中国科学院大学
University of Chinese Academy of Sciences

Introduction to CUDA

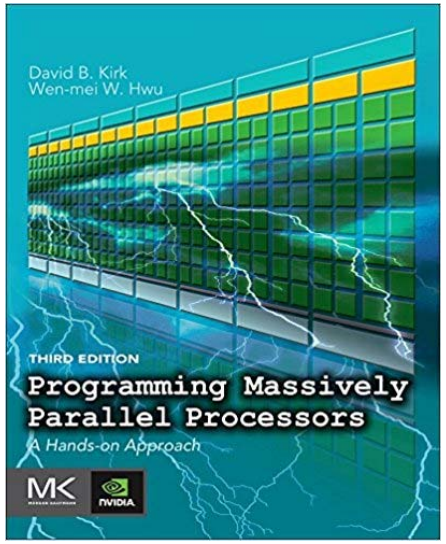
(3) Scalable Parallel Execution

Reference

- CUDA C Programming Guide,
  - https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

- **Programming Massively Parallel Processors,**
  - **A Hands-on Approach**
  - **Third Edition**

  - **Chapter 3**



Content

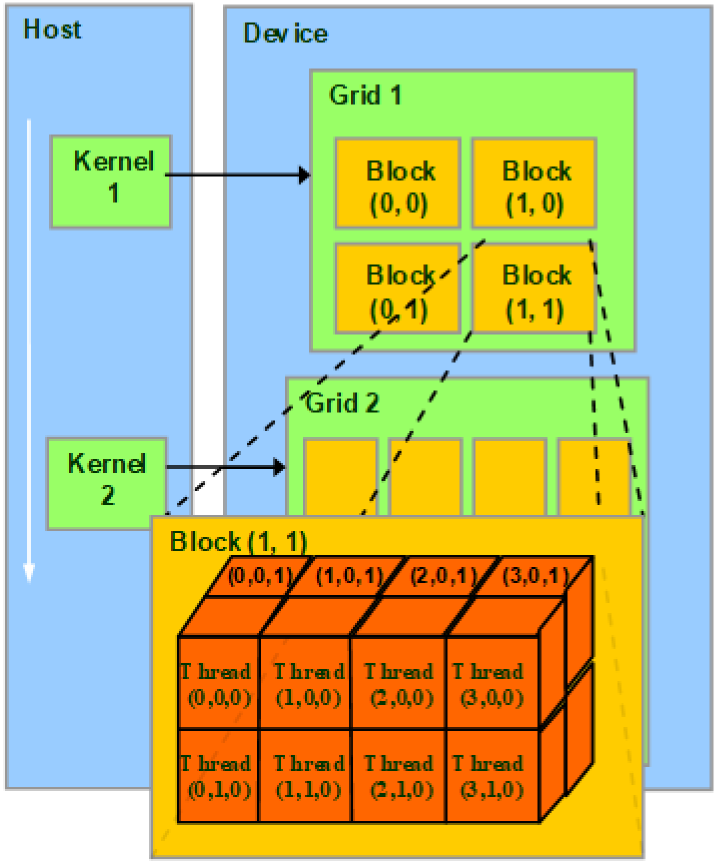
- CUDA Thread Organization
- Mapping Threads to Multidimensional Data
  - Example – Image Blur
- Synchronization and Transparent Scalability
- Thread Scheduling and Latency Tolerance

Thread Organization

Two levels:

- A grid is a 3-D array of blocks;
- And each block is a 3-D array of threads.

- use fewer than three dimensions *by setting the size of the unused dimensions to 1.*



Threads Configuration

- Execution configuration parameters:

<<< A, B>>>

  - A: dimensions of the grid in the number of blocks.
  - B: dimensions of each block in the number of threads.

- Each such parameter is of the *dim3* type:
  - a C struct with three unsigned integer fields:
  - *x*, *y*, and *z*.

Threads Configuration

- Configures to launch the vecAddkernel():

```
dim3 dimGrid(32, 1, 1);
dim3 dimBlock(128, 1, 1);
vecAddKernel<<<dimGrid, dimBlock>>>(…);
```

- The grid and block dimensions can also be calculated from other variables

```
dim3 dimGrid(ceil(n/256.0), 1, 1);
dim3 dimBlock(256, 1, 1);
vecAddKernel<<<dimGrid, dimBlock>>>(…);
```

Threads Configuration

- special shortcut for launching a kernel with 1-D grids and blocks:
  - takes the arithmetic expression as the *x* dimensions and assumes that the *y* and *z* dimensions are 1.

```
vecAddKernel<<<ceil(n/256.0), 256>>>(…);
```

  - the *x* field is the first field of the dim3 structures gridDim(x, y, z) and blockDim{x, y, z).

Built-in Variables

- gridDim and blockDim：
  - always reflect the dimensions of the grid and the blocks.
  - part of the CUDA C specification and *cannot be changed*.
- gridDim:
  - the allowed values of gridDim.x, gridDim.y and gridDim.z range from 1 to 65,536.
  - All threads in a block share the same blockIdx.x, blockIdx.y, and blockIdx.z values.
  - all blocks in a grid have the same dimensions and sizes.

Built-in Variables

- gridDim and blockDim：
  - always reflect the dimensions of the grid and the blocks.
  - part of the CUDA C specification and *cannot be changed*.
- blockDim:
  - The total size of a block is ***limited to 1024 threads***, with flexibility in distributing these elements into the three dimensions as long as the total number of threads does not exceed 1024.
    - blockDim(512, 1, 1) ? 512
    - blockDim(8, 16, 4) ? 512
    - blockDim(32, 32, 2) ? 2048

Dimensionality

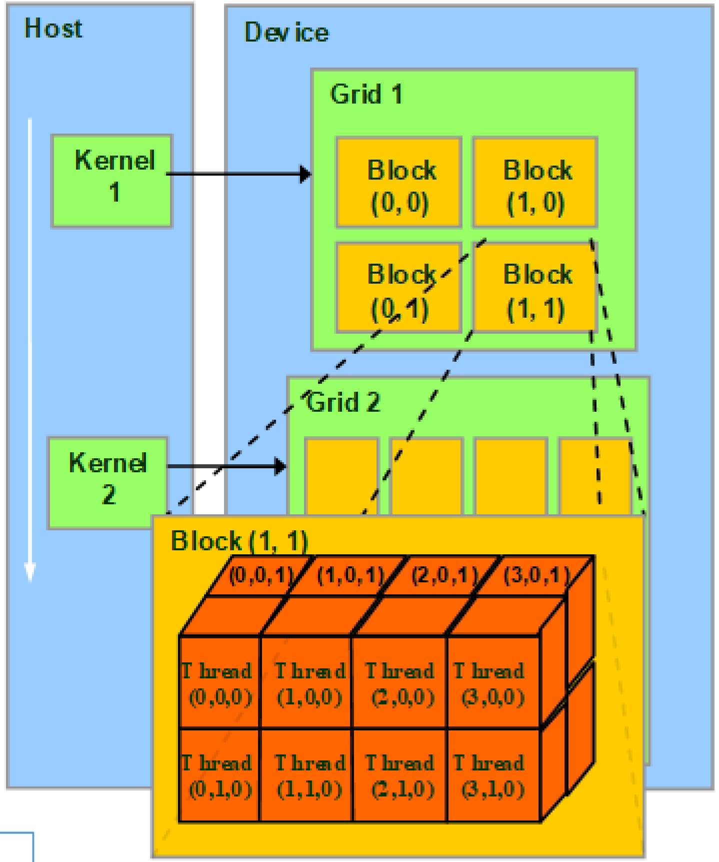
- The grid can have higher dimensionality than its blocks and vice versa:

```
dim3 dimGrid(2, 2, 1);
dim3 dimBlock(4, 2, 2);
```

The labels are ordered such that the highest dimension comes first.

(blockIdx.y,blockIdx.x)

(threadIdx.z, threadIdx.y, threadIdx.x)



Block(1,0) has blockIdx.y=1 and blockIdx.x=0.

Thread(1,0,2) has threadIdx.z=1, threadIdx.y=0, and threadIdx.x=2.

Content

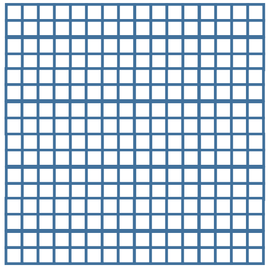
- CUDA Thread Organization
- Mapping Threads to Multidimensional Data
  - Example – Image Blur
- Synchronization and Transparent Scalability
- Thread Scheduling and Latency Tolerance

Thread organizations

- How to choose 1D, 2D, or 3D?
  - based on the nature of the data.

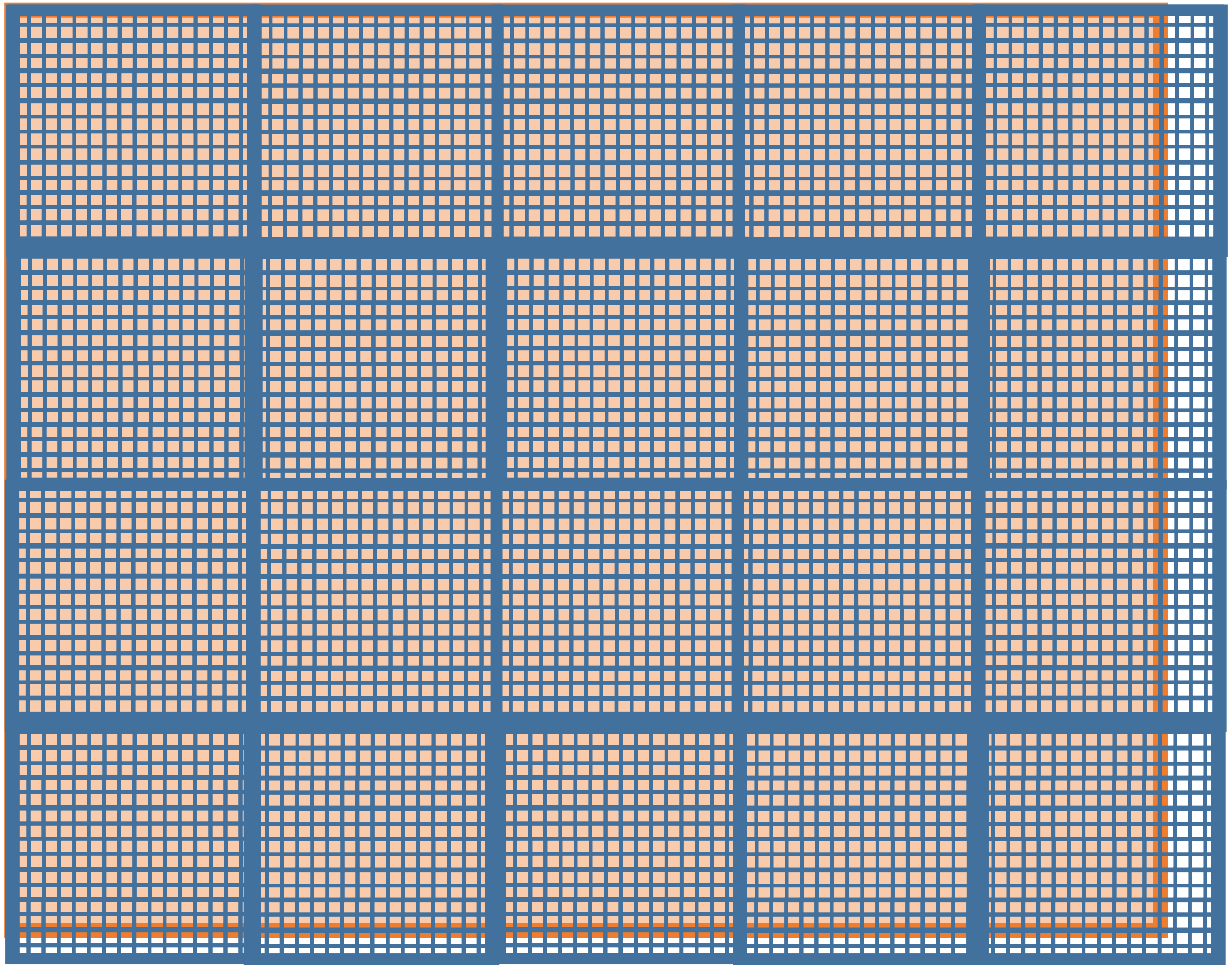
- Pictures are 2D array of pixels.
  - Using a 2D grid that consists of 2D blocks is often convenient for processing the pixels in a picture..

Processing a Picture with a 2D Grid



16×16 blocks

dimGrid = ?
dimBlock= ?



76x62 picture

Processing a Picture with a 2D Grid

- Thread organization for 76x62 pic:
  - Assume that we decided to use a 16 × 16 block: with 16 threads in the *x* direction and 16 threads in the *y* direction.
  - 5 blocks in the *x* direction and 4 blocks in the *y* direction.

- Identify the Pin element processed by thread(0,0) of block(1,0) with the formula:

$$P_{\text{blockIdx.y*blockDim.y+threadIdx.y,blockIdx.x*blockDim.x+threadIdx.x}}$$

$$= P_{1*16+0,0*16+0} = P_{16,0}.$$

Processing a Picture with a 2D Grid

- Assume:
  - `m` - number of pixels in the *x* direction
  - `n` - number of pixels in the *y* direction
- launch a 2D kernel:

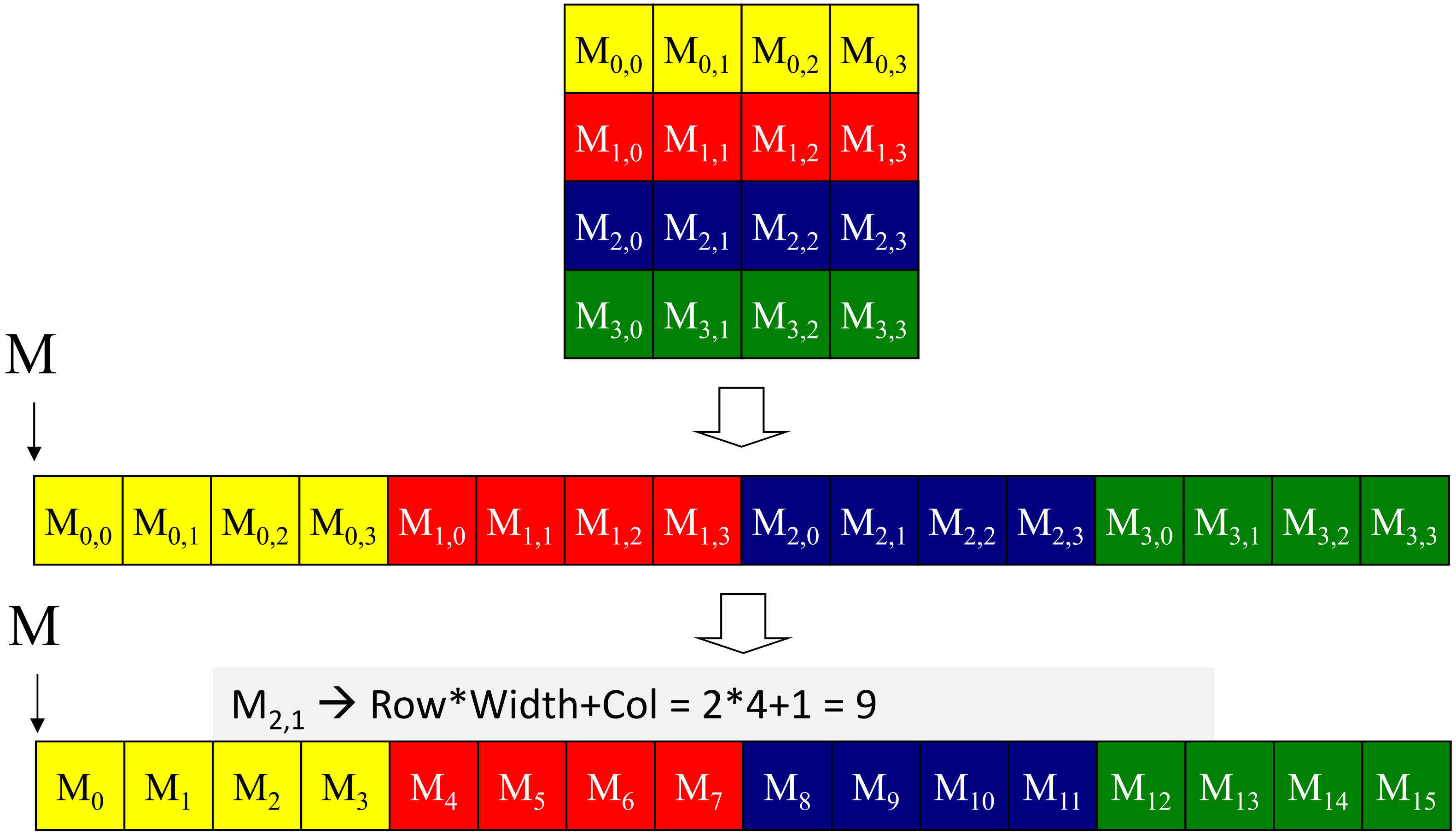
```
dim3 dimGrid(ceil(m/16.0), ceil(n/16.0), 1);
dim3 dimBlock(16, 16, 1);
colorToGrey<<<dimGrid,dimBlock>>>(d_Pin,d_Pout,m,n);
```

- Q: to process a 2000 × 1500 (3-million-pixel) picture:
  - ? gridDim.x, gridDim.y
  - ? blockDim.x, and blockDim.y

Access Elements of Array

- How C statements access elements of dynamically allocated multidimensional arrays:
  - All locations are labeled with an address ranging from 0 to the largest number.
  - Every location has only one address; thus, we say that the memory space has a “flat” organization.
  - In reality, all multidimensional arrays in C are linearized because of the use of a “flat” memory space in modern computers.

Row-Major Layout of 2D arrays in C/C++

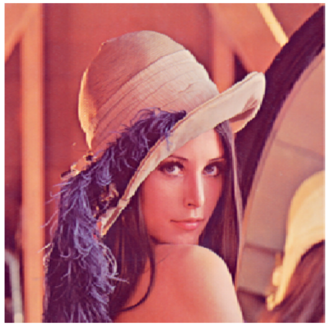


colorToGreyscaleConversion Kernel with 2D thread mapping to data

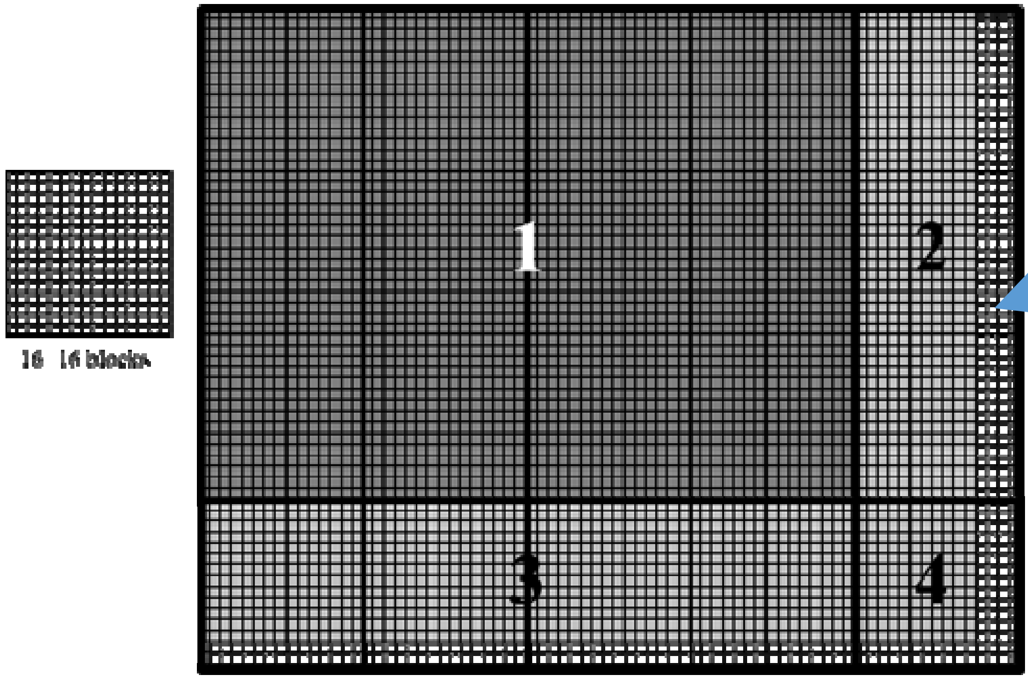
```
// we have 3 channels corresponding to RGB
// The input image is encoded as unsigned characters [0, 255]
__global__
void colorToGreyscaleConvertion(unsigned char * Pout,  unsigned char * Pin,
                    int width, int height) {

 int Col =   threadIdx.x + blockIdx.x * blockDim.x;
 int Row = threadIdx.y + blockIdx.y * blockDim.y;

 if (Col < width && Row < height) {
   // get 1D coordinate for the grayscale image
   int greyOffset = Row*width + Col;
   // one can think of the RGB image having
   // CHANNEL times columns of the gray scale image
   int rgbOffset = greyOffset*CHANNELS;
   unsigned char r =  rgbImage[rgbOffset     ]; // red value for pixel
   unsigned char g = rgbImage[rgbOffset + 1]; // green value for pixel
   unsigned char b = rgbImage[rgbOffset + 2]; // blue value for pixel
   // perform the rescaling and store it
   // We multiply by floating point constants
   grayImage[grayOffset] = 0.21f*r + 0.71f*g + 0.07f*b;
 }
}
```



Covering a 76×62 picture with 16×16 blocks

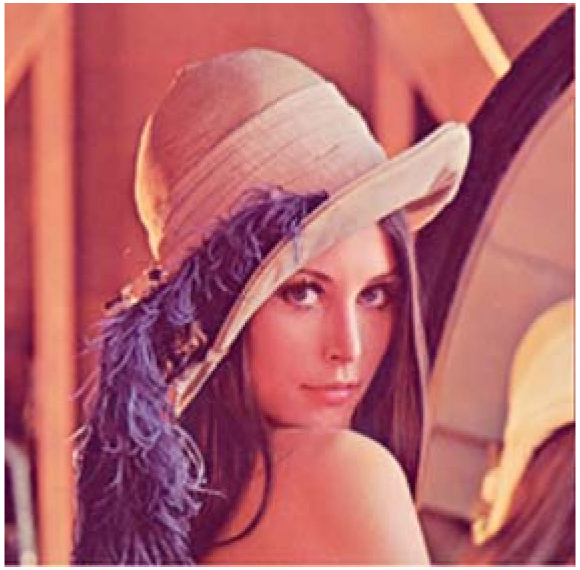


Test (Col < width)

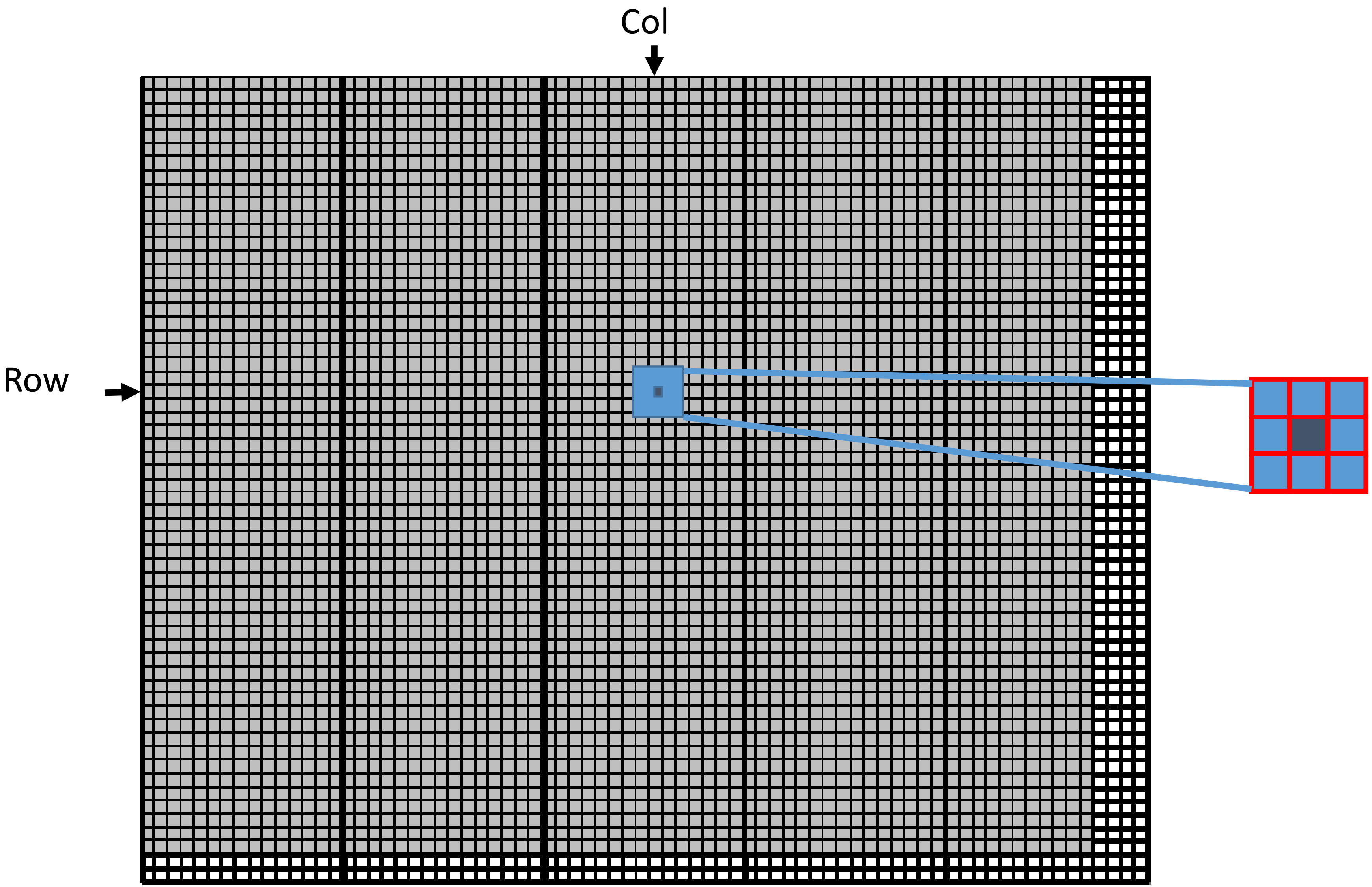
if (Col < width && Row < height)

Example

- Image Blur: A More Complex Kernel



Each output pixel is the average of pixels around it (BLRU_SIZE = 1)



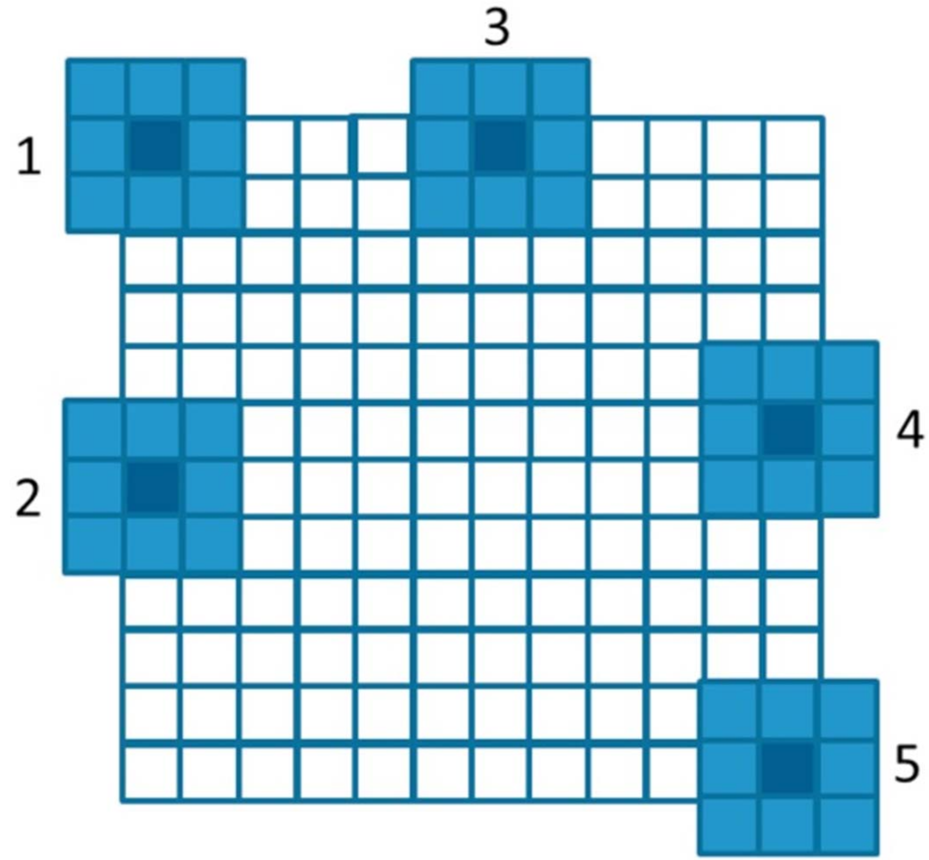
An Image Blur Kernel

```
  __global__
  void blurKernel(unsigned char * in, unsigned char * out, int w, int h) {
    int Col  = blockIdx.x * blockDim.x + threadIdx.x;
    int Row  = blockIdx.y * blockDim.y + threadIdx.y;

    if (Col < w && Row < h) {
1.      int pixVal = 0;
2.      int pixels = 0;
        // Get the average of the surrounding BLUR_SIZE x BLUR_SIZE box
3.      for(int blurRow = -BLUR_SIZE; blurRow < BLUR_SIZE+1; ++blurRow) {
4.        for(int blurCol = -BLUR_SIZE; blurCol < BLUR_SIZE+1; ++blurCol) {

5.          int curRow = Row + blurRow;
6.          int curCol = Col + blurCol;
            // Verify we have a valid image pixel
7.          if(curRow > -1 && curRow < h && curCol > -1 && curCol < w) {
8.            pixVal += in[curRow * w + curCol];
9.            pixels++; // Keep track of number of pixels in the avg
            }
          }
        }
      // Write our new pixel value out
10    out[Row * w + Col] = (unsigned char)(pixVal / pixels);
    }
  }
```

Handling boundary conditions for pixels near the edges of the image



Content

- CUDA Thread Organization
- Mapping Threads to Multidimensional Data
  - Example – Image Blur
- <u>Synchronization and Transparent Scalability</u>
- Thread Scheduling and Latency Tolerance

Barrier Synchronization

- An API function call in CUDA
  - __syncthreads()

- All threads in the same block must reach the __syncthreads() before any can move on.

- Best used to coordinate tiled algorithms
  - To ensure that all elements of a tile are loaded
  - To ensure that all elements of a tile are consumed

Synchronization

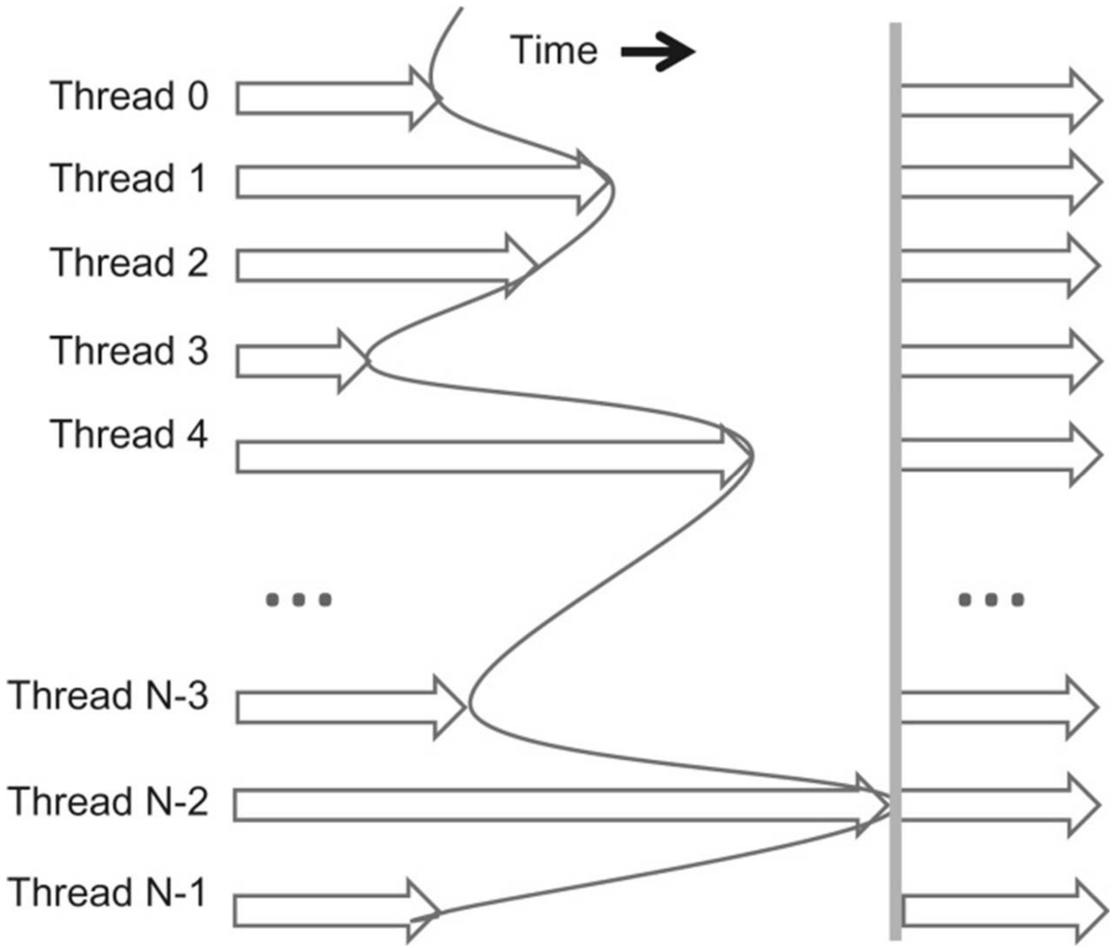
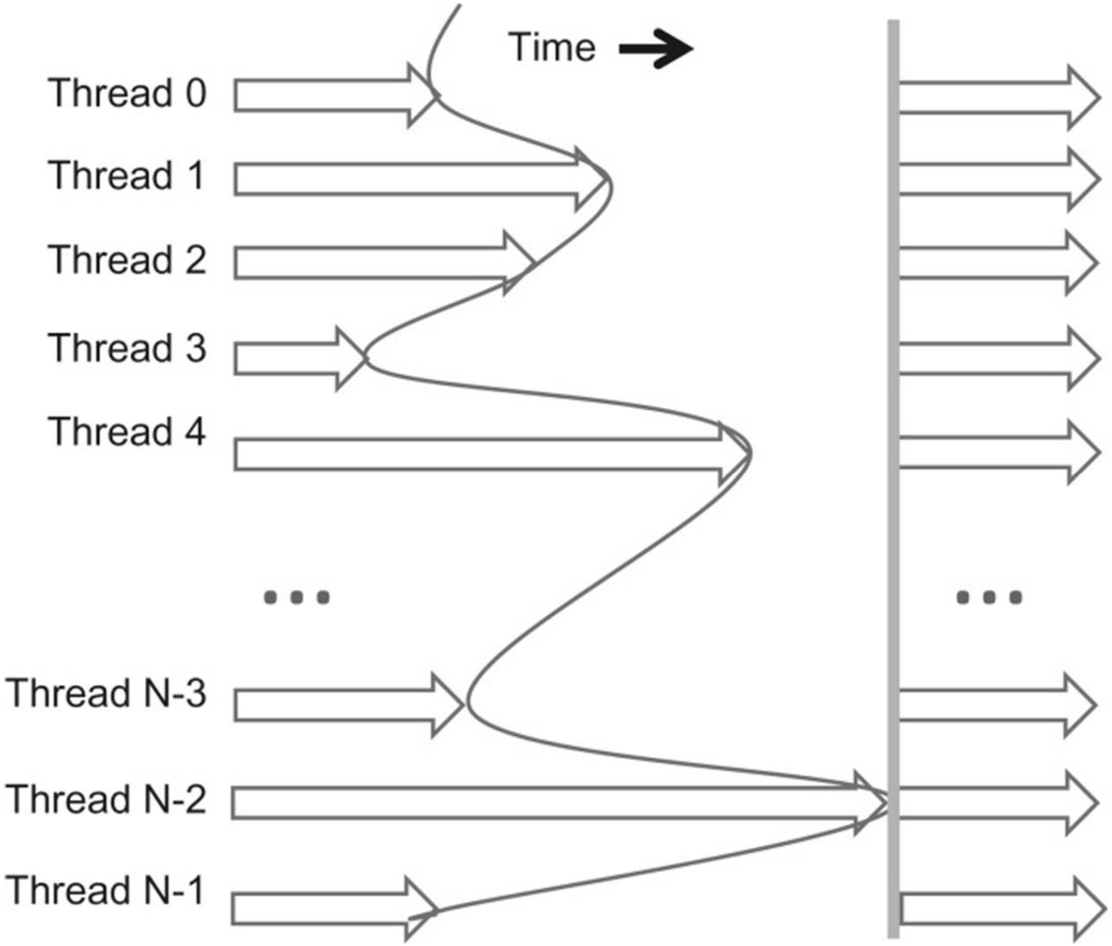


FIGURE 3.10: An example execution timing of barrier synchronization.

Synchronization




```
If
    ....
    __syncthreads()

else
    ....
    __syncthreads()
```

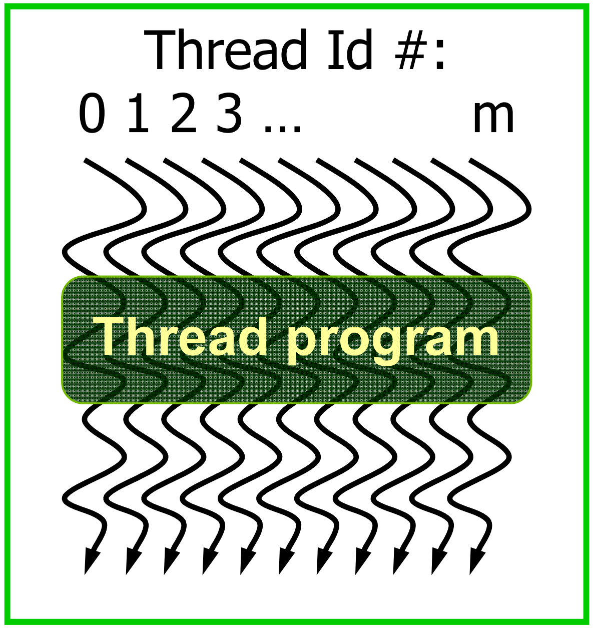
Dead lock

FIGURE 3.10: An example execution timing of barrier synchronization.

CUDA Thread Block (review)

- All threads in a block execute the same kernel program (SPMD)
- Programmer declares block:
  - Block size 1 to **1024** concurrent threads
  - Block shape 1D, 2D, or 3D
- Threads have thread index numbers within block
  - Kernel code uses thread index and block index to select work and address shared data
- Threads in the same block share data and synchronize while doing their share of the work
- Threads in different blocks cannot cooperate
  - Each block can execute in any order relative to other blocks!

CUDA Thread Block



Courtesy: John Nickolls, NVIDIA

Transparent Scalability

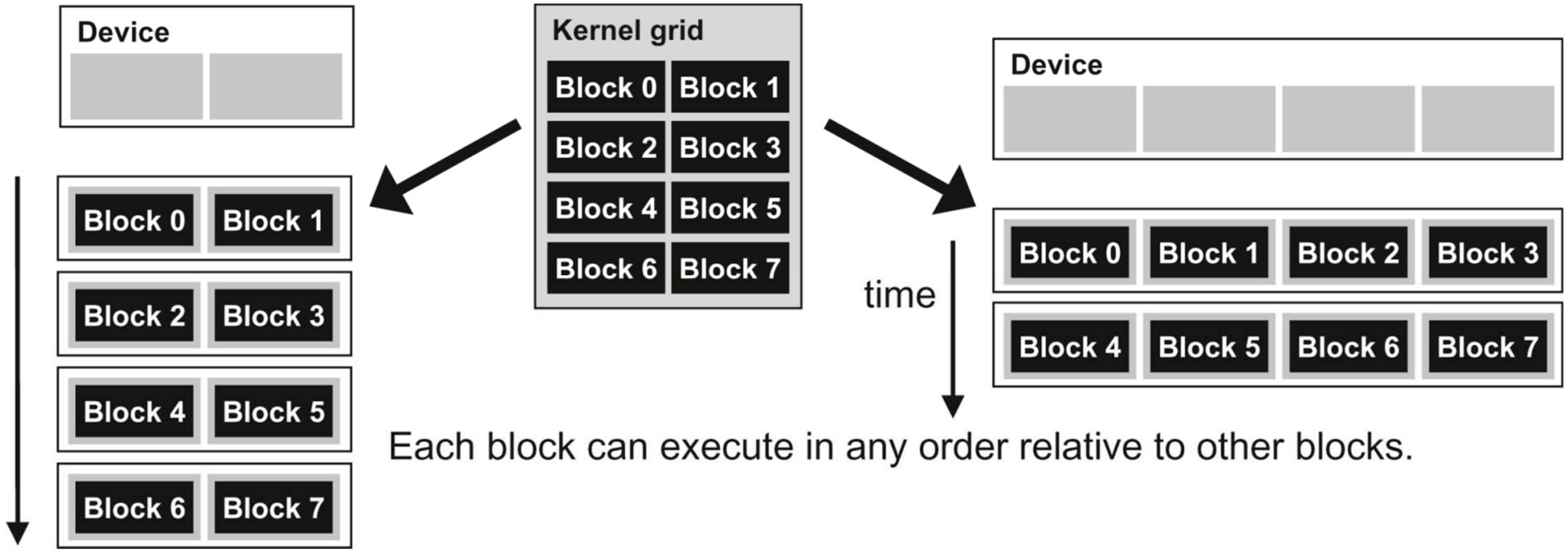
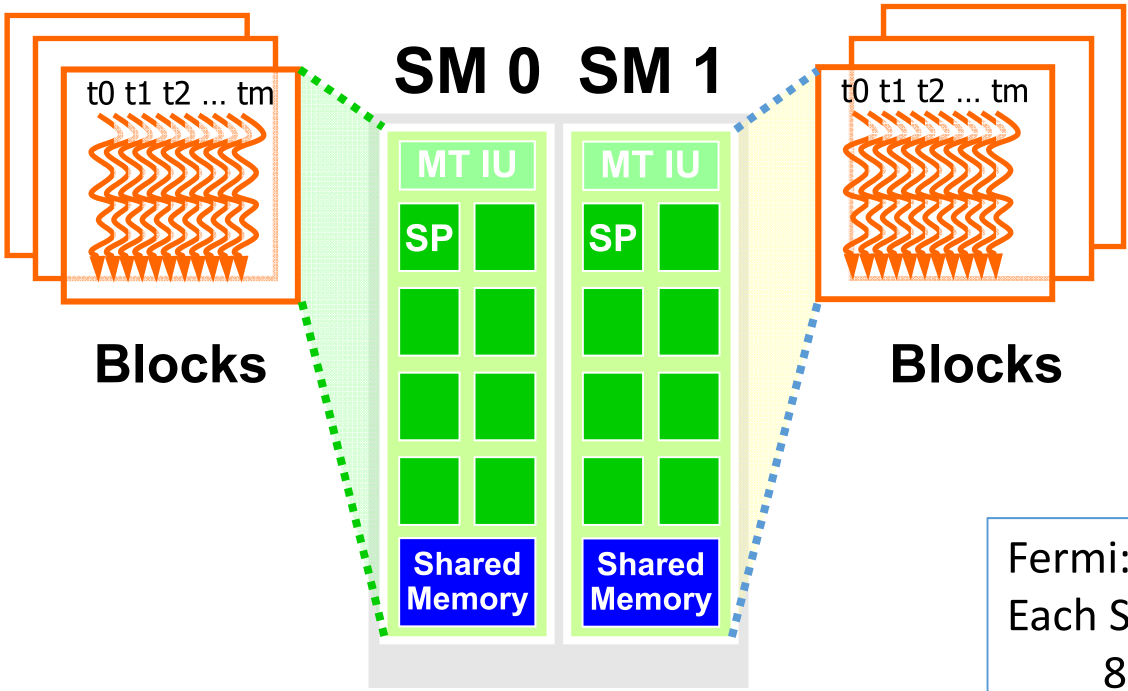


FIGURE 3.11: Lack of synchronization constraints between blocks enables transparent scalability for CUDA programs.

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

Thread Blocks Assignment



- Limitations:
  - Number of Streaming Multiprocessors
  - Number of Blocks in a SM
  - Number of Threads in a SM

Fermi:
Each SM has up to:
8 blocks
1536 threads
Total 30 SMs

Compute Capabilities are GPU Dependent

Table 1. A Comparison of Maxwell GM107 to Kepler GK107

GPU	GK107 (Kepler)	GM107 (Maxwell)
CUDA Cores	384	640
Base Clock	1058 MHz	1020 MHz
GPU Boost Clock	N/A	1085 MHz
GFLOP/s	812.5	1305.6
Compute Capability	3.0	5.0
Shared Memory / SM	16KB / 48 KB	64 KB
Register File Size / SM	256 KB	256 KB
Active Blocks / SM	16	32
Memory Clock	5000 MHz	5400 MHz
Memory Bandwidth	80 GB/s	86.4 GB/s
L2 Cache Size	256 KB	2048 KB
TDP	64W	60W
Transistors	1.3 Billion	1.87 Billion
Die Size	118 mm^2	148 mm^2
Manufactoring Process	28 nm	28 nm

Querying Device Properties

- How do find amount of resources available?
  - Number of SM
  - Number of Blocks in a SM
  - Number of Threads in a SM

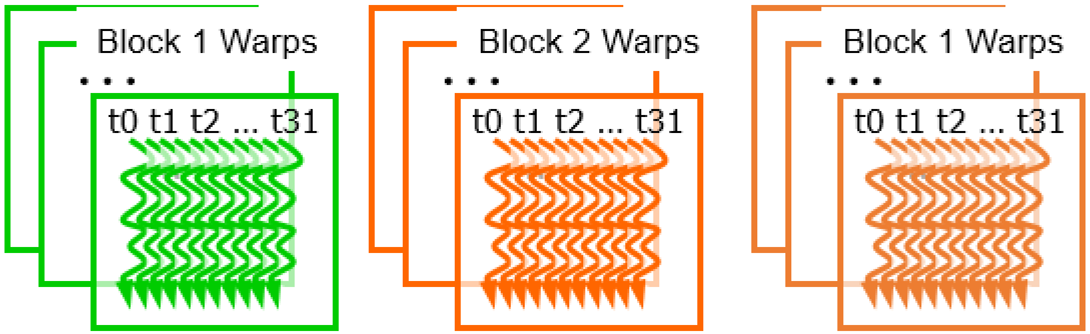
- CUDA Runtime API
  - cudaDeviceProp dev_prop;
  - cudaGetDeviceProperties(&dev_prop,i)

  - dev_prop.maxThreadsPerBlock
  - dev_prop.multiProcessorCount
  - ……
  - dev_prop.warpSize

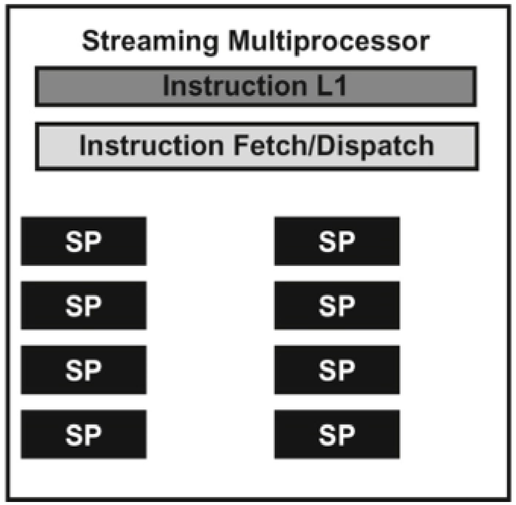
Content

- CUDA Thread Organization
- Mapping Threads to Multidimensional Data
  - Example – Image Blur
- Synchronization and Transparent Scalability
- <u>Thread Scheduling and Latency Tolerance</u>

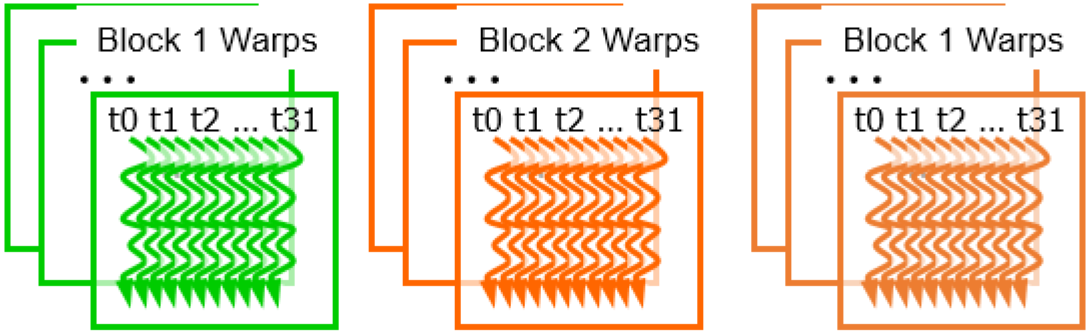
Thread Scheduling (1/2)



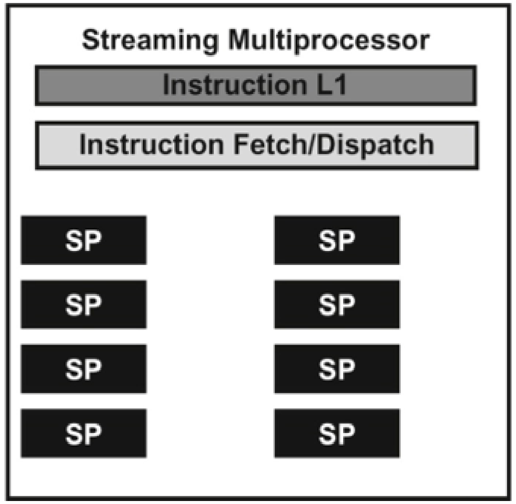
- Each block is executed as 32-thread warps
  - An implementation decision, not part of the CUDA programming model
  - Warps are scheduling units in SM
  - An SM is designed to execute all threads in a warp following the Single Instruction, Multiple Data (SIMD) model



Thread Scheduling (1/2)

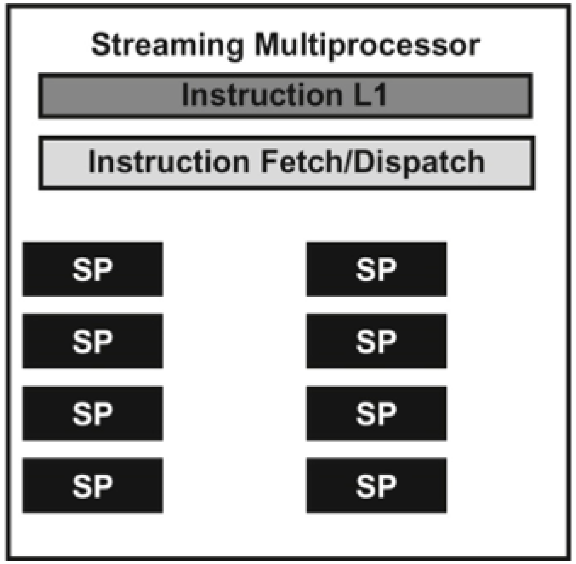


- If 3 blocks are assigned to an SM and each block has 256 threads,
- how many warps are there in an SM?
  - Each block is divided into 256/32 = 8 warps
  - 8 warps/blk * 3 blks = 24 warps



Thread Scheduling (1/2)

- Streaming Processors (SPs)
  - hardware that actually execute instructions.
- In general, there are fewer SPs than the threads assigned to each SM

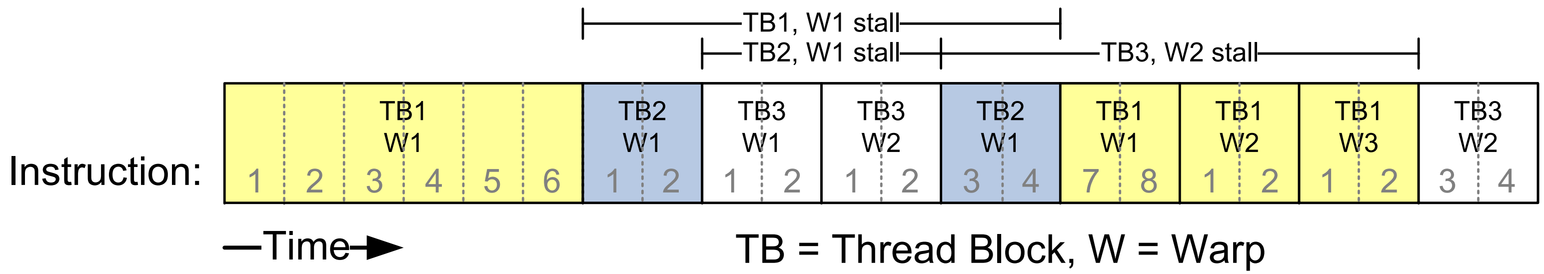


- Actually, hardware can only execute small subset of warps.
- So, why we need to have so many warps in an SM if it can only execute a small subset of them at any instant?

Thread Scheduling (2/2)

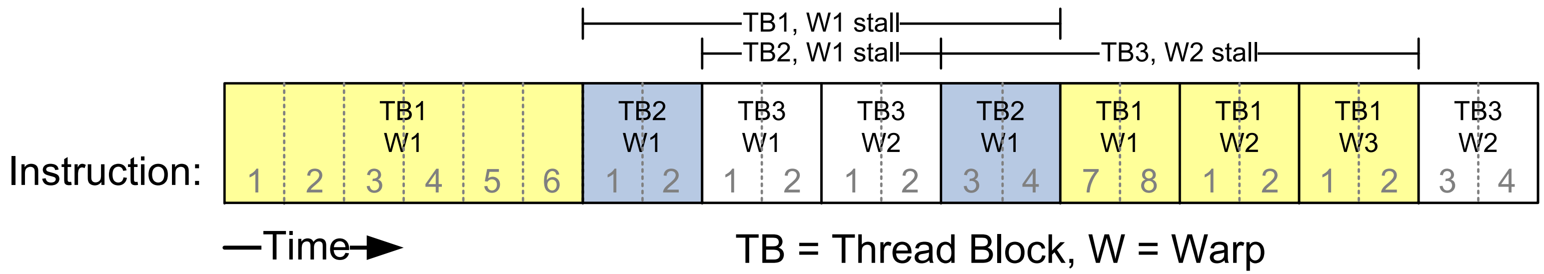
- Latency Tolerance

- SM implements zero-overhead warp scheduling
  - Warps whose next instruction has its operands ready for consumption are eligible for execution;
  - avoids introducing idle or wasted time into the execution timeline.



Thread Scheduling (2/2)

- With warp scheduling:
  - the long waiting time of warp instructions is “hidden” by executing instructions from other warps;
  - Much less chip area to cache memories and branch prediction mechanisms;
  - More space for floating-point execution resources.



Summary

- CUDA Thread Organization
- Mapping Threads to Multidimensional Data
  - Example – Image Blur
- Synchronization and Transparent Scalability
- Thread Scheduling and Latency Tolerance