CUDA Thread 2

Prof. Seokin Hong

Agenda

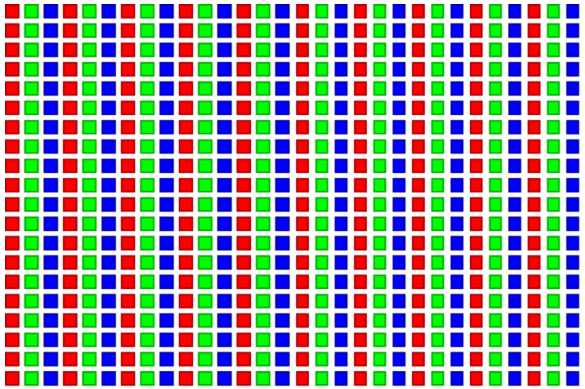
- What is Thread?
- CUDA Thread Organization
- Extended VecAdd.cu
- Maxtrix Addition Kernel
- Mapping Threads to Multidimensional Data
- Synchronization and Transparent Scalability
- Resource Assignment
- Querying Device Properties
- Thread Scheduling and Latency Tolerance

Mapping Threads to Multidimensional Data

"Color-to-Grayscale Image Processing Example"

Conversion of a color image to grey–scale image

- **RGB Color Image Representation**
  - Each pixel in an image is an RGB value
  - The format of an image's row is
    (r g b) (r g b) … (r g b)

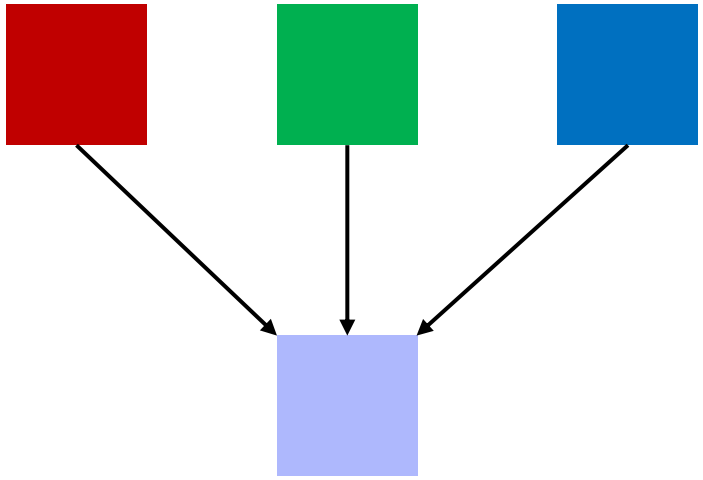


- A grayscale digital image is an image in which the value of each pixel carries only intensity information

Color Calculating Formula

- For each pixel (r g b) at (I, J) do:

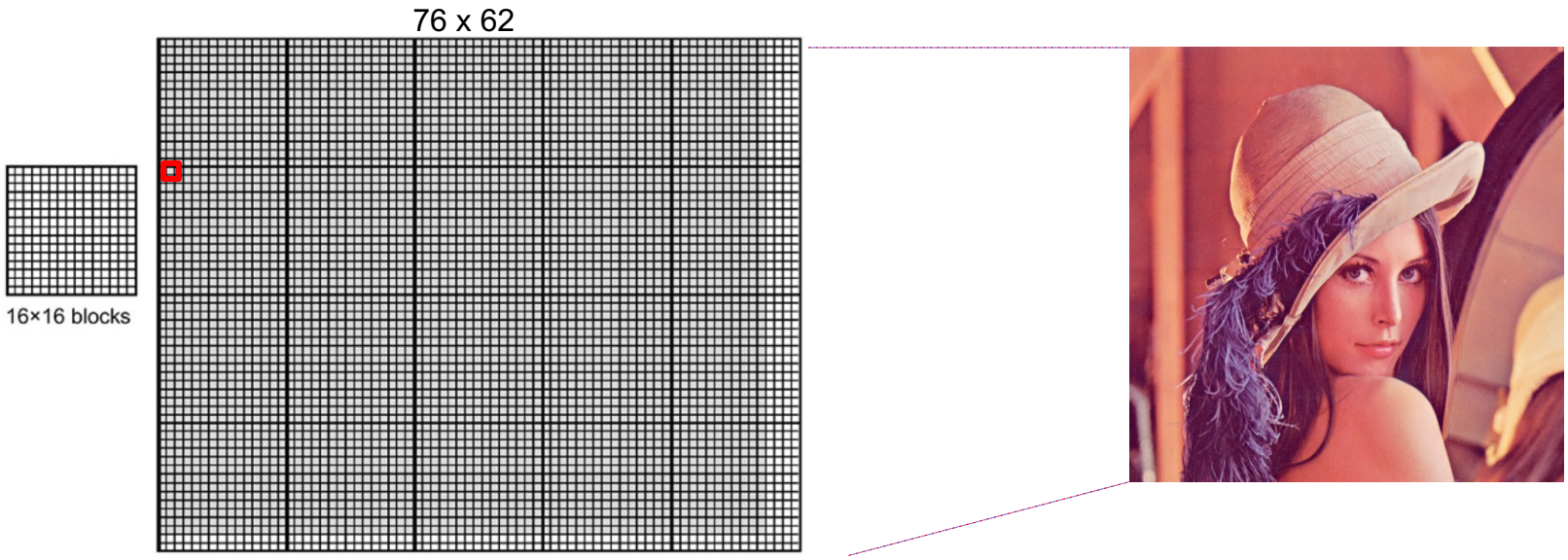
$$grayPixel[I,J] = 0.21*r + 0.71*g + 0.07*b$$



Determining Thread Organization

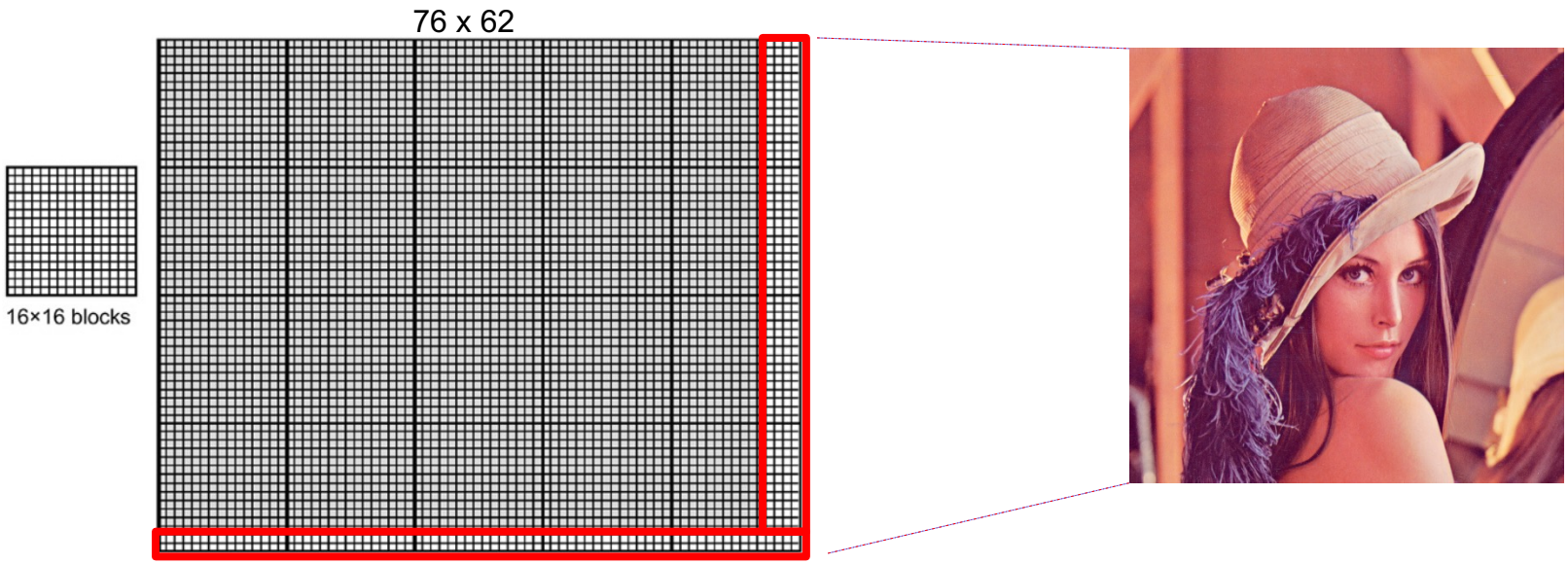
- The choice of thread organization is **based on the nature of data**
  - Pictures is 2D, so
    - Using 2D grid that consists of 2D blocks is often convenient for processing a picture
  - Example: 76 x 62 picture
    - Use a 16x16 block → 16 threads in the x direction, 16 threads in the y direction
    - Use a 5 x 4 grid → 5 blocks in the x direction, 4 blocks in the y direction
    - Pixel that is processed by thread(0,0) of block(1,0):

$$P_{\text{blockIdx.y*blockDim.y+threadIdx.y,blockIdx.x*blockDim.x+threadIdx.x}} = P_{1*16+0,0*16+0} = P_{16,0}$$

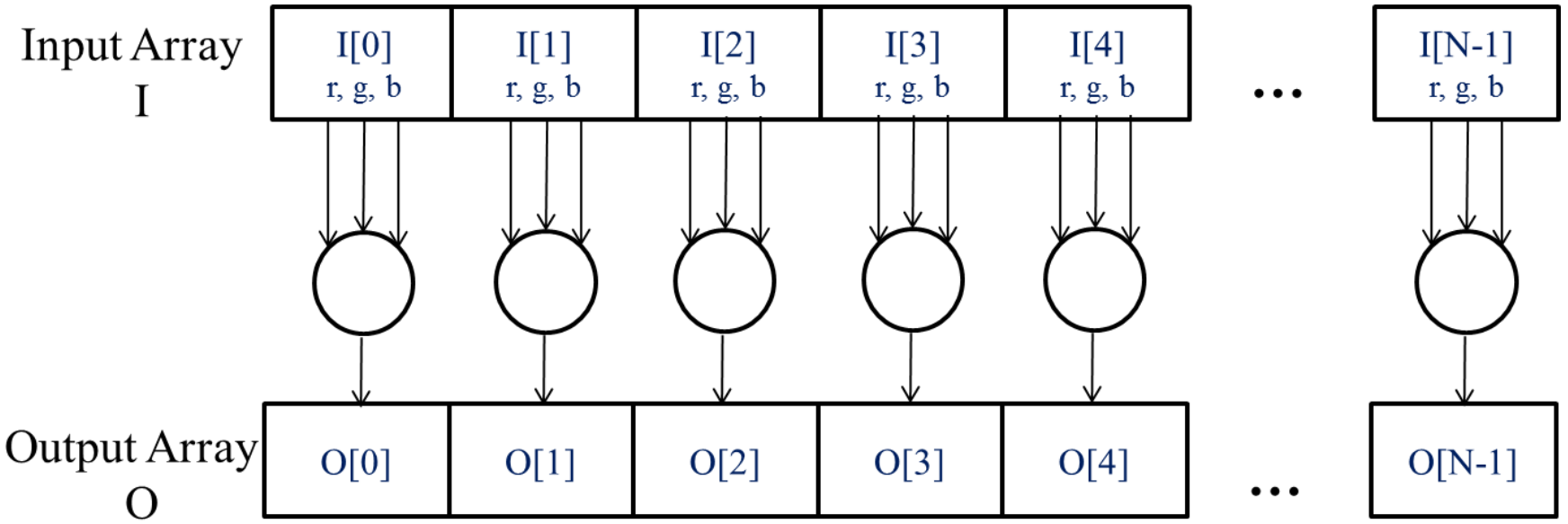


Determining Thread Organization (cont'd)

- Example: 76 x 62 picture (cont'd)
  - Using a 16x16 block and a 5 x 4 grid will have **4 extra threads** in the x direction and **2 extra threads** in the y direction
  - So, in **kernel** function, we should **test** whether the thread indexes threadIdx.x and threadIdx.y fall within the **valid range of pixels**

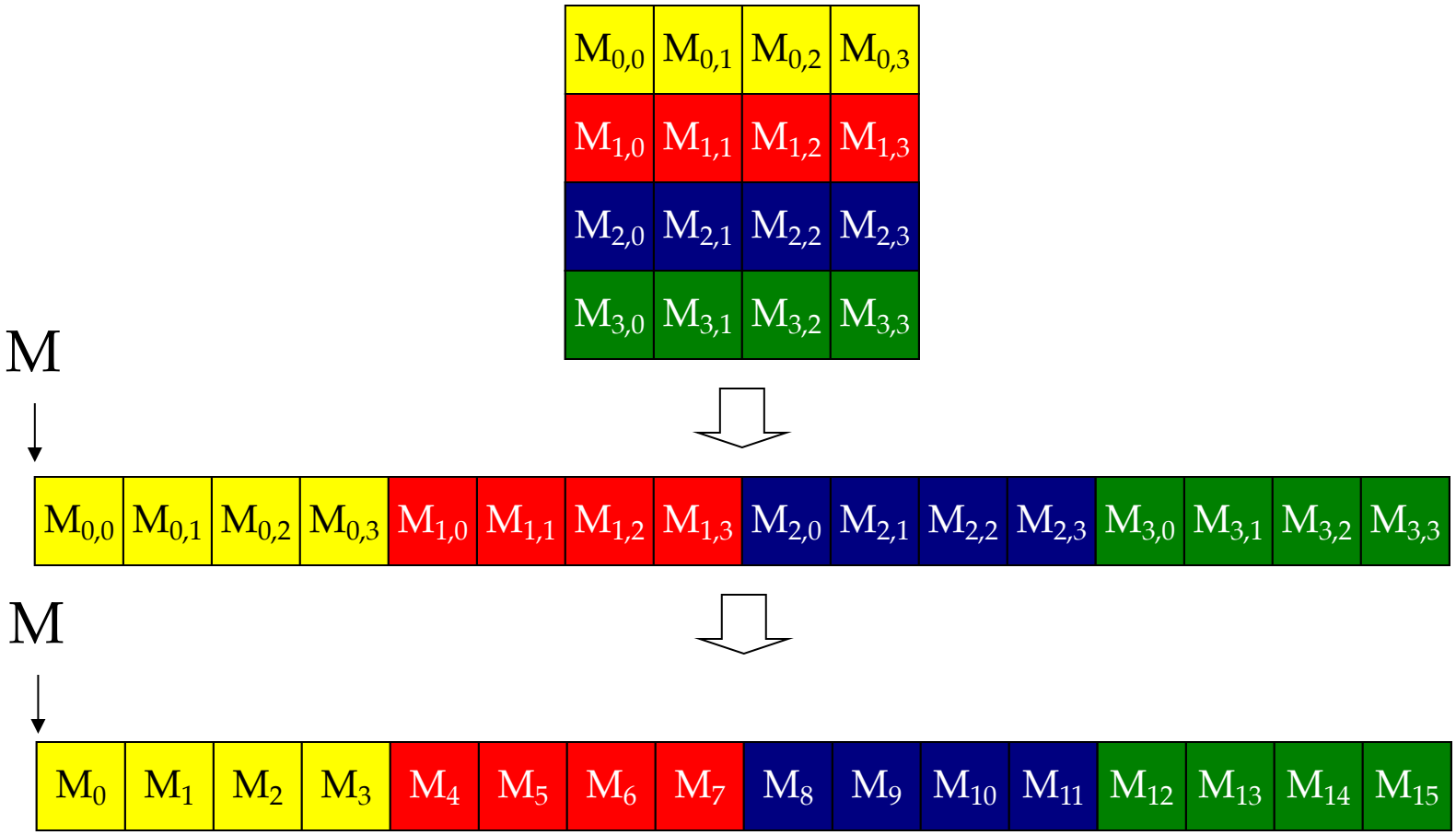


The pixels can be calculated independently of each other (review)



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

- 2D array is linearized of flattened into 1D array



$M_{2,1}$ → Row*Width+Col = 2*4+1 = 9

RGB to Grayscale Conversion: Kernel

rgb_to_grey.cu

```
#define CHANNELS 3 // we have 3 channels corresponding to RGB
// The input image is encoded as unsigned characters [0, 255]
__global__ void colorConvert(unsigned char * grayImage,
                                  unsigned char * rgbImage,
                              int width, int height) {
 int x = threadIdx.x + blockIdx.x * blockDim.x;
 int y = threadIdx.y + blockIdx.y * blockDim.y;

 //check whether the threads with both Row and Col are within range
 if (x < width && y < height) {
    // get 1D coordinate for the grayscale image
    int grayOffset = y*width + x;

    // one can think of the RGB image having
    // CHANNEL times columns than the gray scale image
    int rgbOffset = grayOffset*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;
 }
}
```

RGB to Grayscale Conversion: Kernel (cont'd)

rgb_to_grey.cu

```
#define CHANNELS 3 // we have 3 channels corresponding to RGB
// The input image is encoded as unsigned characters [0, 255]
__global__ void colorConvert(unsigned char * grayImage,
                             unsigned char * rgbImage,
                             int width, int height) {
 int x = threadIdx.x + blockIdx.x * blockDim.x;
 int y = threadIdx.y + blockIdx.y * blockDim.y;

 //check whether the threads with both Row and Col are within range
 if (x < width && y < height) {
    // get 1D coordinate for the grayscale image
    int grayOffset = y*width + x;

    // one can think of the RGB image having
    // CHANNEL times columns than the gray scale image
    int rgbOffset = grayOffset*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;
 }
}
```

RGB to Grayscale Conversion: Kernel (cont'd)

rgb_to_grey.cu

```
#define CHANNELS 3 // we have 3 channels corresponding to RGB
// The input image is encoded as unsigned characters [0, 255]
__global__ void colorConvert(unsigned char * grayImage,
                                unsigned char * rgbImage,
                             int width, int height) {
 int x = threadIdx.x + blockIdx.x * blockDim.x;
 int y = threadIdx.y + blockIdx.y * blockDim.y;

 //check whether the threads with both Row and Col are within range
 if (x < width && y < height) {
    // get 1D coordinate for the grayscale image
    int grayOffset = y*width + x;

    // one can think of the RGB image having
    // CHANNEL times columns than the gray scale image
    int rgbOffset = grayOffset*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;
 }
}
```

RGB to Grayscale Conversion: Host code

rgb_to_grey.cu

```
int main(int argc, char **argv)
{
    std::string rgb_filename, grey_filename;
    unsigned int pixels=0;
    //Check for the input file and output file name
    …….
    //load image into an array and retrieve number of pixels
     ……
    //allocate host memory for grey image
    grey_h= (unsigned char *)malloc(sizeof(unsigned char)*pixels);

    //allocate and initialize memory on device
    CUDA_CHECK(cudaMalloc(&rgb_d, sizeof(unsigned char) * pixels * CHANNELS));
    CUDA_CHECK(cudaMalloc(&grey_d, sizeof(unsigned char) * pixels));
    CUDA_CHECK(cudaMemset(grey_d, 0, sizeof(unsigned char) * pixels));

    //copy rgb image from host to device
    CUDA_CHECK(cudaMemcpy(rgb_d, rgb_h, sizeof(unsigned char)*pixels*CHANNELS, cudaMemcpyHostToDevice));

    //define block and grid dimensions
    const dim3 dimGrid(ceil(cols/16), ceil(rows/16),1);
    const dim3 dimBlock(16,16);

    //execute cuda kernel
    colorConvert<<<dimGrid, dimBlock>>>(grey_d, rgb_d, rows, cols);
    CUDA_CHECK( cudaPeekAtLastError() );

    //copy computed gray image from device to host
    CUDA_CHECK(cudaMemcpy(grey_h, grey_d, sizeof(unsigned char) * pixels, cudaMemcpyDeviceToHost));

    //store the grey image
    ……

    //free memory
    cudaFree(rgb_d);
    cudaFree(rgb_h);
}
```

Complete version is available in /home/shared/lecture6/rgb_to_grey

RGB to Grayscale Conversion: compile and run

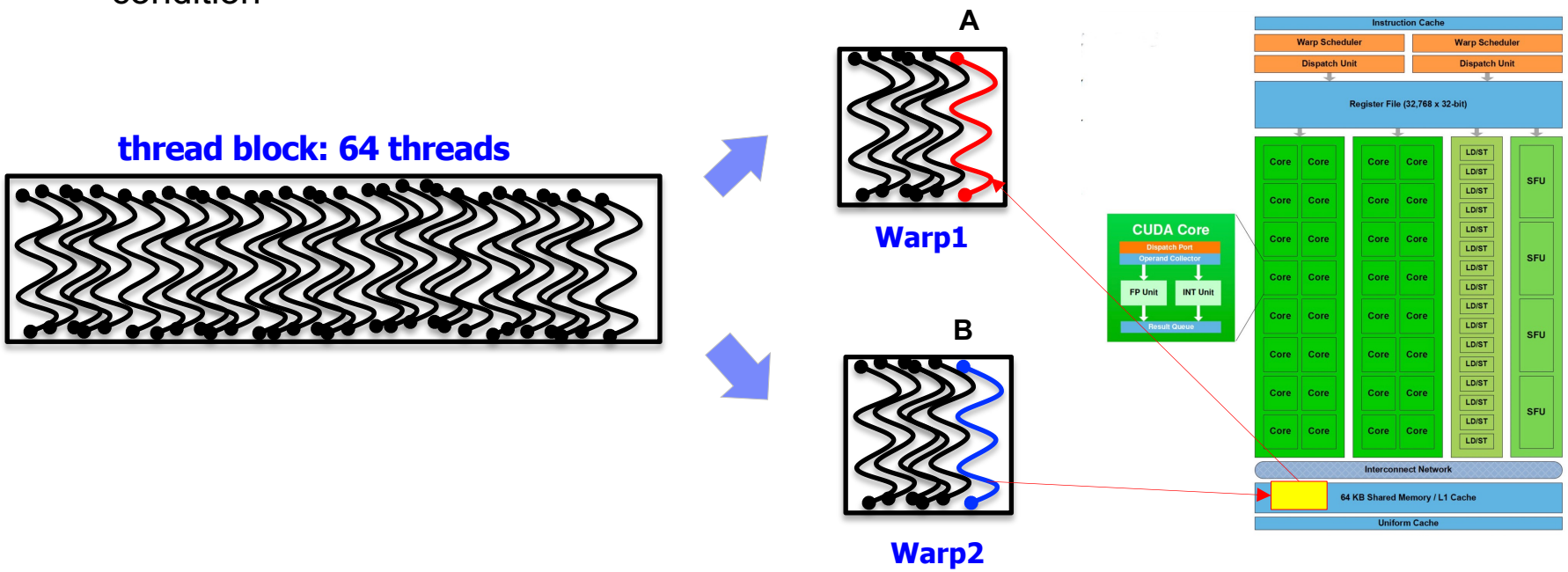
```
$ export LD_LIBRARY_PATH=/usr/local/lib
$ nvcc rgb_to_grey.cu  -lopencv_imgcodecs -lopencv_core -o rgb_to_grey
$ ./rgb_to_grey ./rgb_image.png grey_image.png
```

Copy “grey_image.png” to your PC with a FTP program

Synchronization and Transparent Scalability

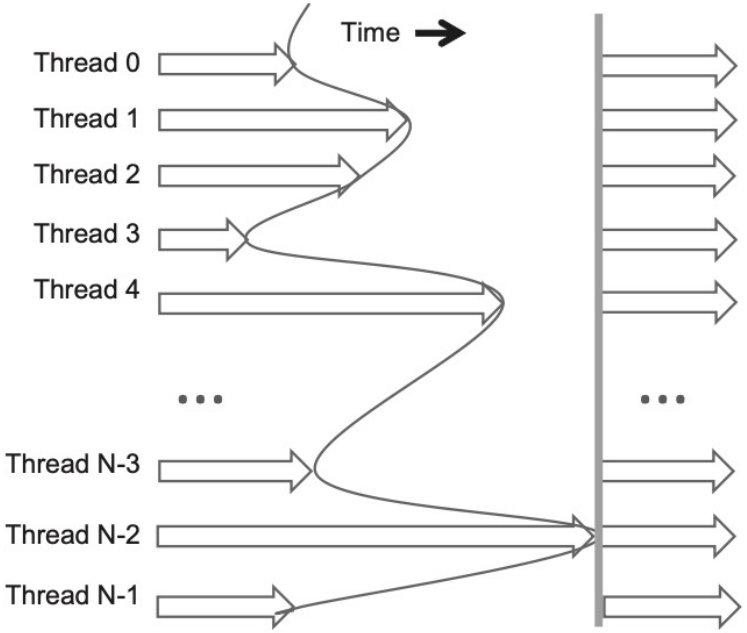
Synchronization

- When sharing data between threads,
  - Need to careful to avoid **race conditions**
  - Because threads in a block run logically in parallel, no all threads can execute physically at the same time
  - **Race condition** example
    - Two threads A and B share data, but they are in two different warps
    - Thread A want to read B's data from shared memory
    - If B has not finished writing its data to shared memory before A tries to read it → Race condition



Synchronization (Cont'd)

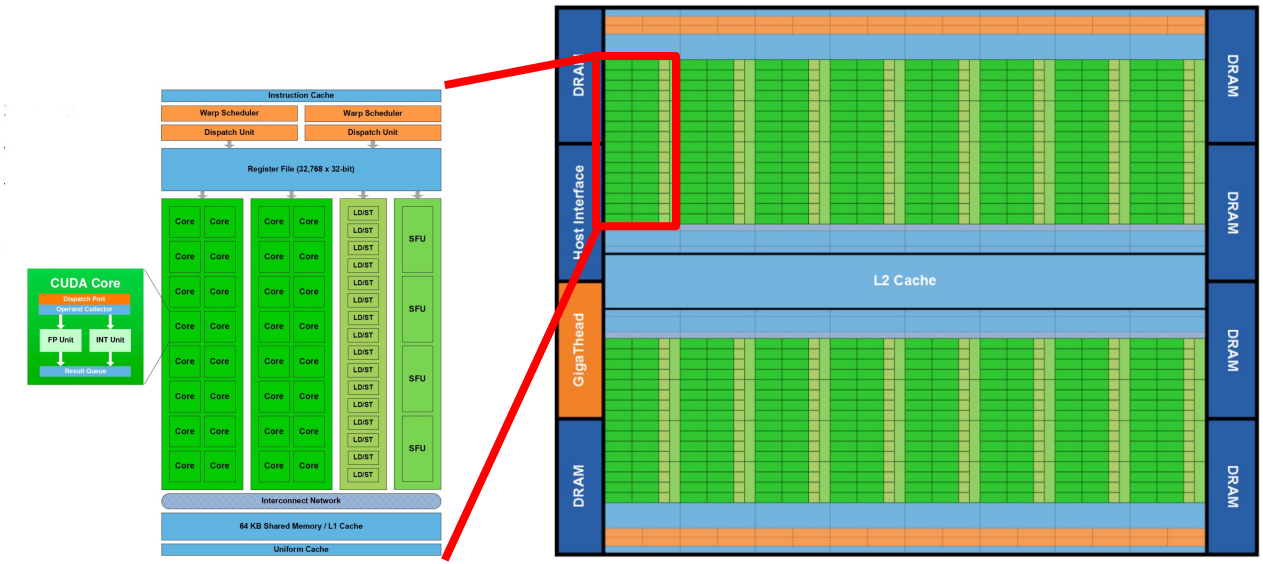
- CUDA allows threads in the same block to coordinate their activities by using __syncthreads()
- When a thread calls __syncthreads(), it will be held at the calling location until every thread in the block reaches the location
- This process ensures that all threads in a block have completed a phase of their execution of the kernel before any of them can proceed to the next phase



Transparent Scalability

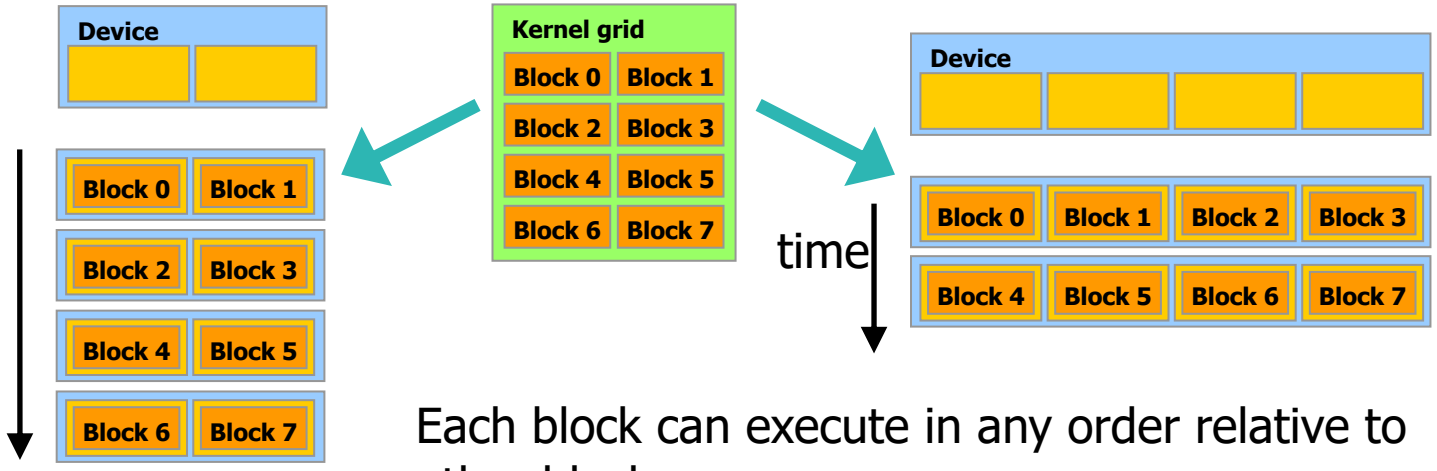
- Scalability Issue:
  - SM is the fundamental processing unit.
  - But, a CUDA device may have various number of SM's
    - a single SM for low-tier tablets / smart phones
    - up to 256 or more SM's for high-end GPUs

- CUDA's solution
  - Block as a unit for SM processing



Transparent Scalability

- Hardware is free to assign blocks to any processor at any time
- Transparent scalability : the ability to execute the same application program on hardware with a different number of execution resources
  - a kernel scales across any number of parallel processors
  - → reduce developer's burden and improves the usability of applications.



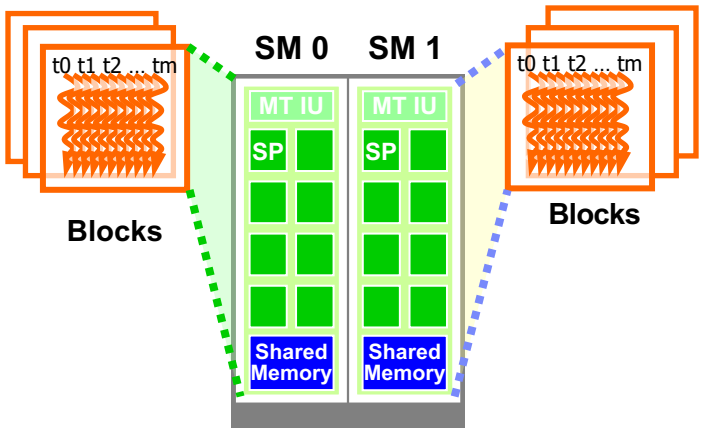
Each block can execute in any order relative to other blocks.

Resource Assignment

Executing Thread Blocks

- Once a kernel is launched, the CUDA runtime system generates a **grid** of threads
- Threads are assigned to Streaming Multiprocessors in block granularity
  - A **block** is composed of threads which can communicate within their own block
  - **Context switching is very fast** because registers and shared memory are allocated for a block as long as that block is active

- GPU device has **a limit on the number of blocks and threads** that can be assigned to each SM in a given time
  - Because each SM has the limited number of hardware resources (registers) to maintain thread/block id or thread execution state
  - Up to 32 blocks to each SM as resource allows
  - Volta SM can take up to 2048 threads
    - Could be 256 (threads/block) * 8 blocks
    - Or 512 (threads/block) * 4 blocks, etc.



Executing Thread Blocks (Cont'd)

- Threads run concurrently
  - SM assign/maintains thread/block id#
  - SM manages/schedules thread execution

- Most grids contain more blocks than the maximum number of blocks that can be assigned to each SM
  - →**CUDA runtime** maintains a list of blocks that need to execute and assigns new blocks to SMs as previously assigned bocks complete execution
  - →But, need to launch a kernel with an optimized thread organization to maximize resource utilization (Occupancy)
    - A small block size will limit the total number of threads → Low occupancy

Querying Device Properties

How to find out the amount of resources?

- Application needs to query the available resources and capabilities of underlying GPU device to find
  - the number of available SMs in a GPU device
  - the maximum number of blocks that can be assigned to each SM
  - the maximum number of threads that can be assigned to each SM

- CUDA Runtime API
  - **cudaGetDeviceCount()**: return the number of CUDA devices (GPU)
  - **cudaGetDeviceProperties()**: return the properties of the device

```
int dev_count;
cudaGetDeviceCount(&dev_count);

cudaDeviceProp dev_prop;
for (int i=0; i< dev_count; i++)
{
    cudaGetDeviceProperties(&dev_prop, i);
    //do something with device's properties
}
```

*All fields of **cudaDeviceProp** data type is available in the appendix

How to find out the amount of resources?

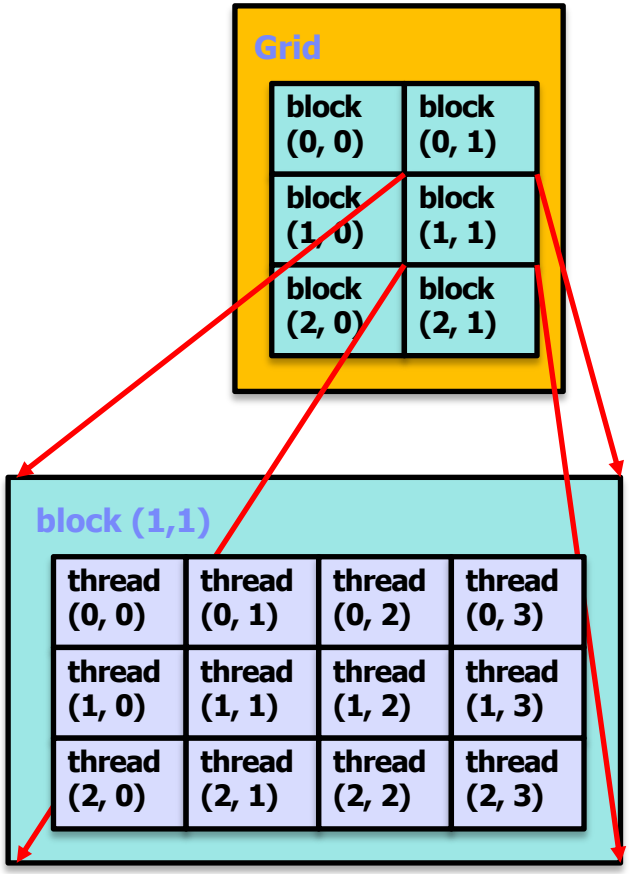
- cudaDeviceProp
  - **warpSize :** the number of threads in a warp
  - totalGlobalMem : the amount of global memomy on the device in bytes
  - **maxThreadsPerBlock** : maximum number of threads allowed in a block
  - **multiProcessorCount** : the number of SMs in the device
  - **clockRate** : the clock frequency of the device
  - **maxThreadsDim[3]** : maximum number of threads allowed along each dimension of a block
    - maxThreadsDim[0] for x
    - maxThreadsDim[1] for y
    - maxThreadsDim[2] for z
  - MaxGridSize[3] : number of blocks allowed along each dimension of a grid
    - MaxGridSize[0] for x
    - MaxGridSize[1] for y
    - MaxGridSize[2] for z

*All fields of **cudaDeviceProp** data type is available in the appendix

Device Query Example: GTX 1070

- 32 threads per warp
- 2048 threads per SM
  - 64 warps per SM
- 1024 threads per block
  - 32 warps per block
- 32 blocks per SM

- For 8x8 blocks, we have 64 threads per Block
  - each SM has 2048/64 = 32 blocks
- For 16x16 blocks, we have 256 threads per Block
  - each SM has 2048/256 = 8 blocks
- For 32x32 blocks, we have 1024 threads per Block.
  - each SM has 2048/1024 = 2 blocks



Slide Credit: Slides are modified from Prof Baek's slides

Device Query Example : GT 520M

- 32 threads per warp
- 1536 threads per SM
  - 48 warps per SM
- 1024 threads per block
  - 32 warps per block
- 8 blocks per SM

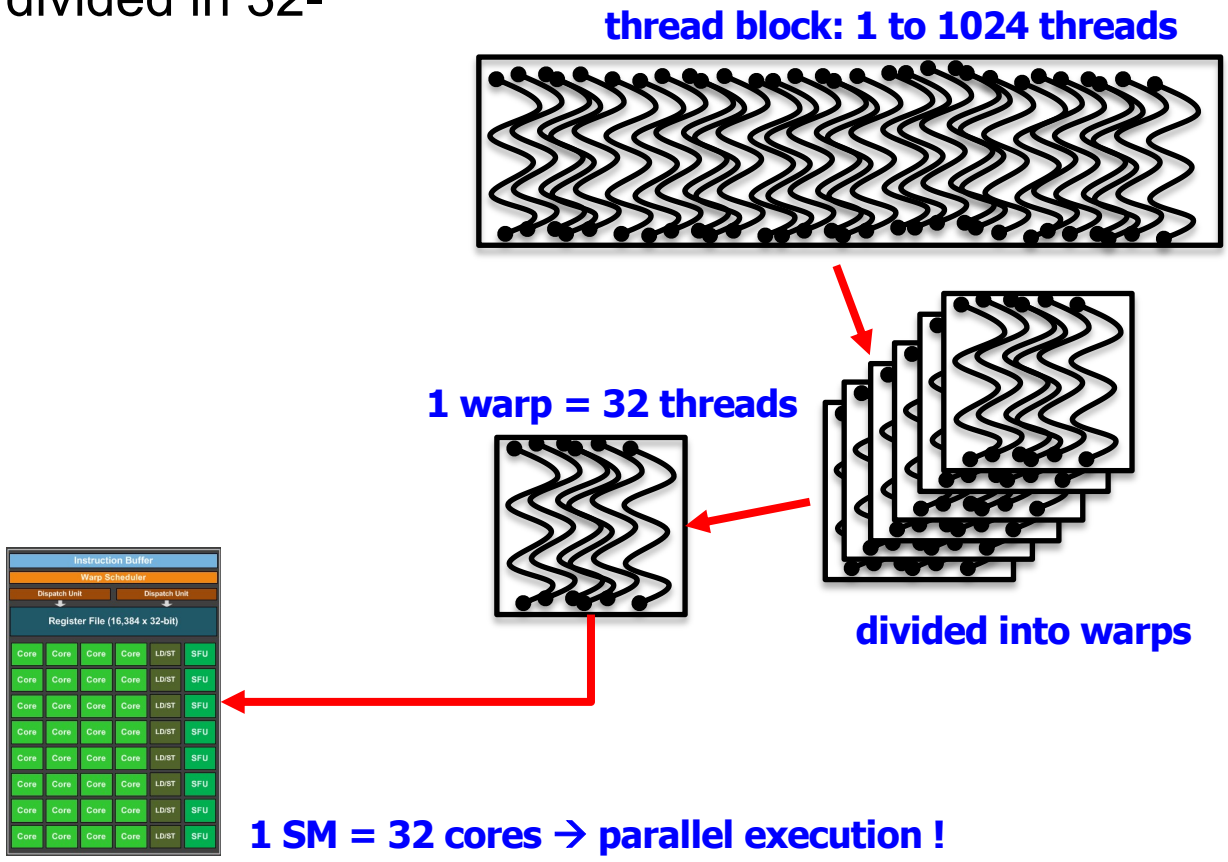
- For 8x8 blocks, we have 64 threads per Block
  - each SM has 1536/64 = 32 blocks
  - but, only 8 blocks per SM
  - finally, 8x64 = 512 threads per SM
- For 16x16 blocks, we have 256 threads per Block
  - each SM has 1536/256 = 6 blocks
  - finally, 256x6 = 1536 threads per SM
- For 32x32 blocks, we have 1024 threads per Block.
  - each SM has 1536/1024 = 1 block
    - finally, 1024 threads per SM

- What is your best choice ?

Thread Scheduling and Latency Tolerance

Review: Thread Scheduling/Execution

- Threads run concurrently
  - SM assigns/maintains thread id #s
  - SM manages/schedules thread execution
- Each Thread Blocks is divided in 32-thread Warps



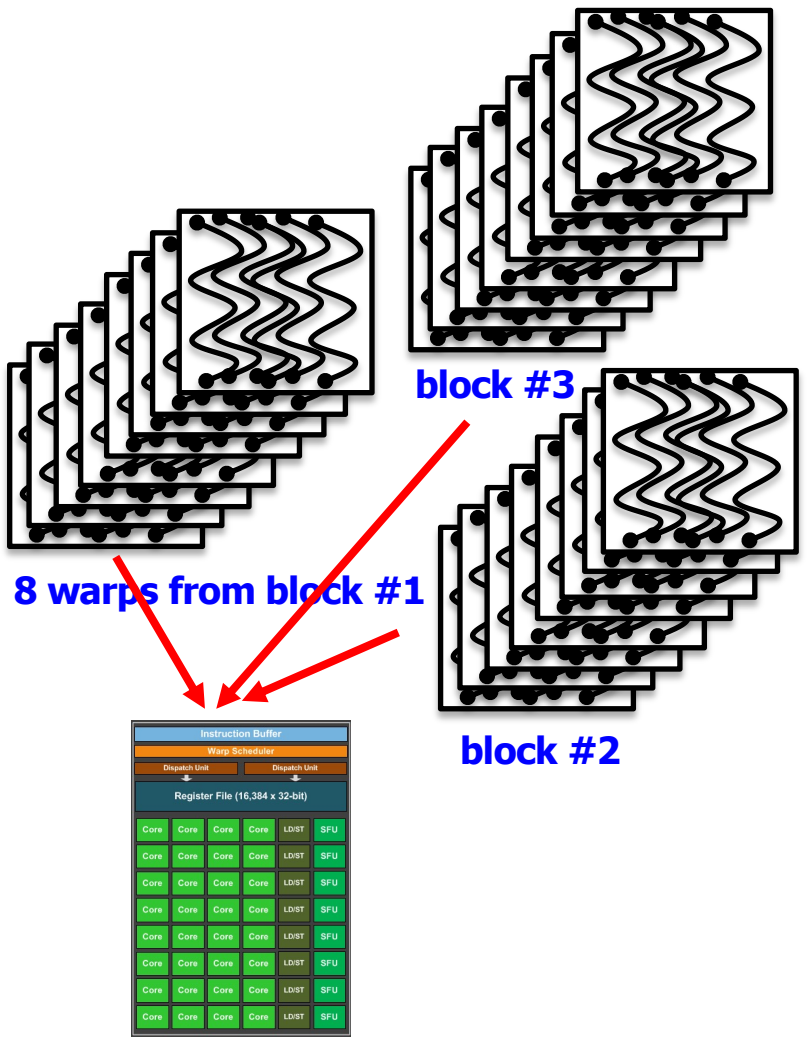
Slide credit : Prof. Baek

Review: Thread Scheduling/Execution

- Warps are scheduling units in SM

- A scenario
  - 3 blocks to an SM
  - each block has 256 threads

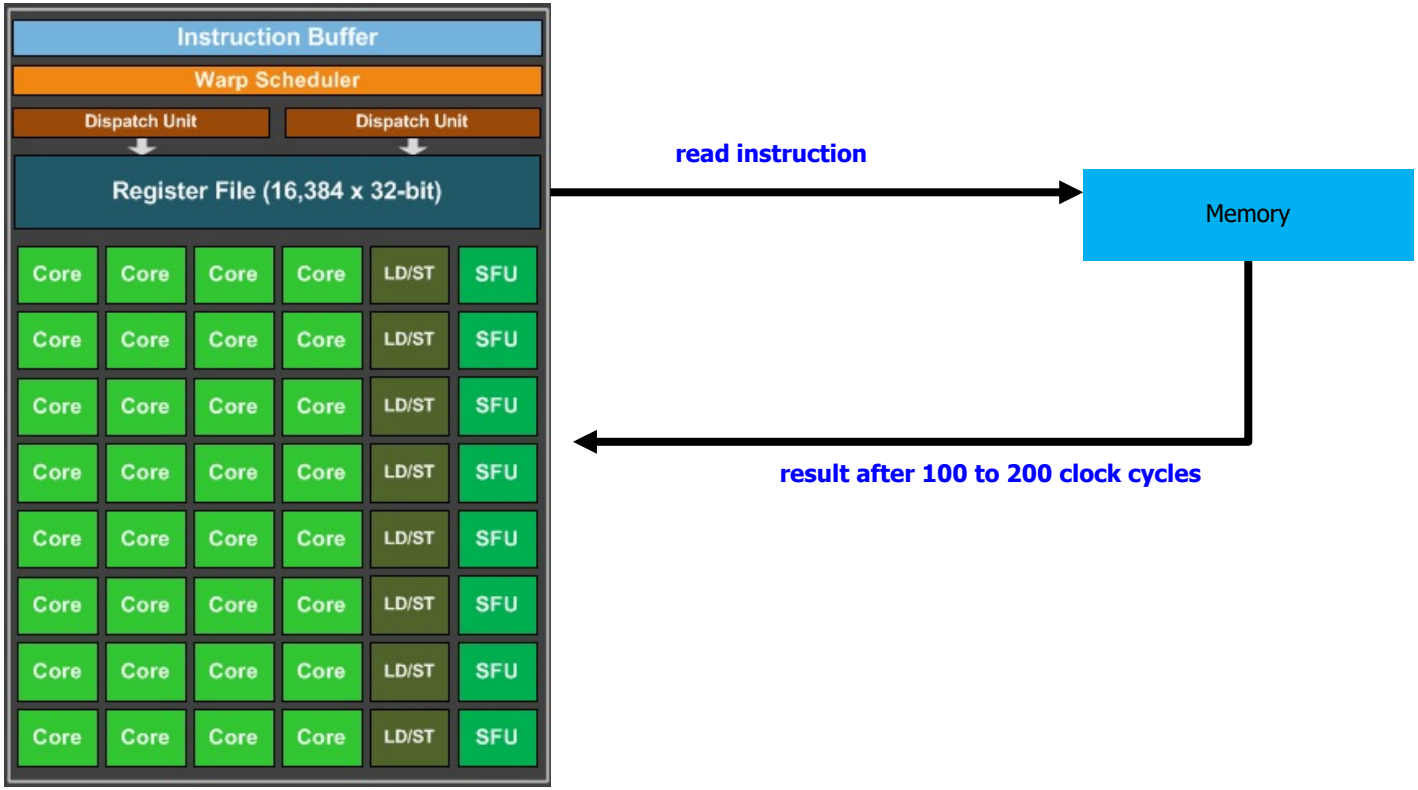
- how many warps?
  - each block has 256 / 32 = 8 warps
  - SM has 3 * 8 = 24 warps
  - At any point in time,
    only one of the 24 Warps will be selected for instruction fetch and execution.



Slide credit : Prof. Baek

Review: SM Warp Scheduling

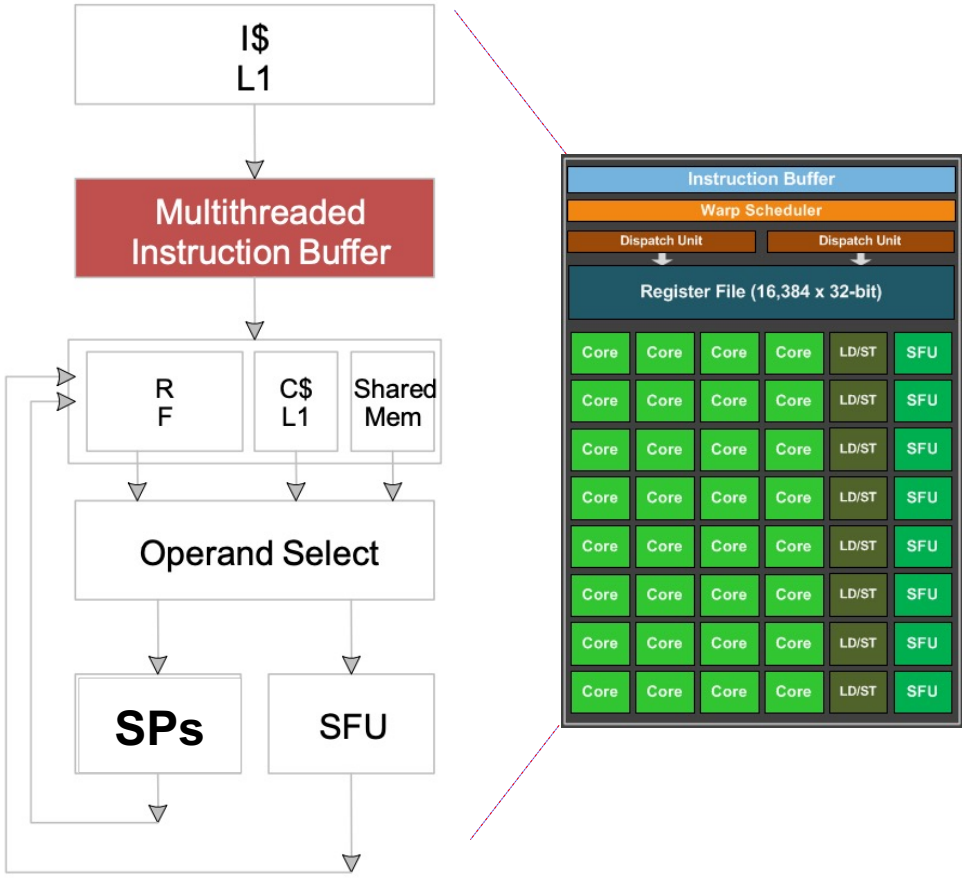
- All threads in a Warp execute the same instruction when selected
  - only one control logic for an SM
- **memory access** → latency problem → scheduling required !



Slide credit : Prof. Baek

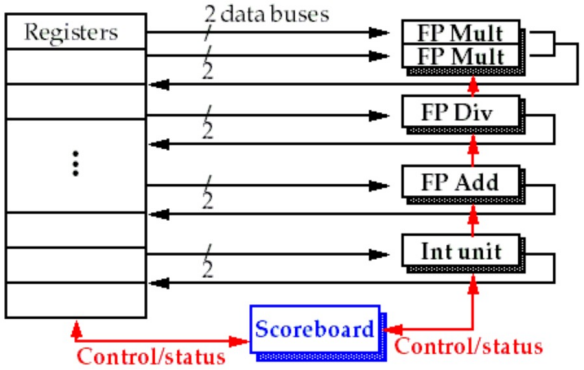
SM Warp Scheduling Details

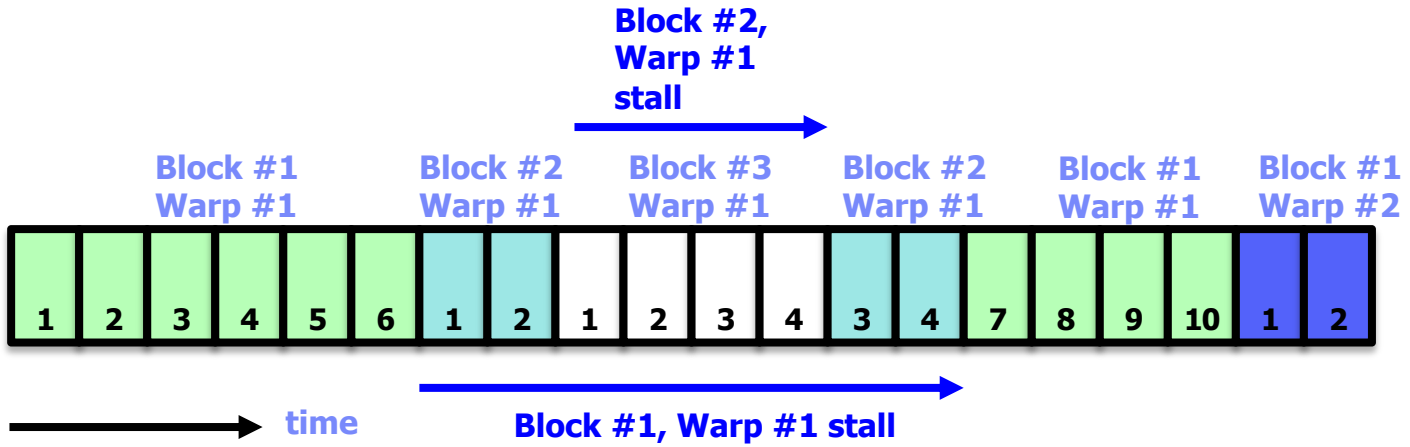
- **Instruction Buffer**
  - Fetch one warp instruction/cycle
    - from instruction L1 cache
    - into any instruction buffer slot
  - Issue one “ready-to-go” warp instruction/cycle from any instruction buffer slot
    - Issue selection based on round robin/age of warp
    - Use operand scoreboarding to prevent hazards



SM Warp Scheduling Details (cont'd)

- **Scoreboarding**
  - Determine if a thread is ready to execute
  - All register operands of all instructions in the Instruction Buffer are scoreboarded
  - warp status becomes ready after the needed values are deposited





Next?

- CUDA Memory Model1

Appendix

CUDA Device Properties

DEVICE PROPERTY	DESCRIPTION
`char name[256];`	An ASCII string identifying the device (e.g., `"GeForce GTX 280"`)
`size_t totalGlobalMem`	The amount of global memory on the device in bytes
`size_t sharedMemPerBlock`	The maximum amount of shared memory a single block may use in bytes
`int regsPerBlock`	The number of 32-bit registers available per block
`int warpSize`	The number of threads in a warp
`size_t memPitch`	The maximum pitch allowed for memory copies in bytes

DEVICE PROPERTY	DESCRIPTION
`int maxTexture2D[2]`	The maximum dimensions supported for 2D textures
`int maxTexture3D[3]`	The maximum dimensions supported for 3D textures
`int maxTexture2DArray[3]`	The maximum dimensions supported for 2D texture arrays
`int concurrentKernels`	A boolean value representing whether the device supports executing multiple kernels within the same context simultaneously

DEVICE PROPERTY	DESCRIPTION
`int maxThreadsPerBlock`	The maximum number of threads that a block may contain
`int maxThreadsDim[3]`	The maximum number of threads allowed along each dimension of a block
`int maxGridSize[3]`	The number of blocks allowed along each dimension of a grid
`size_t totalConstMem`	The amount of available constant memory
`int major`	The major revision of the device's compute capability
`int minor`	The minor revision of the device's compute capability
`size_t textureAlignment`	The device's requirement for texture alignment
`int deviceOverlap`	A boolean value representing whether the device can simultaneously perform a `cudaMemcpy()` and kernel execution
`int multiProcessorCount`	The number of multiprocessors on the device
`int kernelExecTimeoutEnabled`	A boolean value representing whether there is a runtime limit for kernels executed on this device
`int integrated`	A boolean value representing whether the device is an integrated GPU (i.e., part of the chipset and not a discrete GPU)
`int canMapHostMemory`	A boolean value representing whether the device can map host memory into the CUDA device address space
`int computeMode`	A value representing the device's computing mode: default, exclusive, or prohibited
`int maxTexture1D`	The maximum size supported for 1D textures