CS 758: Advanced Topics in Computer Architecture

Lecture #4: Grid-Based Programming, Basic Thread Scheduling, & Simulators

Professor Matthew D. Sinclair

Some of these slides were developed by Tim Rogers at the Purdue University, Tor Aamodt at the University of British Columbia, and Wen-mei Hwu & David Kirk at the University of Illinois at Urbana-Champaign.

Slides enhanced by Matt Sinclair

Announcements

- Updates to schedule per HW0
  - See course schedule
- Euler access
  - Some issues with creating new accounts over the weekend
  - Fixed now

Today's Objectives

- Some more complex kernels
  - Grid-based programming
- More on the memory model
- Basic thread scheduling
- GPU simulators

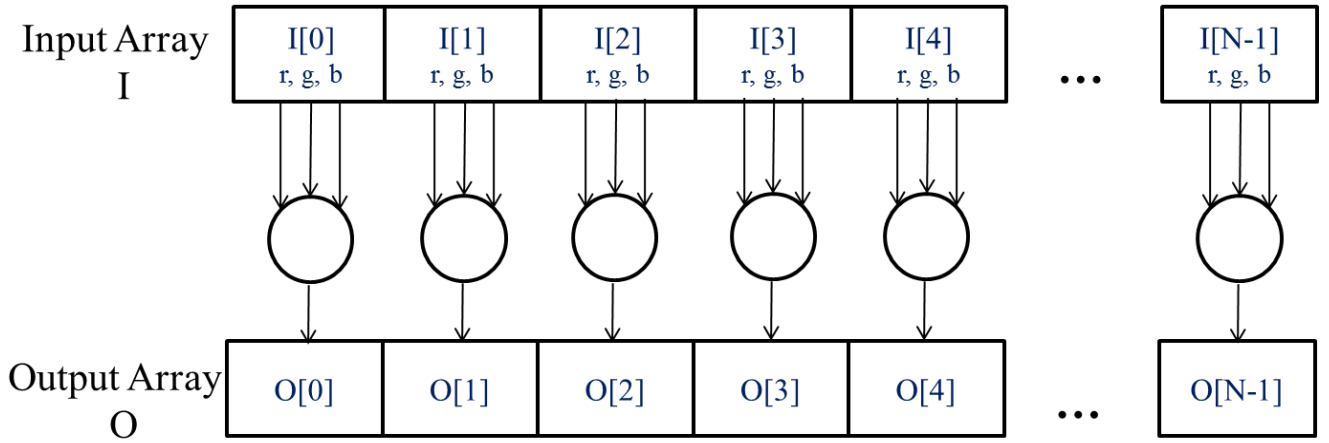
Back to CUDA

- Multi-dimensional kernels
  - Our first kernel was a linear vectorAdd
  - Threads can be arranged in multiple dimension

First Multi-Dimensional Kernel: Conversion to grey-scale

- Every pixel has 3 values to determine the color (R,G,B)
- Compute the **Luminance** value of the pixel
  - Embarrassingly **data-parallel** operation
  - Same operation on every pixel, all independent
  - Parallelism scales 1:1 with input data

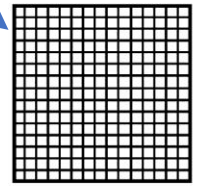
Example of data parallelism



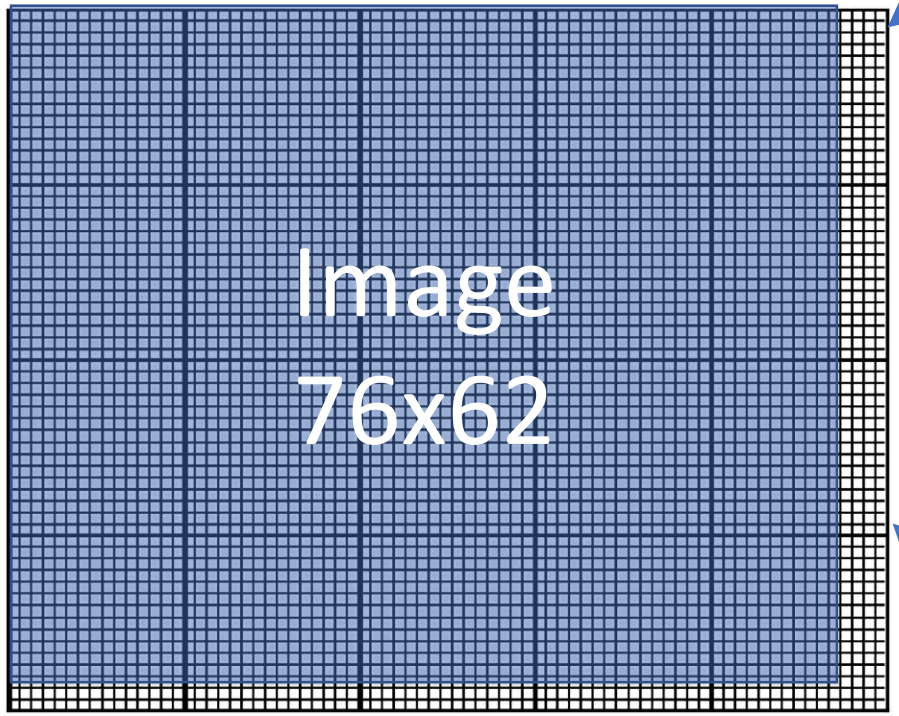
A 2-D Grid configuration

Grid:
20 TB, (80x64) 5120 threads

Threadblock:
256 Threads

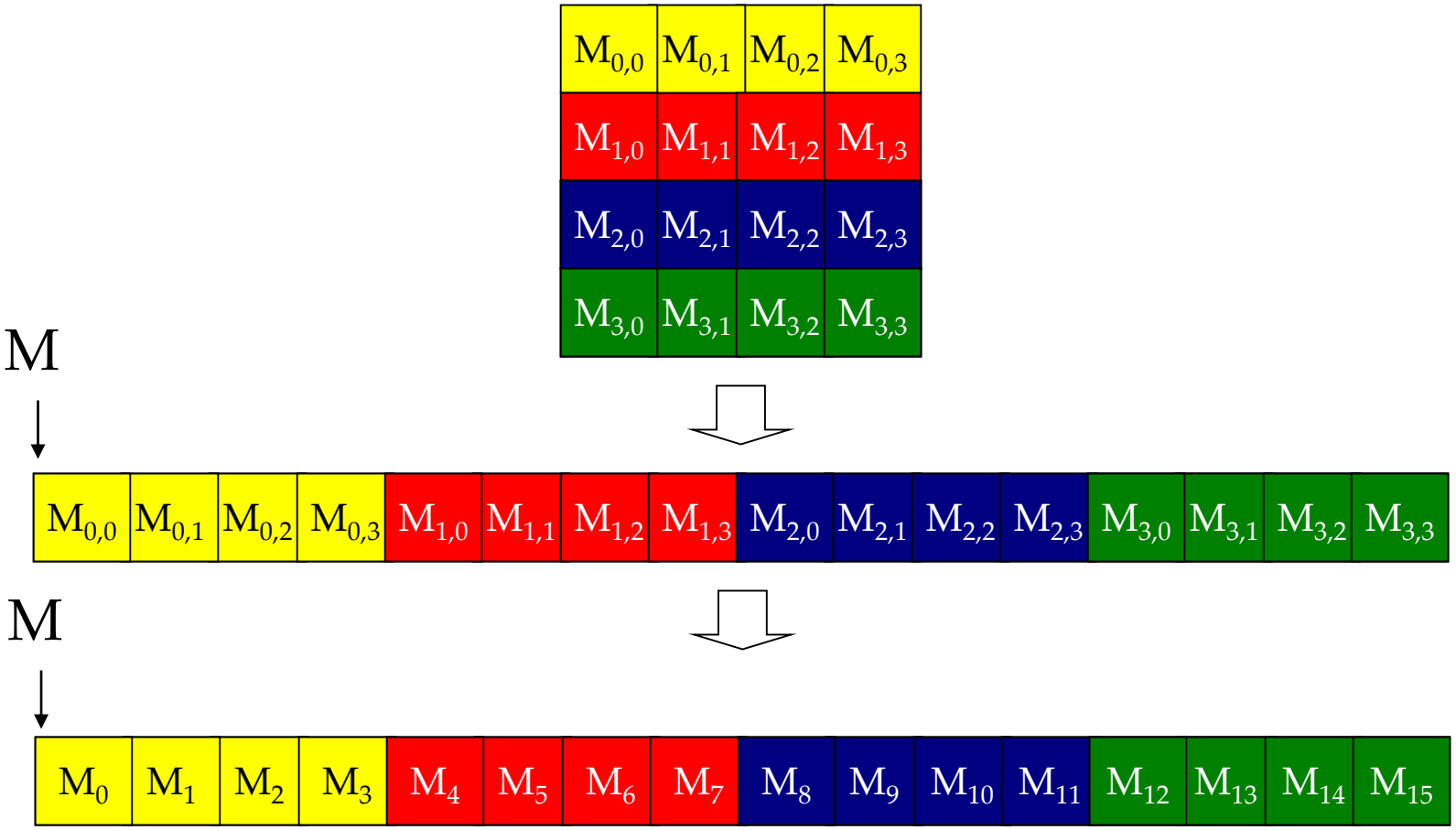


16×16 blocks



Threads at the edges
should do nothing…

Row-Major Memory Layout (C/C++/CUDA)



$M_{2,1}$ → Row*Width+Col = 2*4+1 = 9

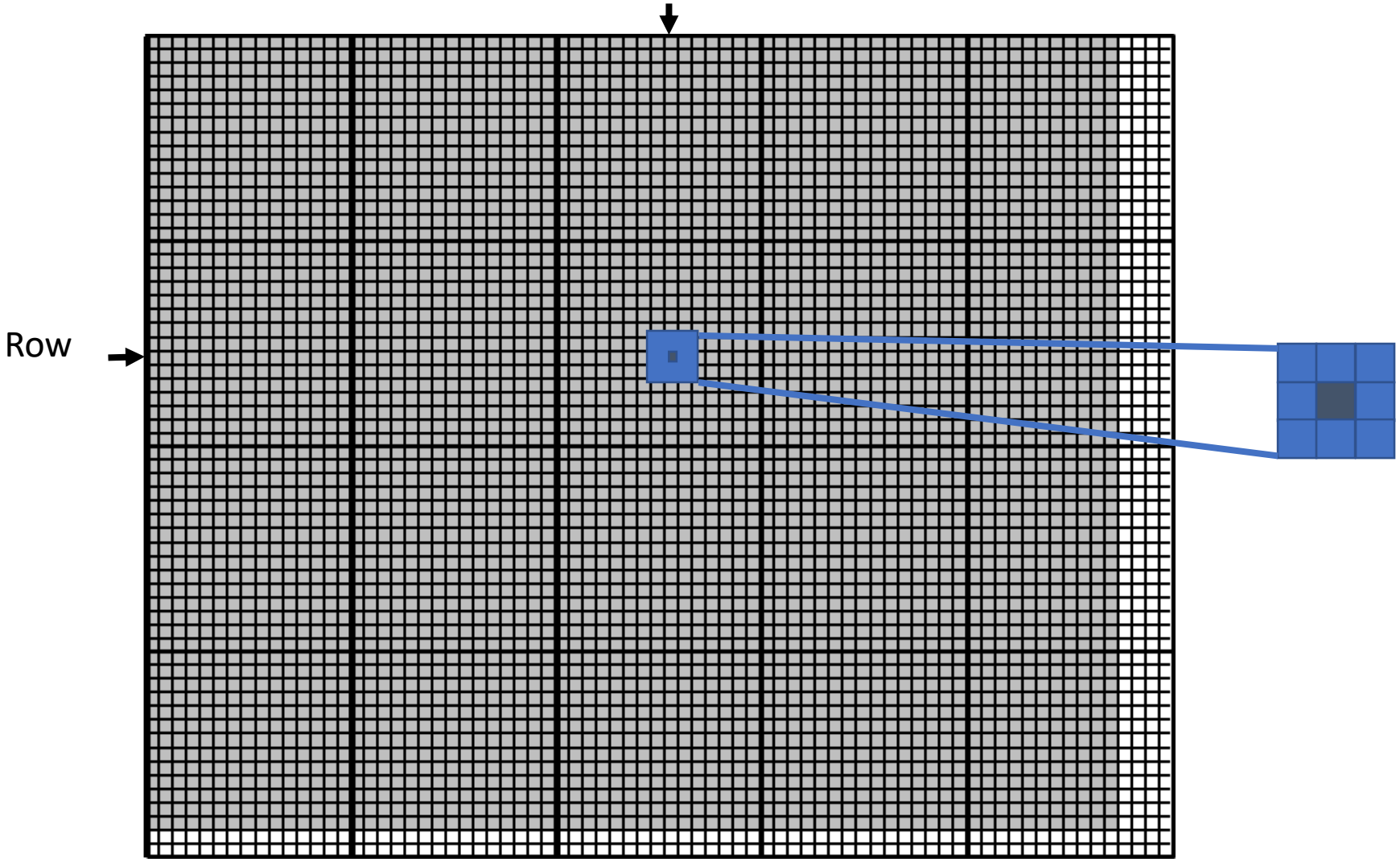
Kernel with 2D thread mapping to data

```
// 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;
 }
}
```

Another 2-D Kernel

- Image Blurring
  - Slightly more complicated: Need data from surrounding pixels – not just 1.

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



Must handle pixels not computed + edge pixels

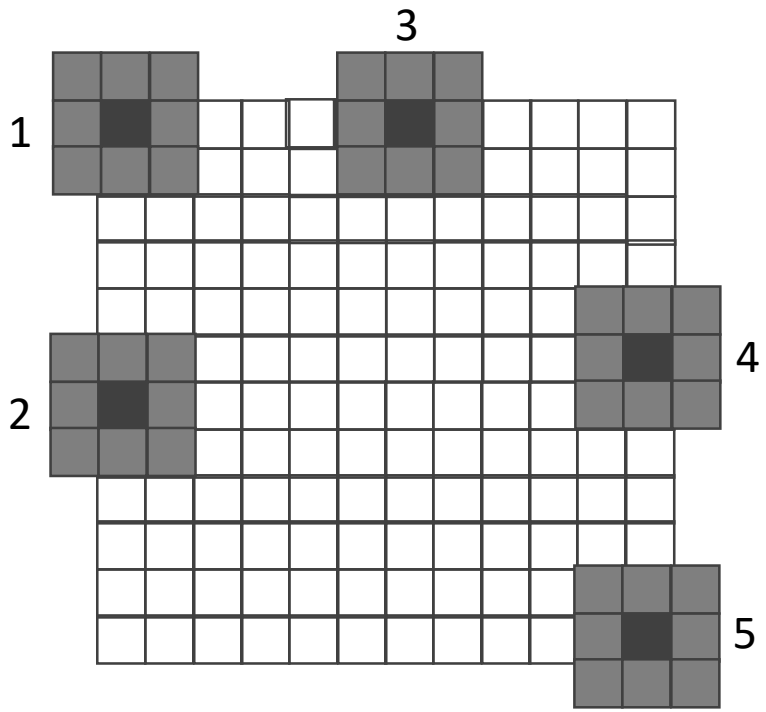


Image Blur Kernel

```
__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);
   }
 }
```

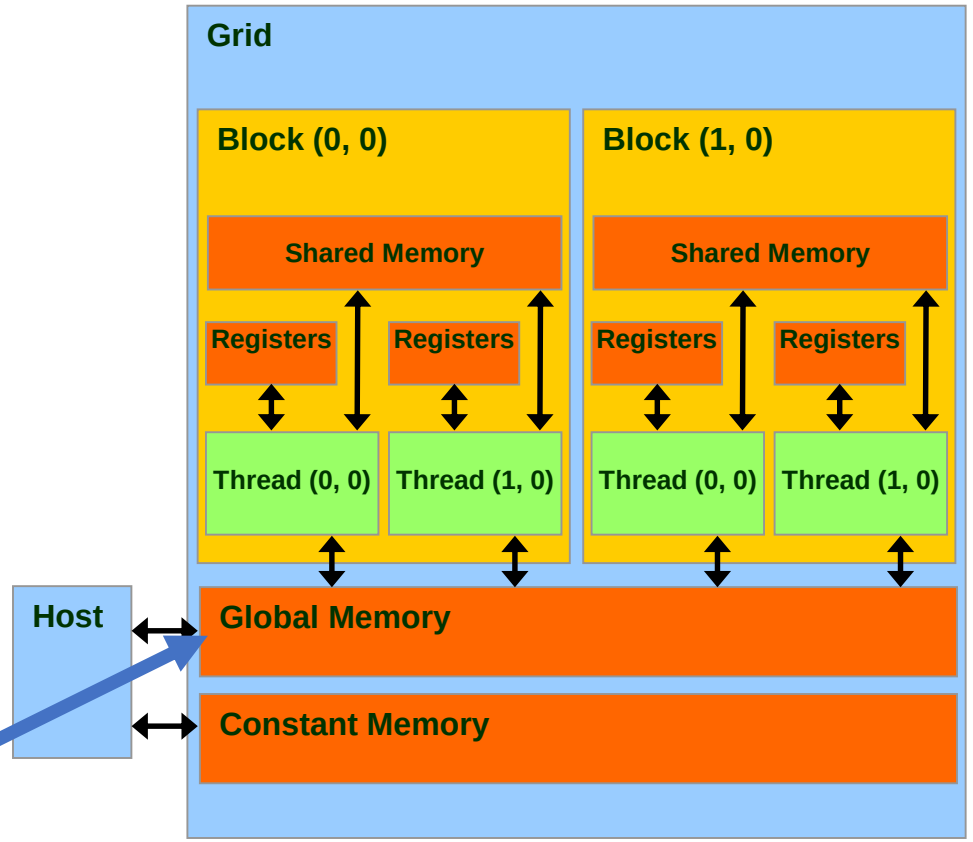
Doubly Nested Loop

Protection against edges

Note: There is shared data amongst threads that is not exploited in this simple kernel

Full Programmer Memory Model

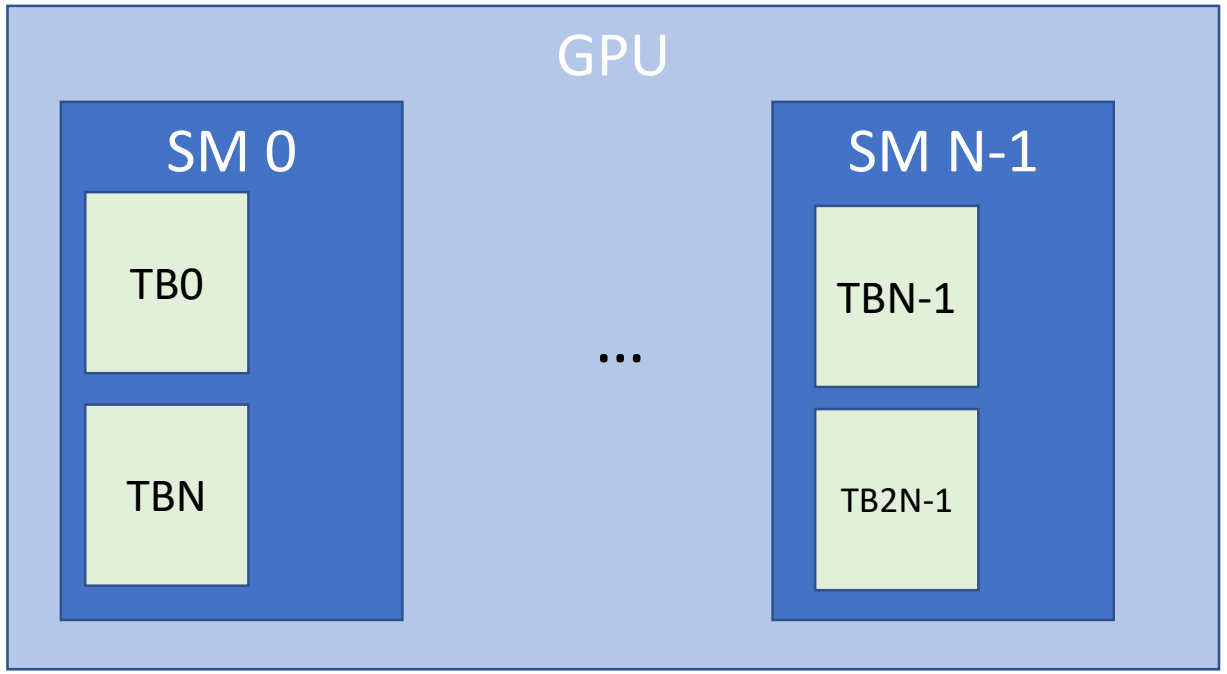
- Each thread can:
  - Read/write per-thread **registers (~1 cycle)**
  - Read/write per-block **shared memory (~5 cycles)**
  - Read/write per-grid **global memory (~500 cycles)**
  - Read/only per-grid **constant memory (~5 cycles with caching)**



Note: Both global and constant memory have a cache hierarchy

High-level Architecture of the GPU (Turing)

Threadblocks / GPU resources usage



- Threads are assigned to Streaming Multiprocessors in block granularity
  - Up to **32** blocks to each SM as resource allows
  - Maxwell SM can take up to **2048** threads
- Threads run concurrently
  - SM maintains thread/block id #s
  - SM manages/schedules thread execution

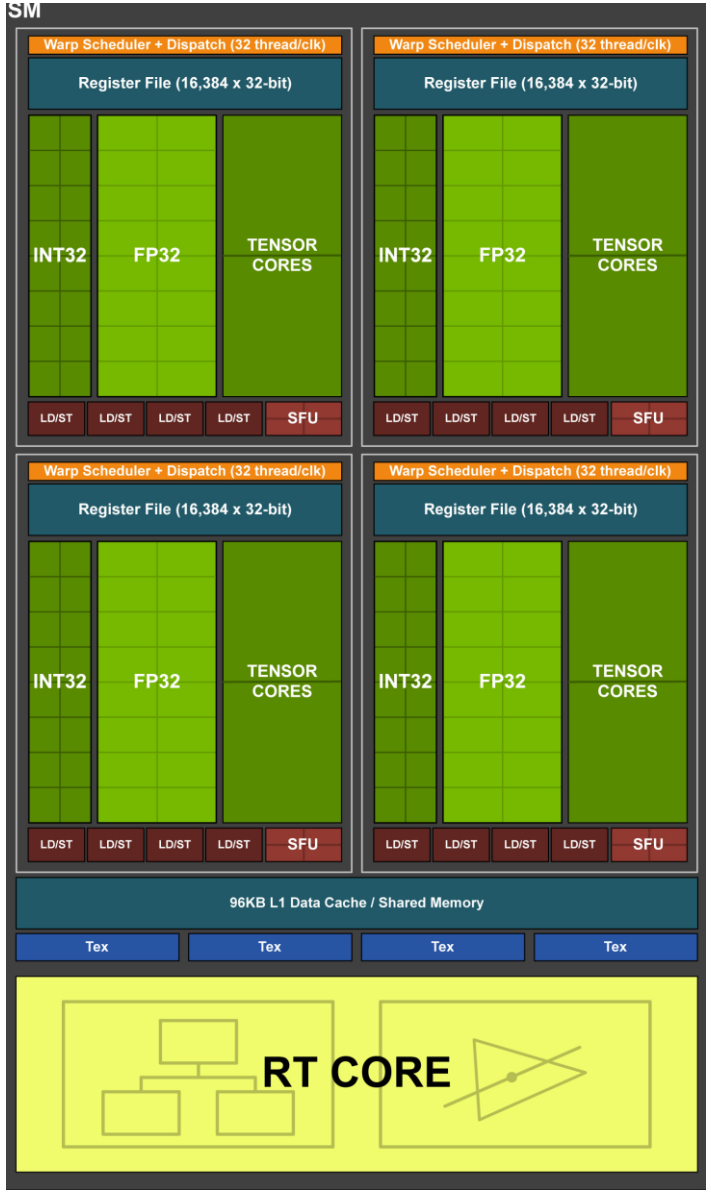
There is no guaranteed scheduling of TB to SM

What is inside an SM? (Pascal)

Reg file/Shared memory are statically partitioned between threads

Finite number of threads/warps/threadblocks can be managed at once

What's inside an SM? (Turing)

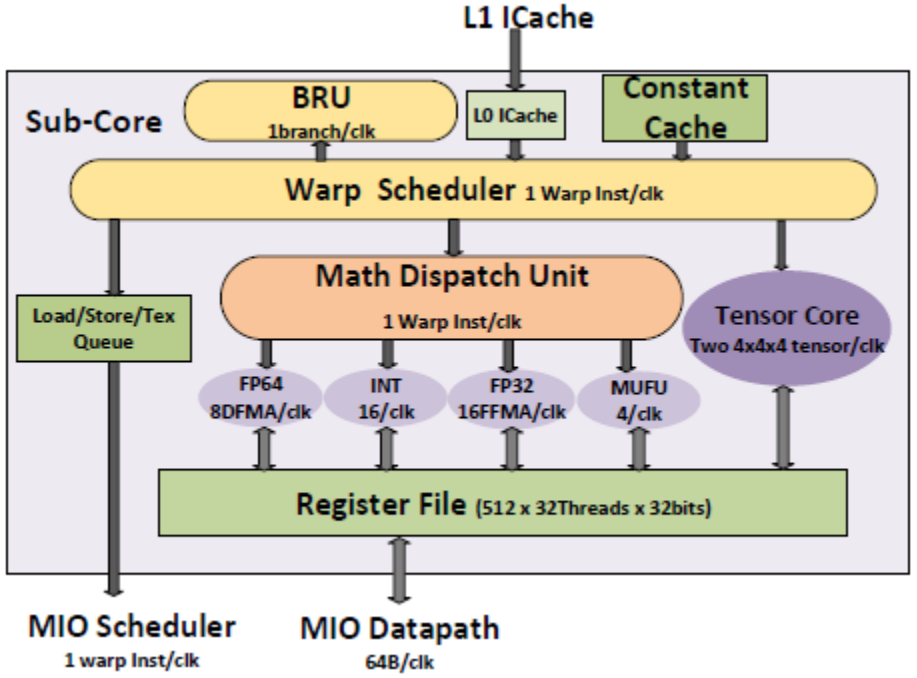


More units, larger memory/RF, HW RT support, TensorCores, ...

TensorCores

- Observation:
  - Machine learning applications (and some others) can use reduced precision
  - Matrix multiplication operations (e.g., FMA) are common
- Solution:
  - Add specialized ALUs to SMs
- Turing: FP16, INT8, INT4
- Volta: FP16

Lots of extra FLOPs for apps that can use them



Volta SM Sub-Core [Choquette 2018]

GPU ISA

- NVIDIA and AMD usually have a “virtual” assembly ISA – PTX, HSAIL
  - PTX == Parallel Thread Execution
  - HSAIL == HSA Intermediate Language
    - AMD recently deprecated HSAIL [Gutierrez 2018]
- Idea: portability across different devices (from the same company)
  - Compiler can generate PTX/HSAIL once (stable platform)
  - Later device will “finalize” PTX/HSAIL to lower-level assembly
    - Idea: lower-level assembly can change from GPU to GPU
    - NVIDIA: SASS
    - SASS details not publicly disclosed

In most cases, you won’t need to write code below the CUDA level

GPU ISA (Cont.)

- Example: [NVIDIA PTX Guide]

```
    .reg .b32 r1, r2;
    .global .f32 array[N];

start:
    mov.b32 r1, %tid.x;
    shl.b32 r1, r1, 2; // shift thread id by 2 bits
    ld.global.b32 r2, array[r1]; // thread[tid] gets array[tid]
    add.f32 r2, r2, 0.5; // add 1/2
```

What CUDA code does this correspond to?

GPU Simulators

- 3 widely used (open-source) GPU simulators:
  - GPGPU-Sim
    - Focuses on (discrete) NVIDIA GPUs -- CUDA and OpenCL support
    - Significant detail, but a little less than gem5
    - Mostly simulates NVIDIA's PTX ISA (some SASS-like support with PTXPlus mode)
  - gem5 (gem5-gpu)
    - Focuses on (integrated) AMD GPUs -- HIP and OpenCL support
    - Lots of detail: models low-level GCN3 ISA, basically all components
    - Lots of extensions: e.g., multiple distributed devices, graphics, NVIDIA GPUs
    - See separate slide deck
  - Multi2Sim (mGPUSim)
    - In between GPGPU-Sim and gem5
    - Recently added Multi-GPU support (in Go!)

AMD GPU Tutorial

- See separate slide deck
- Details not discussed today:
  - Lots of details on how kernels are launched
  - HSA Signals
  - HSA Queues
  - Command and Dispatch
  - Command Queues
  - Doorbell and Event Pages
  - hUMA (Unified Memory)
  - Ruby
  - …

For Next Class

- HW0 Due
- HW1 Assigned
- Read Rogers 2015
- Read GPGPA Chapter 3

Parting Thought

- Can GPUs afford to simultaneously power both TensorCores and traditional cores (e.g., CUDA Cores) in their compute units?