C array.
- The result is an equivalent 1D array accessed by an index expression Row*Width + Col.

3.2 Mapping threads to multidimensional data

- Color to Gray scale conversion
  - Experiment Using the same 3 images of different sizes.
  - Block Size : 16 x 16

1024 x 1024

2048 x 2048

4096 x 4096

3.2 Mapping threads to multidimensional data

- Result

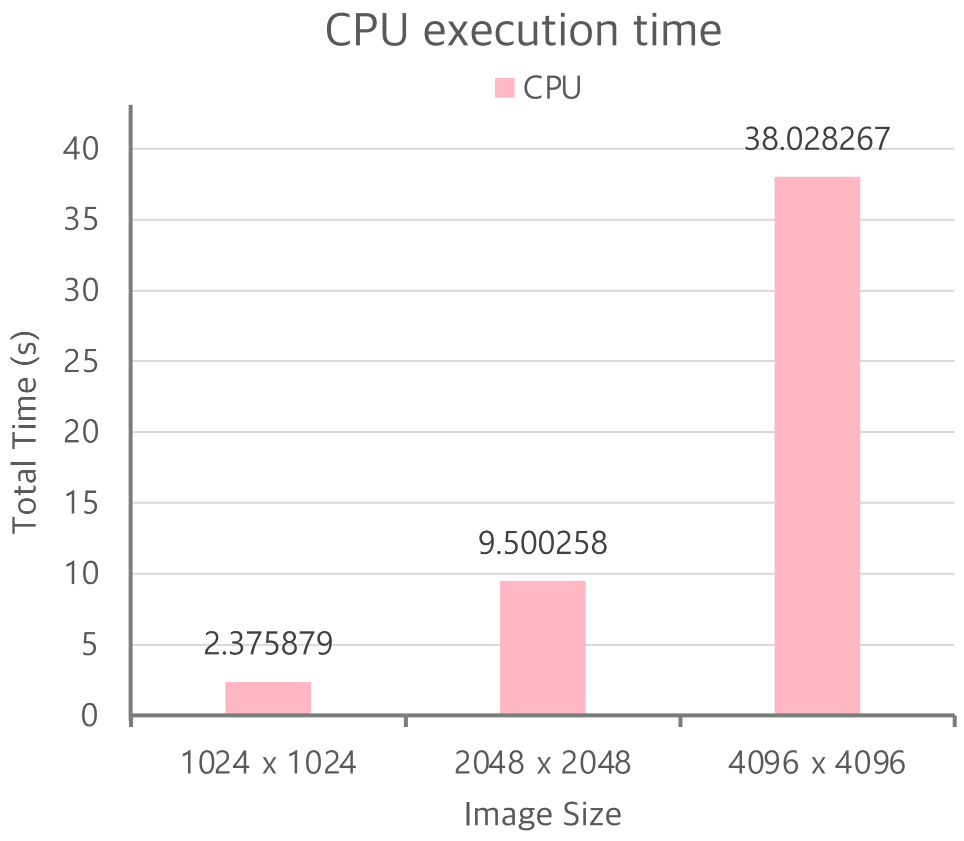
1024 x 1024

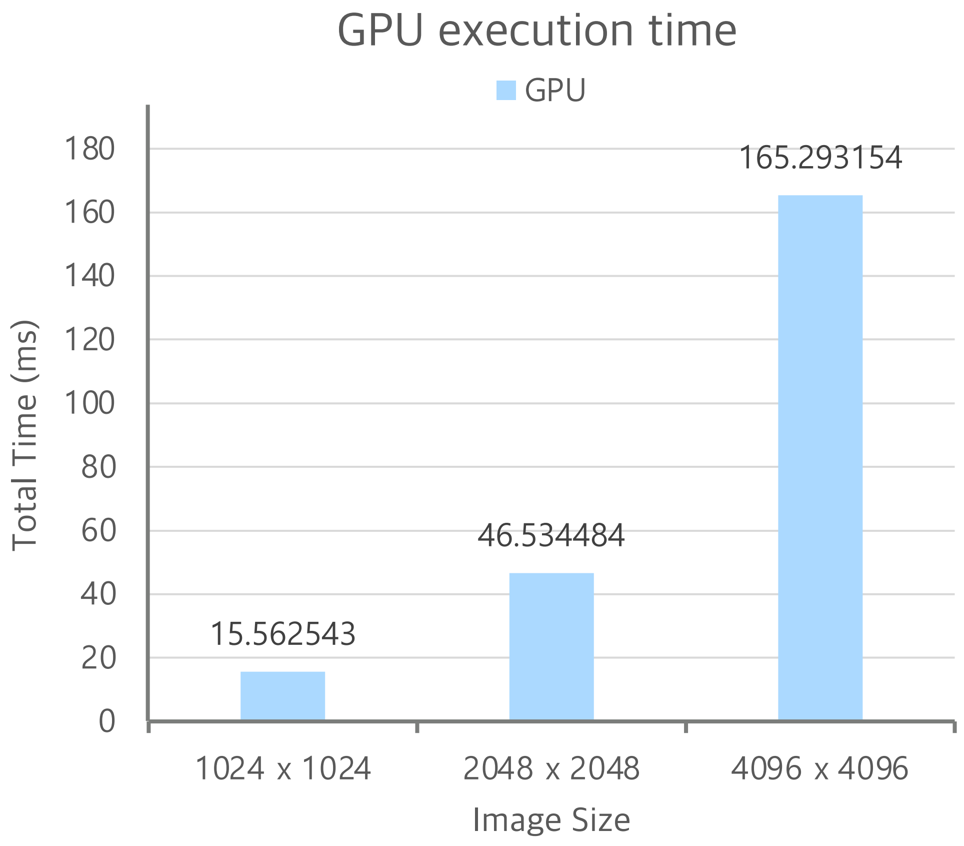
2048 x 2048

4096 x 4096

3.2 Mapping threads to multidimensional data

- Result





3.3 Image blur: a more complex kernel

- Image blur
  - Calculates the value of an output image pixel as the average of a patch of pixels encompassing the pixel in the input image.
  - Experiment using 4 patches of different sizes for 1024 x 1024 size image.

1024 x 1024

3.3 Image blur: a more complex kernel

- Result

3 x 3 patch

5 x 5 patch

3.3 Image blur: a more complex kernel

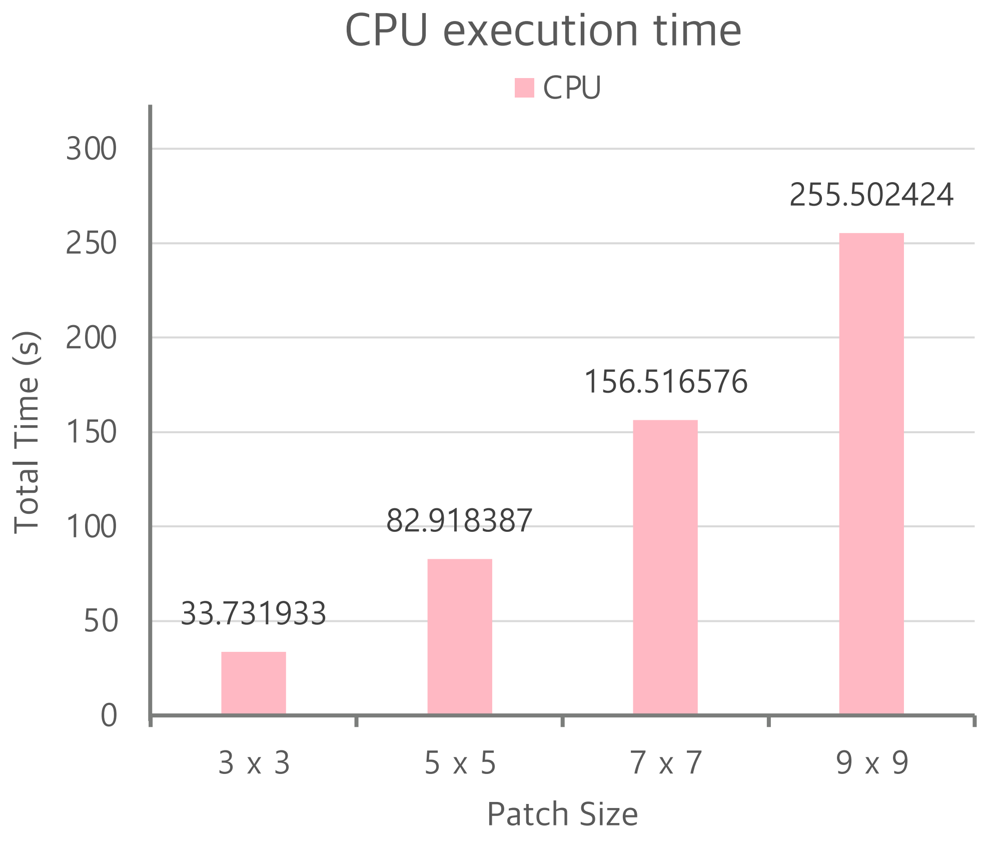
- Result

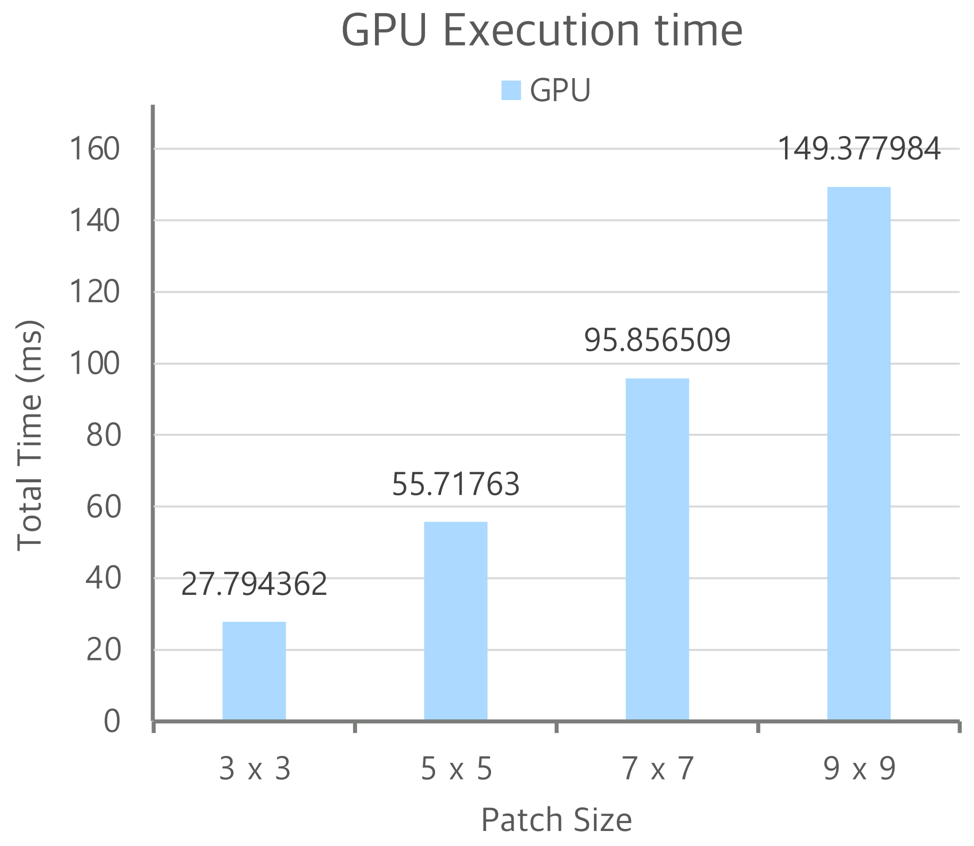
7 x 7 patch

9 x 9 patch

3.3 Image blur: a more complex kernel

- Result





3.4 Matrix multiplication

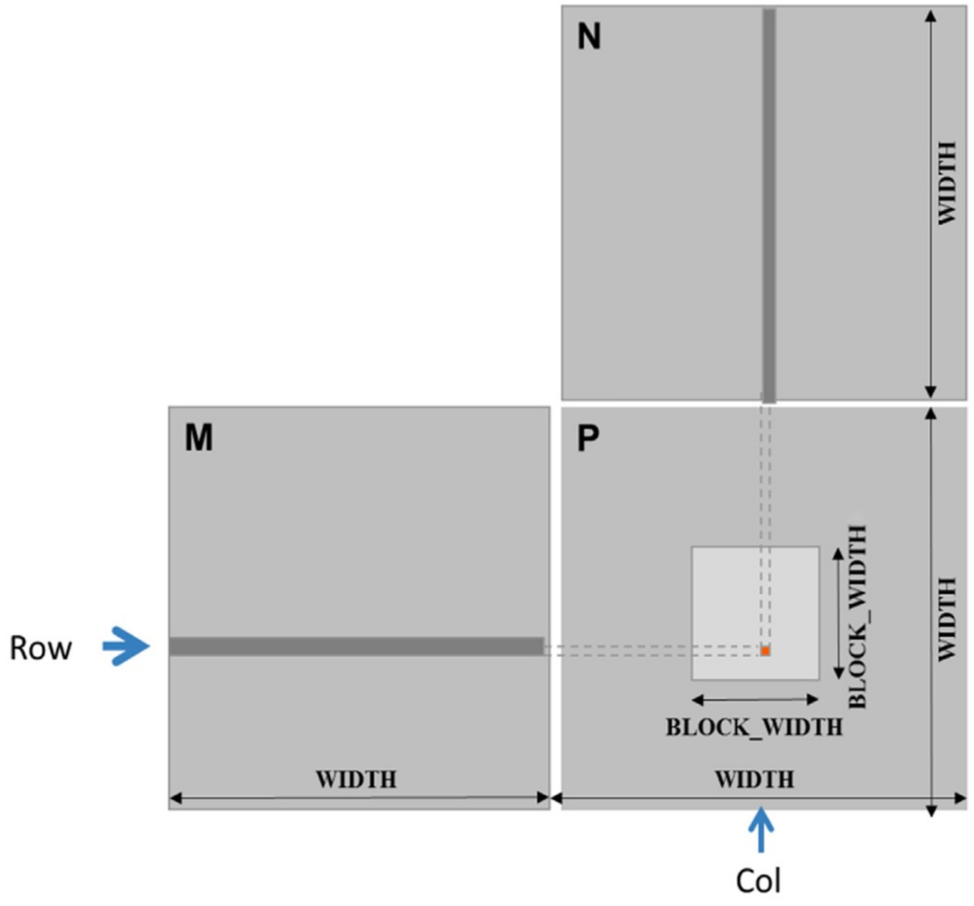
- Matrix multiplication between an i x j (i rows by j columns) matrix M and a j x k matrix produces an i x k matrix P.

- When a matrix multiplication is performed, each element of the output matrix P is an inner product of a row of M and a column of N.

$$P_{row,col} = \sum M_{row,k} * Nk_{,col}$$

(for k = 0, 1,...Width - 1)

3.4 Matrix multiplication



- To implement matrix multiplication using CUDA, we can map the threads in the grid to the elements of the output matrix P.
- Each thread is responsible for calculating one P element.

```
row = blockIdx.y * blockDim.y + threadIdx.y
col = blockIdx.x * blockDim.x + threadIdx.x
```

3.5 Summary

CUDA grids and blocks are multidimensional with up to three dimensions.

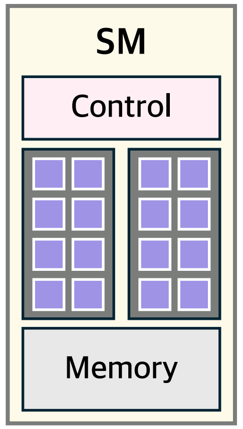
The **multidimensionality of grids and blocks** is useful for organizing threads to be mapped to **multidimensional data**.

Chapter04

Compute architecture and scheduling

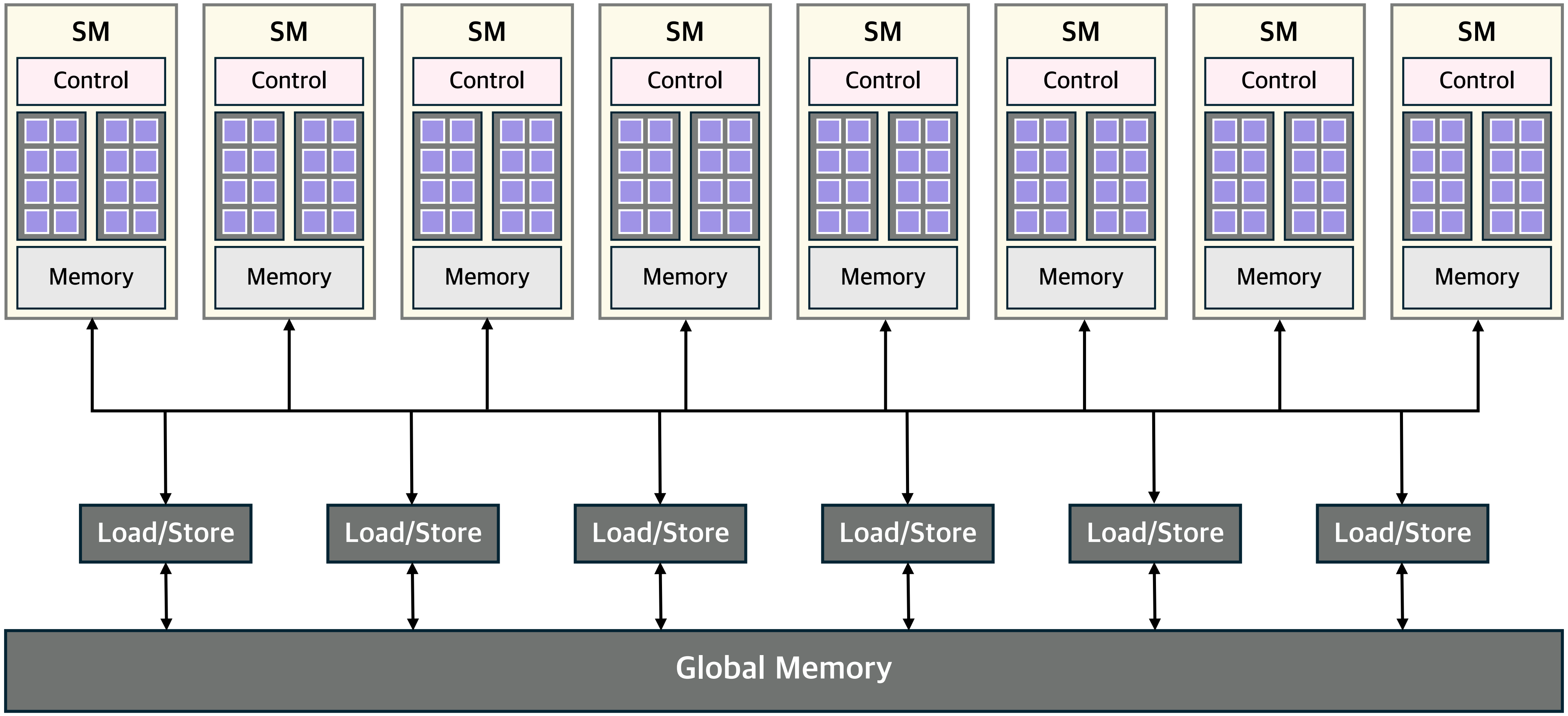
Table of contents

4.1 Architecture of a modern GPU



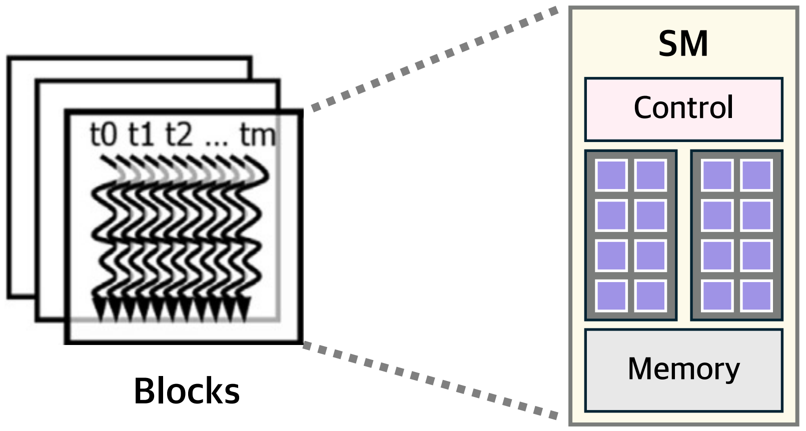
- CUDA-capable GPU is organized into an array of highly threaded **streaming multiprocessors** (SMs).
- Each SM has several processing units called **streaming processors** or **CUDA cores** (hereinafter referred to as just cores for brevity) that share control logic and memory resource.
- The SMs come with different on-chip memory structures and gigabytes of off-chip device memory.

4.1 Architecture of a modern GPU



4.2 Block scheduling

- When a kernel is called, the CUDA runtime system launches a grid of threads that execute the kernel code.
- These threads are assigned to SMs on a block-by-block basis.
- That is, **all thread in a block** are simultaneously assigned to the **same SM**.



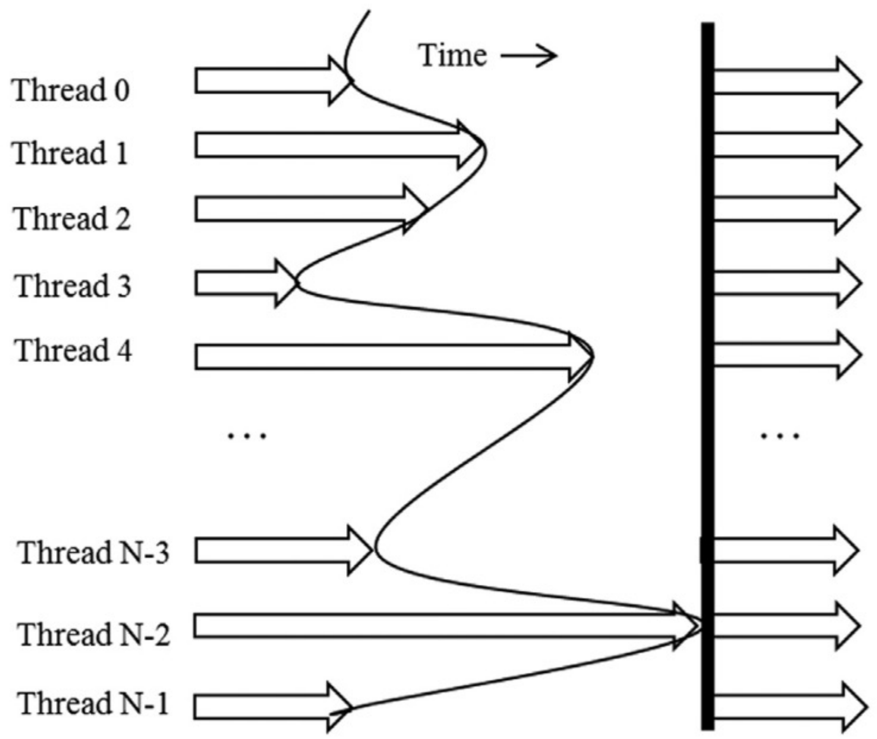
4.2 Block scheduling

- However, blocks need to reserve hardware resources to execute, so only a limited number of blocks can be simultaneously assigned to a given SM.

- With a limited number of SMs and a limited number of blocks that can be simultaneously assigned to each SM, there is a **limit on the total number of blocks that can be simultaneously executing in a CUDA device**.

- To ensure that all blocks in a grid get executed, **the runtime system maintains a list of blocks** that need to execute and assigns new blocks to SMs when previously assigned blocks complete execution.

4.3 Synchronization and transparent scalability

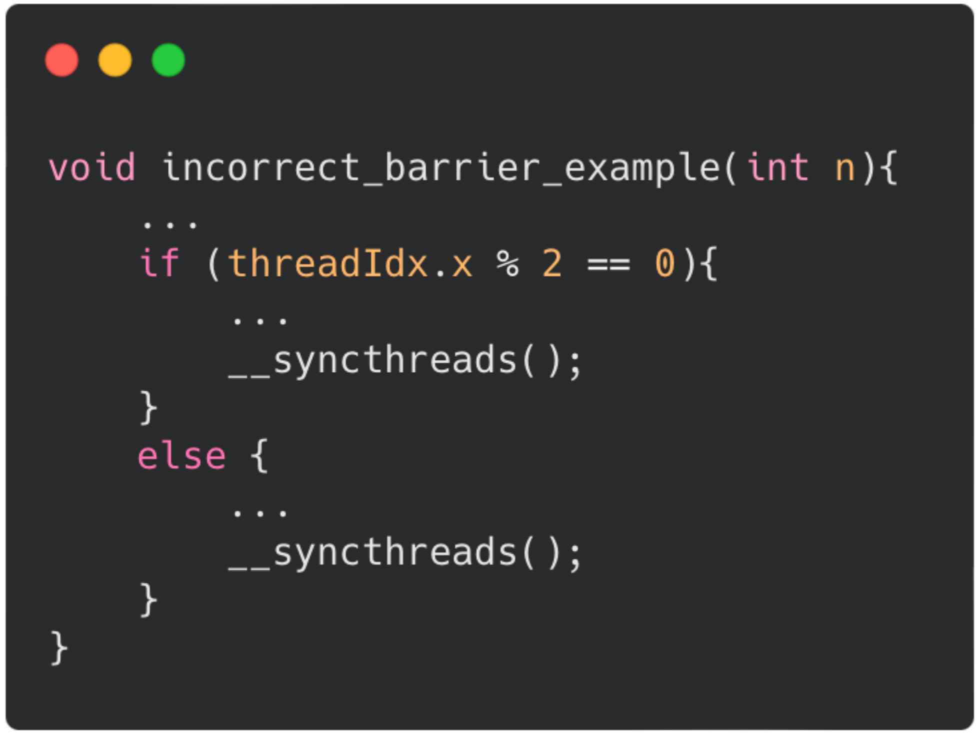


- CUDA allows threads in the same block to coordinate their activities using the barrier synchronization function __syncthreads().

- When a thread calls __syncthreads(), it will be held at the program location of the call until every thread in the same block reaches that location.

- This ensures that all threads in a block have completed a phase of their execution before any of them can move on to the next phase.

4.3 Synchronization and transparent scalability



```
void incorrect_barrier_example(int n){
    ...
    if (threadIdx.x % 2 == 0){
        ...
        __syncthreads();
    }
    else {
        ...
        __syncthreads();
    }
}
```

An incorrect use of __syncthreads()

- In CUDA, if a __syncthreads() statement is present, it must be executed by all threads in a block.

- In general, incorrect usage of barrier synchronization can result in incorrect result, or in threads waiting for each other forever, which is referred to as a deadlock.

4.3 Synchronization and transparent scalability

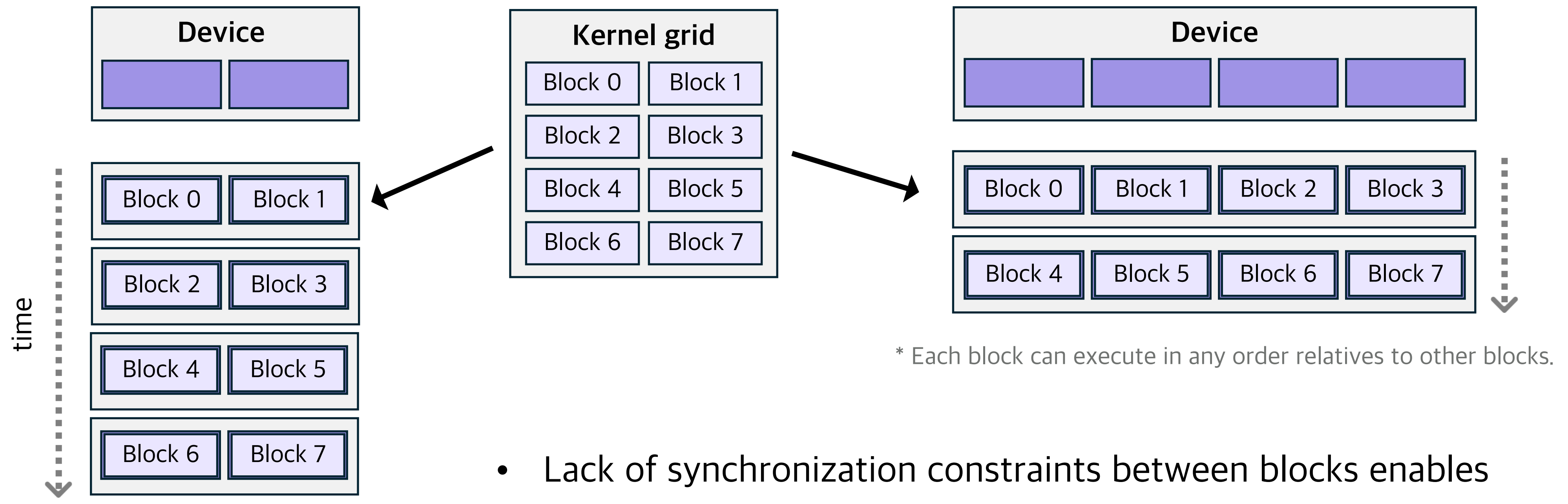
- Barrier synchronization imposes execution constraints on threads within a block.

  1. Threads within a block should execute in close time proximity with each other to avoid excessively long waiting times.
     - Not only do all threads in a block have to be assigned to the same SM, but also they need to be assigned to that SM simultaneously.

  2. The system needs to make sure that all threads involved in the barrier synchronization have access to the necessary resources to eventually arrive at the barrier.
     - a block can begin execution only when the runtime system has secured all the resources needed by all threads in the block to complete execution.

4.3 Synchronization and transparent scalability

- By not allowing threads in different blocks to perform barrier synchronization with each other, the CUDA runtime system can execute blocks in any order relative to each other, since none of them need to wait for each other.



* Each block can execute in any order relatives to other blocks.

- Lack of synchronization constraints between blocks enables transparent scalability for CUDA programs.

4.3 Synchronization and transparent scalability

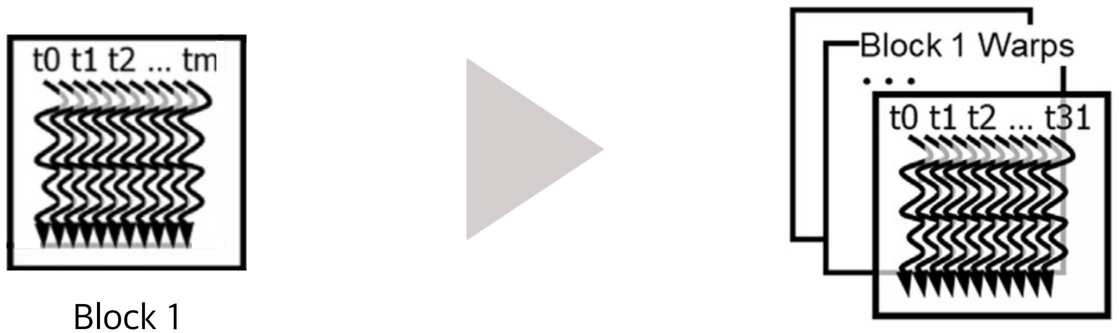
Transparent scalability

The ability to execute the same application code on different hardware
with different amounts of execution resources is referred to as transparent scalability.

4.4 Warps and SIMD hardware

- Once a block has been assigned to an SM, it is further divided into 32-thread units called warps.
  A warp is the unit of thread scheduling in SMs.

- Each warp consists of 32 threads of consecutive threadIdx values.



Block 1

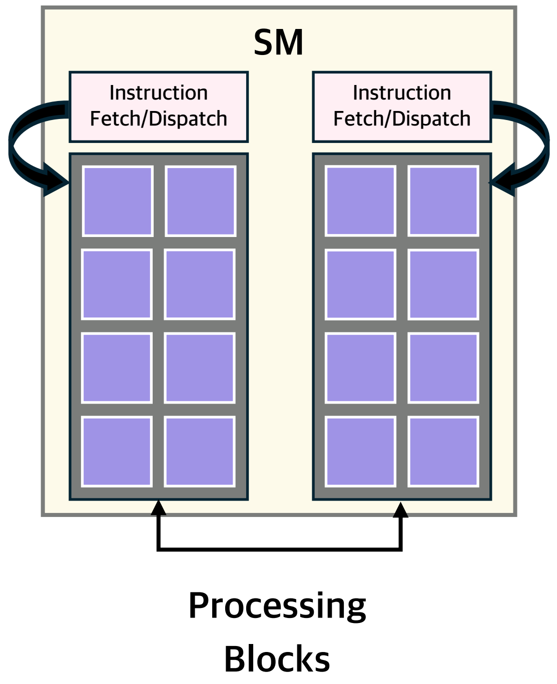
4.4 Warps and SIMD hardware

- Blocks are partitioned into warps on the basis of thread indices.

  1. block that consist of one-dimensional threads
     - Only threadIdx.x is used. The threadIdx.x values within a warp are consecutive and increasing.
  2. block that consist of multiple-dimensional threads
     - The dimensions will be projected into a linearized row-major layout before partitioning into warps.
     - The linear layout is determined by placing the rows with larger y and z coordinates after those with lower ones. (z -> y -> x)
  3. For a block whose size is not a multiple of 32, the last warp will be padded with inactive threads to fill up the 32 thread position.

4.4 Warps and SIMD hardware

- An SM is designed to execute all threads in a warp following the single-instruction, multiple-data (SIMD) model. That is, at any instant in time, one instruction is fetched and executed for all thread in the warp.



- Cores in SM are grouped into processing blocks and share an instruction fetch/ dispatch unit.
- Threads in the same warp are assigned to the same processing block, which fetches the instruction for the warp and executes it for all threads in the warp at the same time.
- The advantage of SIMD is that the cost of the control hardware, such as the instruction fetch/dispatch unit, is shared across many execution units.

4.5 Control divergence

- SIMD execution works well when all threads within a warp follow the same execution path, more formally referred to as control flow, when working on their data.

- However, when threads within a warp take different control flow paths, the SIMD hardware will take multiple passes through these paths, one pass for each path.

- When threads in the same warp follow different execution paths, we say that these threads exhibit control divergence, that is, they diverge in their execution.

4.5 Control divergence

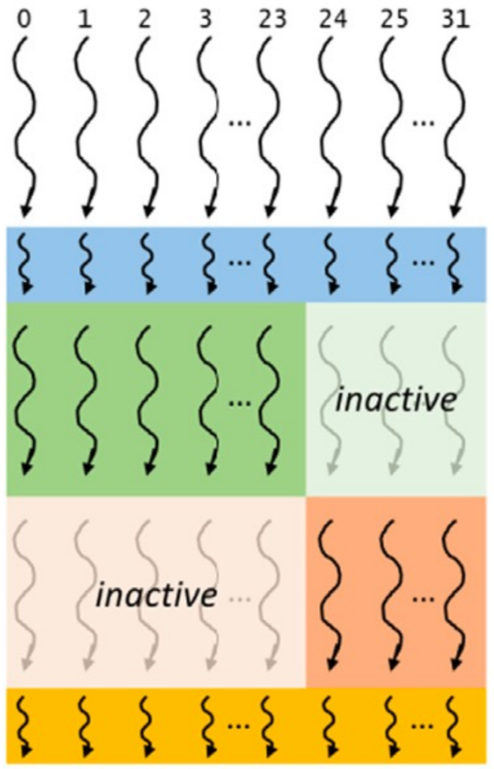
```
if(threadIdx.x < 24) {

    A

} else {

    B

}
C
```



- While the hardware executes the same instruction for all threads in a warp, it selectively lets these threads take effect in only the pass that corresponds to the path that they took, allowing every thread to appear to take its own control flow path.

4.5 Control divergence

An important implication of control divergence is that one cannot assume that all threads in a warp have the same execution timing. Therefore, if all threads in a warp must complete a phase of their execution before any of them can move on, one must use a barrier synchronization mechanism such as **__syncwarp()** to ensure correctness.

4.6 Warp scheduling and latency tolerance

- When threads are assigned to SMs, there are usually more threads assigned to an SM than there are cores in the SM.

- That is, each SM has only enough execution units to execute a subset of all the threads assigned to it at any point in time.

- The mechanism of filling the latency time of operations from some threads with work from other threads is often called "**latency tolerance**" or "**latency hiding**".

4.6 Warp scheduling and latency tolerance

- The selection of warps that are ready for execution does not introduce any idle or wasted time into the execution timeline, which is referred to as **zero-overhead thread scheduling**.
- GPU SMs achieves zero-overhead scheduling by holding all the execution states for the assigned warps in the hardware registers so there is no need to save and restore states when switching from one warp to another.

4.7 Resource partitioning and occupancy

- It is desirable to assign many warps to an SM in order to tolerate long-latency operations. However, it may not always be possible to assign to the SM the maximum number of warps that the SM supports.

- The execution resources in an SM include registers, shared memory, thread block slots, and thread slots. These resources are dynamically partitioned across threads to support their execution.

  (+) This ability to dynamically partition thread slots among blocks makes SMs versatile.

  (-) Dynamic partitioning of resources can lead to subtle interactions between resource limitations, which can cause underutilization of resources.

4.7 Resource partitioning and occupancy

$$\text{Occupancy} = \frac{\text{the number of threads assigned to an SM}}{\text{the maximum number of threads supported by an SM}}$$

4.8 Querying device properties

- The CUDA runtime system provides API functions that provide information about the available resources and capabilities of the device.

- API functions

cudaGetDeviceCount	Return the number of available CUDA device.
cudaGetDeviceProperties	Return the properties of the device.

- Built-in type

cudaDeviceProp	C struct type with fields that represent the properties of a CUDA device.

4.8 Querying device properties

- cudaDeviceProp fields

maxThreadsPerBlock	The maximum number of threads allowed in a block
multiProcessorCount	The number of SMs in the device
clockRate	Clock frequency of the device
maxThreadsDim[i]	The maximum number of threads allowed along each dimension of a block (i = 0 for the x dimension, i = 1 for the y dimension, i = 2 for the z dimension)
maxGridSize[i]	The maximum number of blocks allowed along each dimension of a grid (i = 0 for the x dimension, i = 1 for the y dimension, i = 2 for the z dimension)
regsPerBlock	The number of registers that are available in each SM
warpSize	The size of warps