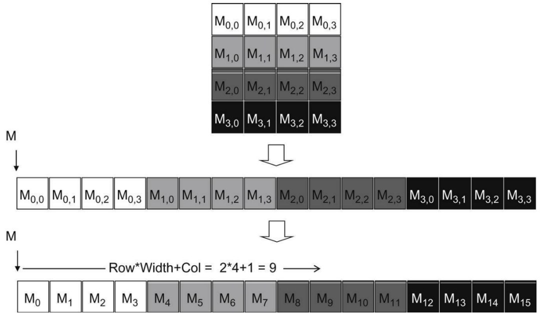
CSc 8530
Parallel Algorithms

Spring 2019

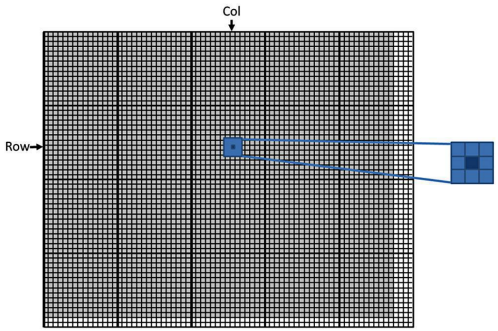
March 12th, 2019

Memory organization



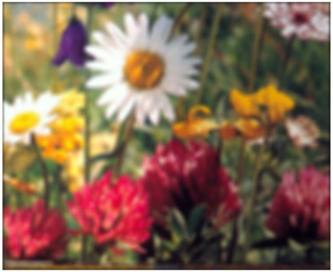
- The number of columns of dynamically allocated arrays is not known at compile time
- Thus, in CUDA C we cannot access elements directly, i.e., d_Pin[j][i]
- We have to treat 2D (and 3D) arrays as "flat", 1D arrays

Image blurring



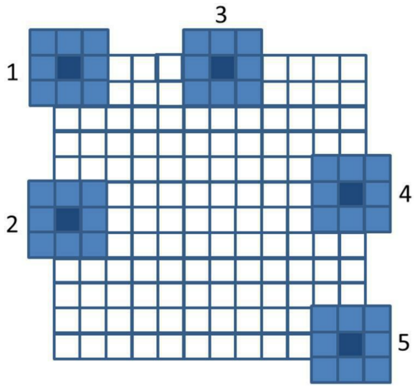
- Here, the neighbors of a pixel are the six pixels that share either an edge or a corner
  1. More generally, we can consider larger neighborhoods (e.g., those that are three pixels away or less)
- In image blurring, we replace the original value by a (potentially weighted) sum of the values of its neighborhood

Image blurring



- Image blurring removes high-frequency details from an image
- It can be useful for reducing certain types of noise
  - e.g., the uneven illumination from flash in dark images

Border handling

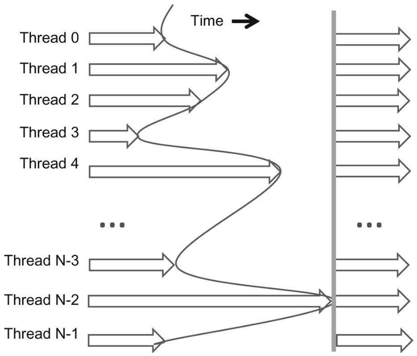


- We need special checks to handle border and corner pixels
- Only valid pixels are used to compute the average value

Thread synchronization

- We have not yet explored how to synchronize the work done by different threads
  - So far, the work done by each thread was independent of the others
- CUDA has mechanisms for coordinating activity across threads
- Essentially, a thread can request to wait until other threads have finished their processing (**barrier synchronization**)
  - By calling a built-in CUDA function: `__syncthreads()`
  - Note that the name of this function has two underscores
- When a thread calls `__syncthreads()`, it will be frozen until all the other threads in its block reach the same instruction

Barrier synchronization



- All synchronized threads must wait until every one of them has reached the barrier location
- Faster threads will have to wait idle while the slower ones catch up

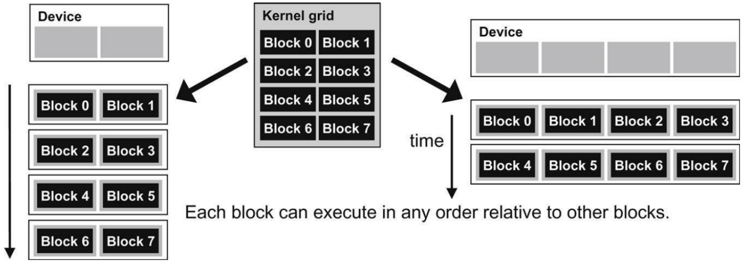
Barrier synchronization

- In CUDA, all threads in a block have to execute the `__syncthreads()` call
- If its inside an **if**-statement, then all threads have to execute the same path
  - It is the responsibility of the programmer to ensure this happens
- Ideally, all threads should execute as much in lock-step as possible, to minimize waiting times
- One must also make sure that all threads have the necessary resource to finish
  - A single, hanging thread will freeze the entire program
- CUDA solves this problem by giving the same access to a resource to all threads in a block

Barrier synchronization

- All threads in a block have the same access to resources
    - i.e., resource access is at the block level
- Conversely, threads in different blocks **cannot** synchronize with each other
- This design choice allows CUDA to execute blocks in arbitrary order
- The number of blocks that can be executed in parallel is a function of the hardware
    - Thus, performance can be scaled according to available resources
- This feature is called **transparent scalability**

Transparent scalability

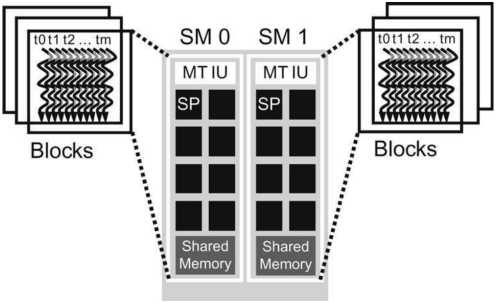


- The effective level of parallelization can be scaled to the available resources
- Above, a GPU with twice as many processors can execute the code twice as fast

Resource assignment

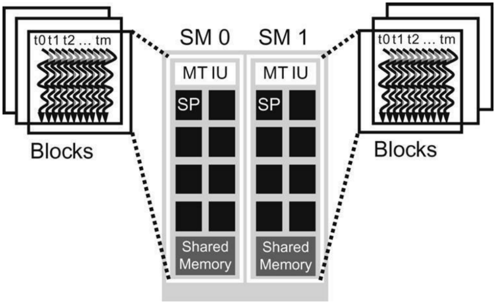
- CUDA initializes a grid of threads when a kernel function is called
- All the threads in a block have access to the same resources
- Currently, these *execution resources* are organized into **Streaming Multiprocessors (SM)**
- An SM can handle multiple blocks simultaneously
  - How many is device dependent
- The system maintains a list of blocks that haven't been executed and assigns them to SMs when they become available

Streaming Multiprocessors (SM)



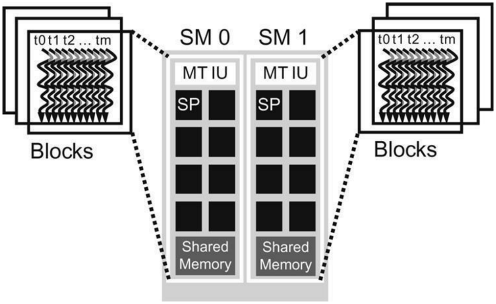
- Above, each SM is assigned 3 blocks
- SMs use built-in registers to track all their assigned threads
  - The number of registers sets a limit on the number of simultaneous threads

Streaming Multiprocessors (SM)



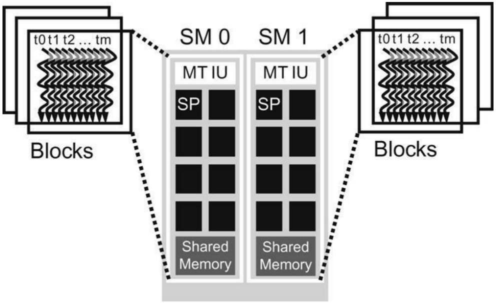
- Each GPU generation has a limit on the number of blocks and threads that an SM can handle
  - e.g., the Fermi architecture allows up to 8 blocks and 1536 threads per SM

Streaming Multiprocessors (SM)



- Each GPU generation has a limit on the number of blocks and threads that an SM can handle
  - e.g., the Fermi architecture allows up to 8 blocks and 1536 threads per SM
  - Viable: [6 blocks, 256 threads], [3 blocks, 512 threads]

Streaming Multiprocessors (SM)



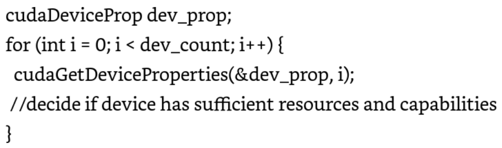
- Each GPU generation has a limit on the number of blocks and threads that an SM can handle
  - e.g., the Fermi architecture allows up to 8 blocks and 1536 threads per SM
  - Viable: [6 blocks, 256 threads], [3 blocks, 512 threads]
  - Not viable: [4 blocks, 512 threads], [12 blocks, 128 threads]

Querying device properties

- CUDA automatically schedules grids based on the available resources on the device
- However, sometimes we want to know exactly what resources are available
  - e.g., to scale some part of our algorithm up or down
- CUDA has several functions to query device properties
- For example, to find out the number of GPUs:
  - `int dev_count;`
  - `cudaGetDeviceCount(&dev_count);`
  - Note how the value is assigned by reference, not as a return value (unfortunately)

Querying device properties

- We can use the following **for**-loop to query the properties of every device in our system:



```
cudaDeviceProp dev_prop;
for (int i = 0; i < dev_count; i++) {
 cudaGetDeviceProperties(&dev_prop, i);
 //decide if device has sufficient resources and capabilities
}
```

- Each GPU is assigned a number from 0 to `dev_count-1`
- However, this assignment **can vary arbitrarily**
  - Your code should never assume that a GPU has a particular index

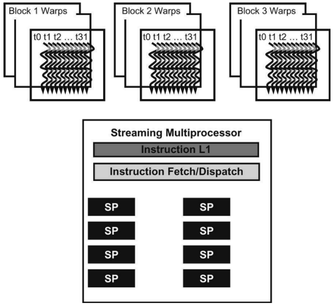
Device properties

- The cudaDeviceProp variable is a C struct
  - **Note:** the name of the variable, e.g., `dev_prop`, is arbitrary
- Some relevant fields:
  - `dev_prop.maxThreadsPerBlock`
  - `dev_prop.multiProcessorCount` (number of SMs)
  - `dev_prop.clockRate` (the clock frequency)
  - `dev_prop.maxThreadsDim[0]` (max number of threads in the x direction)
    - `dev_prop.maxThreadsDim[1]` and `dev_prop.maxThreadsDim[2]` for y and z, resp.
  - `dev_prop.maxGridSize[0]` (max number of blocks in the x direction)
    - `dev_prop.maxGridSize[1]` and `dev_prop.maxGridSize[2]` for y and z, resp.

Thread scheduling

- How threads actually get scheduled on an SM is implementation- and hardware-specific
- Currently, most blocks are subdivided into *warps* of size 32
    - Warps are **not** part of the CUDA specification
    - But knowledge is useful in practice
- `dev_prop.warpSize` tells us the exact warp size for the current device
- An SM executes all the threads in a warp using Single Instruction, Multiple Data (SIMD)
    - Thus, all the threads in a warp execute in lock-step

Warp scheduling



- Each SM has a number of **Streaming Processors (SP)**
  - These actually execute the threads
- In early GPUs, an SM could only execute one instruction for one warp at a time
- Current GPUs, each SM can handle a few warps at a time

Latency tolerance

- SMs define far more warps than the SPs they have available
  - For reducing *latency*
  - Here, latency refers to how long the processor sits idle
- For example, if a warp needs to wait for an I/O operation, it will not be selected for execution
- A priority mechanism is used for selecting among warps that are ready to be executed
  - The details are beyond the scope of the course
- With many warps, the SM will generally always find one that is ready to be executed
  - Which maximizes hardware utilization
- Intuitively, the GPU hides the latency of one warp by executing another warp in the meantime

A simple exercise

- Assume a CUDA device allows up to 8 blocks and 1024 threads per SM
    - And it only allows up to 512 threads per block
- For image blurring, should we use 8×8, 16×16, or 32×32 blocks?

A simple exercise: answer

- 8×8 blocks:
  - 64 threads per block
  - $1024/64 = 12$ blocks to fully utilize an SM
    - However, each SM only allows 8 blocks
  - So, we can only schedule $64*8 = 512$ threads at a time
- 16×16 blocks:
  - 256 threads per block
  - $1024/256 = 4$ blocks to fully utilize an SM
    - Within the SM limit
  - So, we can schedule $256*4 = 1024$ threads at a time
- 32×32 blocks:
  - 1024 threads per block
    - Exceeds the 512 per-block limit
  - Not viable on this hardware
- **Answer:** 16×16 blocks give the best resource utilization

Memory access efficiency

- The CUDA kernels we have looked at so far will only achieve a small, real-world speed-up
  - Long access latencies (hundreds of clock cycles)
  - Access to global memory is a bottleneck
- Traffic congestion in accessing global memory can reduce the number of threads that can execute in parallel
  - Leaving many SMs idle
- CUDA provides additional methods for reducing accesses to global memory