LAU
Lebanese American University

CSC 447: Parallel Programming for Multi-Core and Cluster Systems

CUDA Thread Scheduling

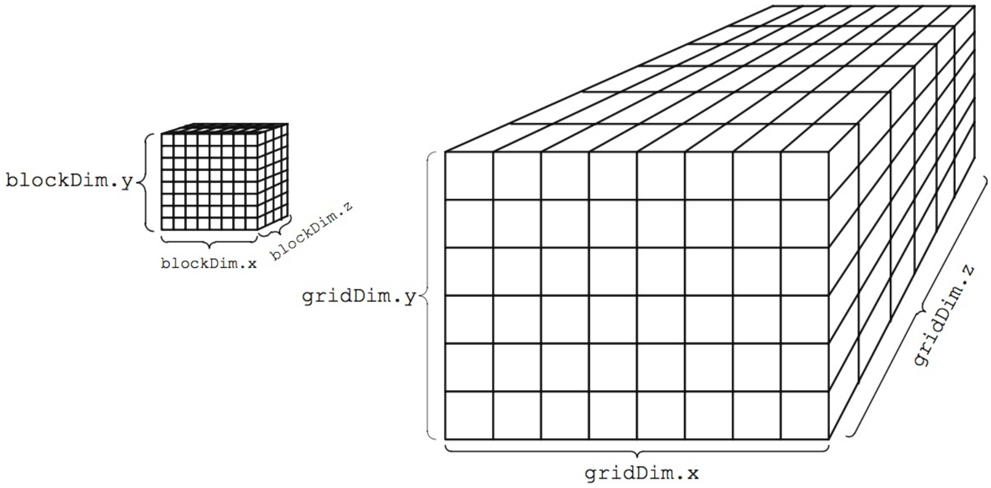
Instructor: Haidar M. Harmanani

Spring 2018

Blocks, Grids, and Threads

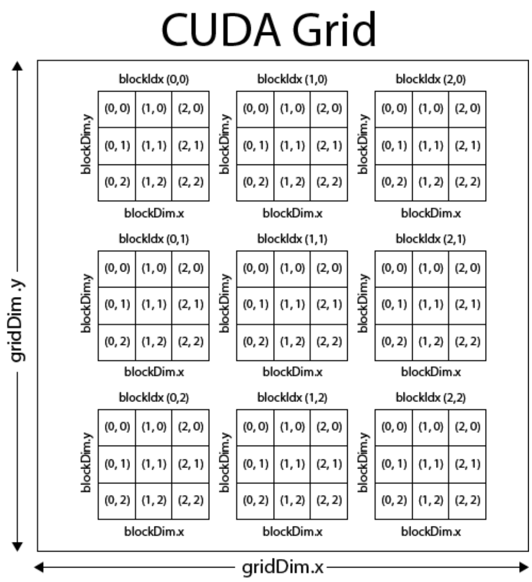
- When a kernel is launched, CUDA generates a grid of threads that are organized in a three-dimensional hierarchy
  - Each grid is organized into an array of thread blocks or *blocks*
  - Each block can contain up to 1,024 threads
  - Number of threads in a block is given in the `blockDim` variable
  - The dimension of thread blocks should be a multiple of 32
- Each thread in a block has a unique `threadIdx` value
  - Combine the **`threadIdx`** and **`blockIdx`** values to create a unique global index

Blocks, Grids, and Threads



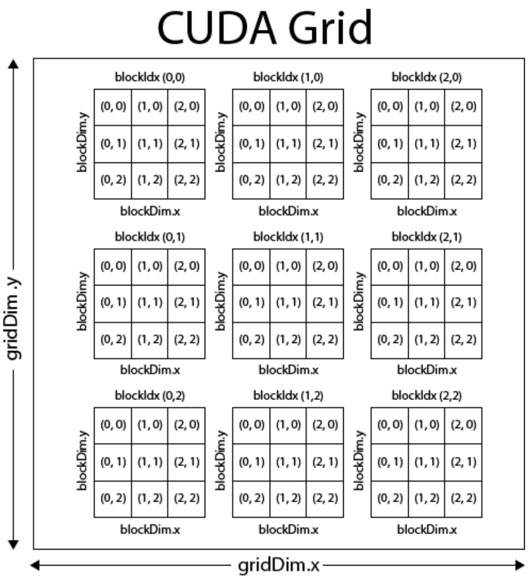
Global Thread IDs: 2D grid of 2D blocks

- tx = threadIdx.x
- ty = threadIdx.y
- bx = blockIdx.x
- by = blockIdx.y
- bw = blockDim.x
- bh = blockDim.y
- id_x = tx + bx * bw
- id_y = ty + by * bh

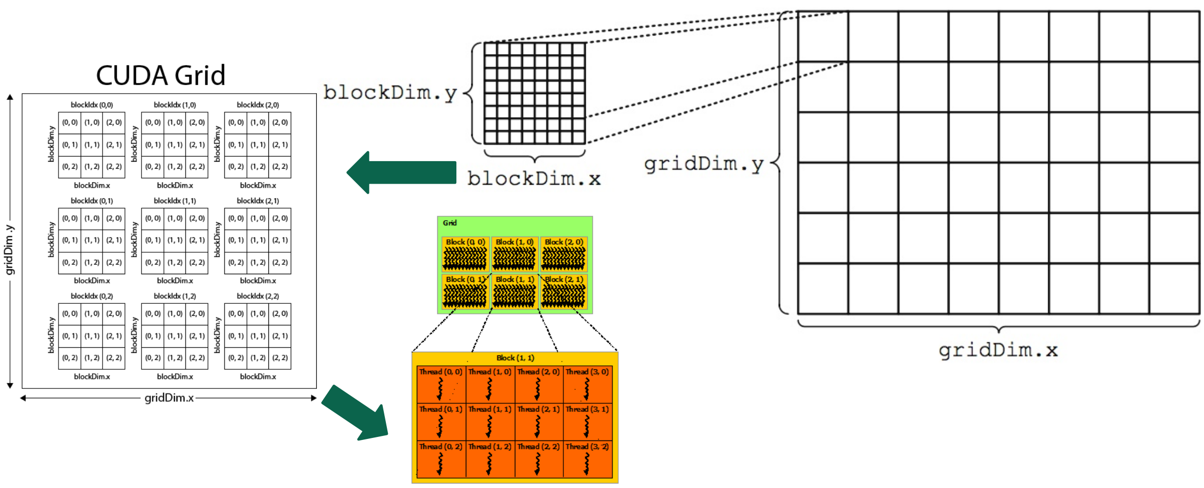


Blocks, Grids, and Threads

- `blockIdx`: The block index within the grid
- `gridDim`: The dimensions of the grid
- `blockDim`: The dimensions of the block
- `threadIdx`: The thread index within the block.



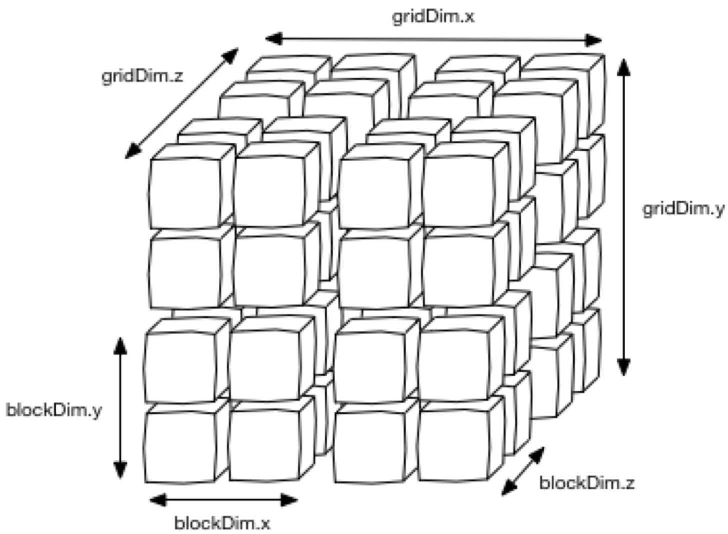
Blocks, Grids, and Threads



- Thread index = threadIdx.x + blockIdx.x * blockDim.x

Global Thread IDs: 3D grid of 3D blocks

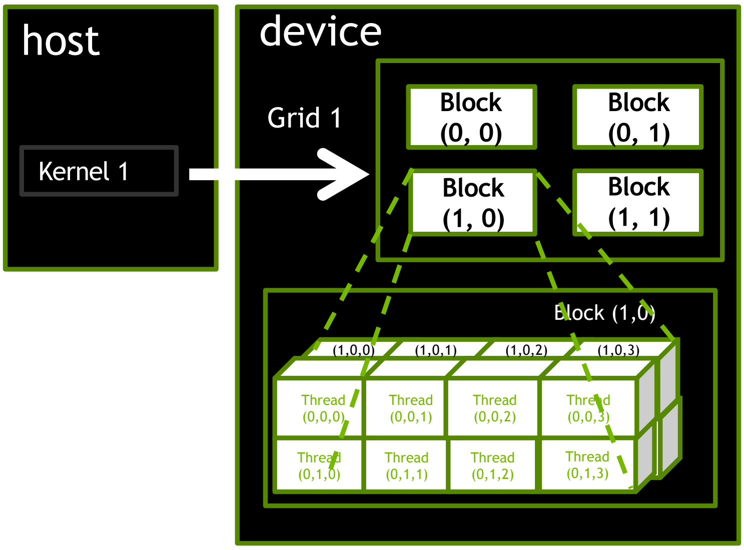
- tx = threadIdx.x
- ty = threadIdx.y
- tz = threadIdx.z
- bx = blockIdx.x
- by = blockIdx.y
- bz = blockIdx.y
- bw = blockDim.x
- bh = blockDim.y
- bd = blockDim.z
- id_x = tx + bx * bw
- id_y = ty + by * bh
- id_z = tz + bz * hd



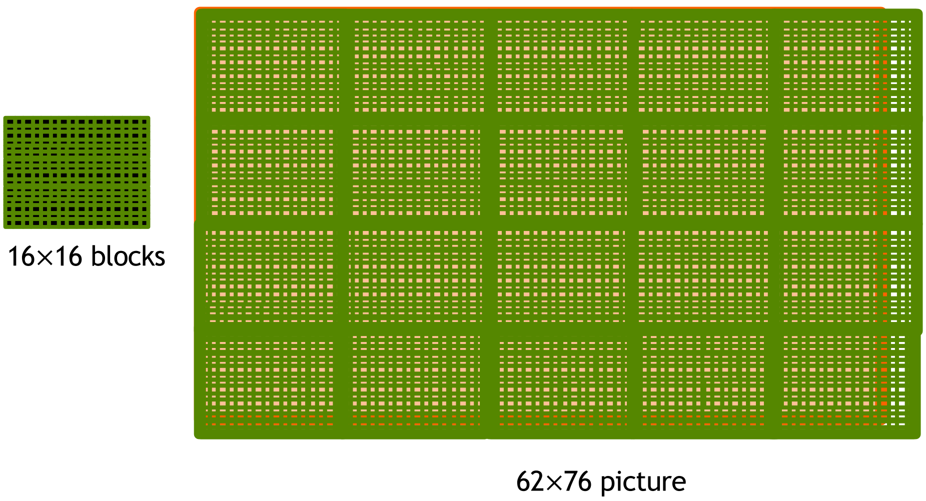
Blocks Must be Independent

- Any possible interleaving of blocks should be valid
  - presumed to run to completion without pre-emption
  - can run in any order
  - can run concurrently OR sequentially
- Blocks may coordinate but not synchronize
- Independence requirement gives *scalability*

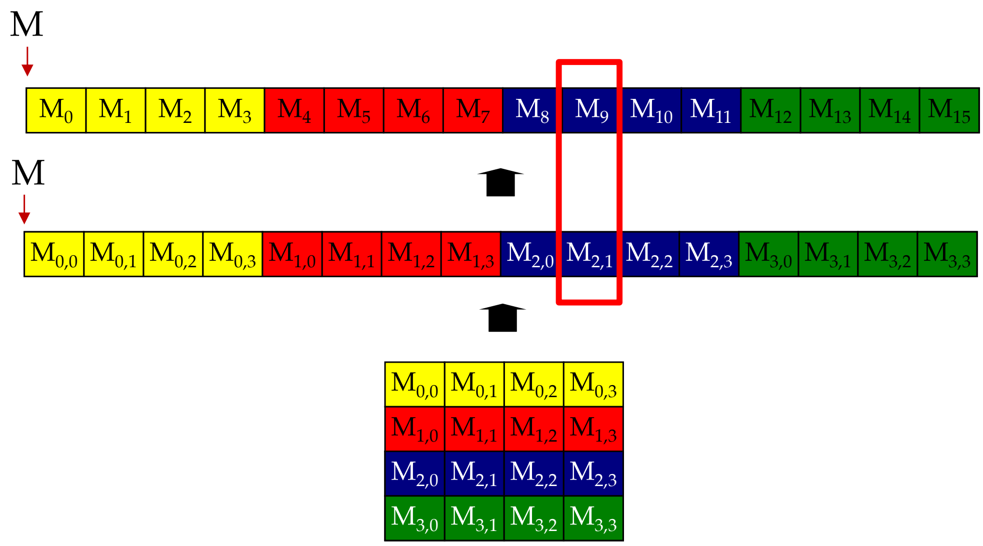
Example 2: A Multi-Dimensional Grid



Processing a Picture with a 2D Grid



Row-Major Layout in C/C++



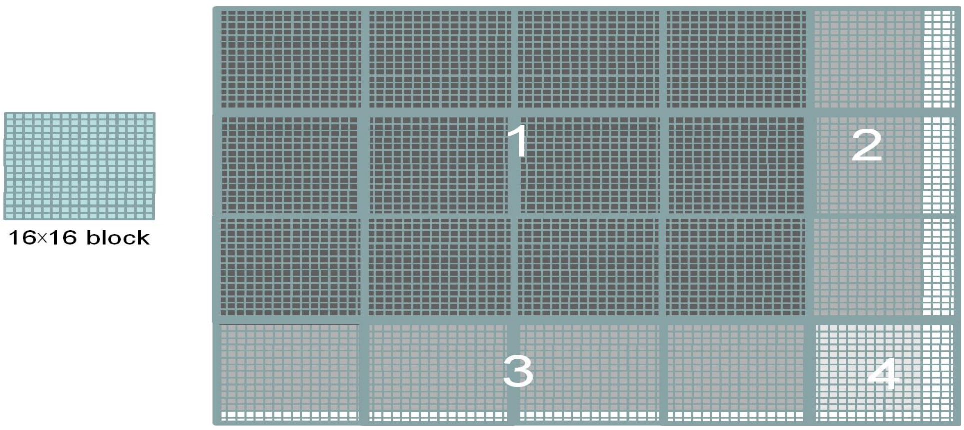
Source Code of a PictureKernel

```
__global__ void PictureKernel(float* d_Pin, float* d_Pout, int height, int width)
{

  // Calculate the row # of the d_Pin and d_Pout element
  int Row = blockIdx.y*blockDim.y + threadIdx.y;

  // Calculate the column # of the d_Pin and d_Pout element
  int Col = blockIdx.x*blockDim.x + threadIdx.x;

  // each thread computes one element of d_Pout if in range
  if ((Row < height) && (Col < width)) {
    d_Pout[Row*width+Col] = 2.0*d_Pin[Row*width+Col];
  }
}
```

Host Code for Launching PictureKernel

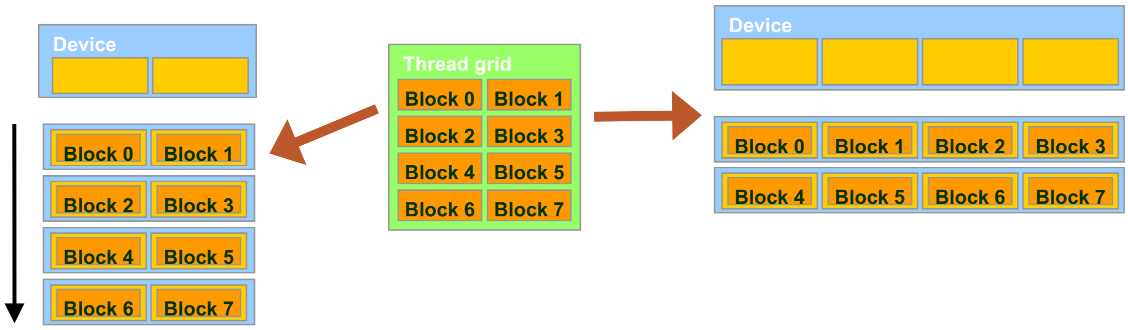
```
// assume that the picture is m×n,
// m pixels in y dimension and n pixels in x dimension
// input d_Pin has been allocated on and copied to device
// output d_Pout has been allocated on device
…
dim3 DimGrid((n-1)/16 + 1, (m-1)/16+1, 1);
dim3 DimBlock(16, 16, 1);
PictureKernel<<<DimGrid,DimBlock>>>(d_Pin, d_Pout, m, n);
…
```


Covering a 62×76 Picture with 16×16 Blocks



Not all threads in a Block will follow the same control flow path.

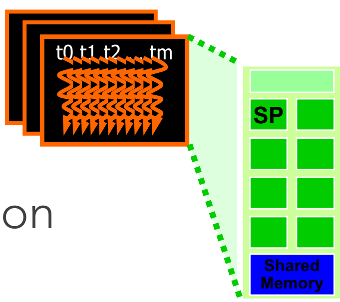
Transparent Scalability



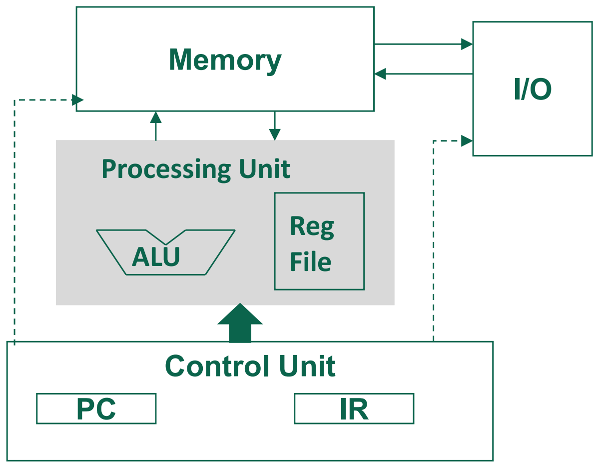
- Each block can execute in any order relative to others
  - Concurrently or sequentially
  - Facilitates scaling of the same code across many devices
- Hardware is free to assign blocks to any processor at any time
  - A kernel scales to any number of parallel processors

Example 1: Executing Thread Blocks

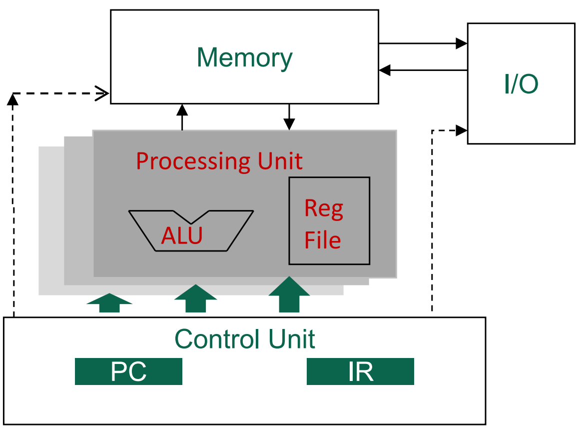
- Threads are assigned to *Streaming Multiprocessors (SM)* in block granularity
  - Up to 8 blocks to each SM as resource allows
  - Fermi SM can take up to 1536 threads
    - Could be 256 (threads/block) * 6 blocks
    - Or 512 (threads/block) * 3 blocks, etc.
- SM maintains thread/block idx #s
- SM manages/schedules thread execution



The Von-Neumann Model



The Von-Neumann Model with SIMD units



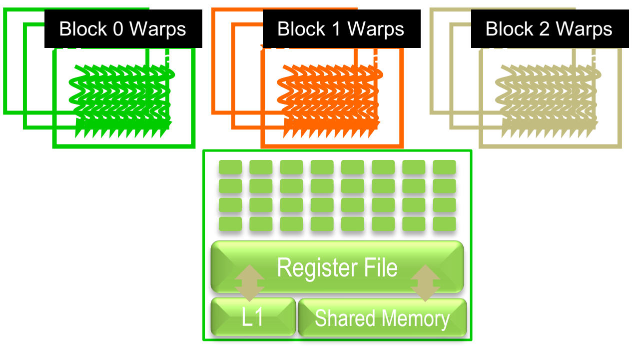
Single Instruction Multiple Data (SIMD)

Warps as Scheduling Units

- Each Block is executed as 32-thread Warps
  - An implementation decision, not part of the CUDA programming model
  - Warps are scheduling units in SM
  - Threads in a warp execute in SIMD
  - Future GPUs may have different number of threads in each warp

Warp Example

- 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
  - There are 8 * 3 = 24 Warps



Blocks, Grids, and Threads

- Instructions *are issued per warp*
  - It takes 4 clock cycles to issue a single instruction for the whole warp
- If an operand is not ready the warp will stall
- Threads in any given warp execute in lock-step, but to synchronise across warps, you need to use `__syncthreads()`

Example: Thread Scheduling (Cont.)

- SM implements zero-overhead warp scheduling
  - Warps whose next instruction has its operands ready for consumption are eligible for execution
  - Eligible Warps are selected for execution based on a prioritized scheduling policy
  - All threads in a warp execute the same instruction when selected

Fermi Architecture

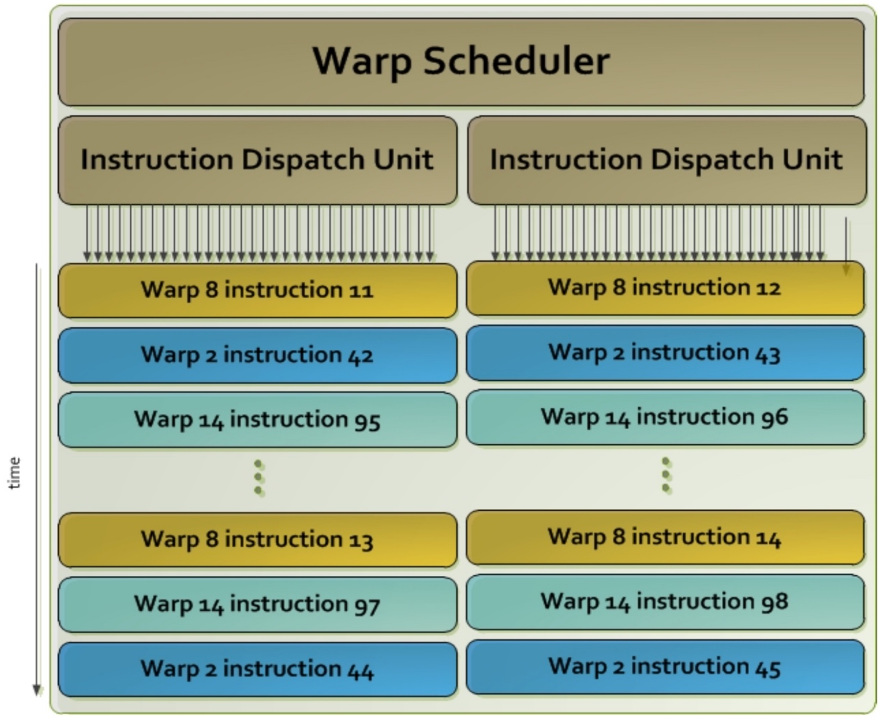
- Has 16 SM that each can process at most 8 blocks
- Each SM has 32 cores for a total of 512 cores

Block Granularity Considerations

- For Matrix Multiplication using multiple blocks, should we use 8X8, 16X16 or 32X32 blocks for Fermi?
  - For 8X8, we have 64 threads per block.
    - We will need 1536/64 = 24 blocks to fully occupy an SM since each SM can take up to 1536 threads
    - However, each SM has only 8 Blocks, only 64x8 = 512 threads will go into each SM!
    - This means that the SM execution resources will likely be underutilized because there will be fewer warps to schedule around long latency operations.

Block Granularity Considerations

- For Matrix Multiplication using multiple blocks, should I use 8X8, 16X16 or 32X32 blocks for Fermi?
  - For 16X16, we have 256 threads per Block. Since each SM can take up to 1536 threads, it can take up to 6 Blocks and achieve full capacity unless other resource considerations overrule.
  - For 32X32, we would have 1024 threads per Block. Only one block can fit into an SM for Fermi. Using only 2/3 of the thread capacity of an SM.



Each Kepler SMX contains 4 Warp Schedulers, each with dual Instruction Dispatch Units. A single Warp Scheduler Unit is shown above.

Performance Tuning¶

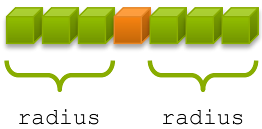
- For optimal performance, the programmer has to juggle
  - finding enough parallelism to use all SMs
  - finding enough parallelism to keep all cores in an SM busy
  - optimizing use of registers and shared memory
  - optimizing device memory access for contiguous memory
  - organizing data or using the cache to optimize device memory access for contiguous memory

Example: Cooperating Threads

- Memory Management
- Blocks
- Threads
- Indexing
- Shared memory
- __syncthreads()
- Asynchronous operation
- Handling errors
- Managing devices

1D Stencil

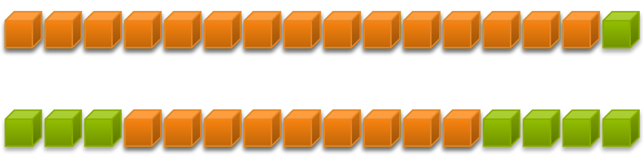
- Consider applying a 1D stencil to a 1D array of elements
  - Each output element is the sum of input elements within a radius
- If radius is 3, then each output element is the sum of 7 input elements:



Implementing Within a Block

- Each thread processes one output element
  - blockDim.x elements per block

- Input elements are read several times
  - With radius 3, each input element is read seven times

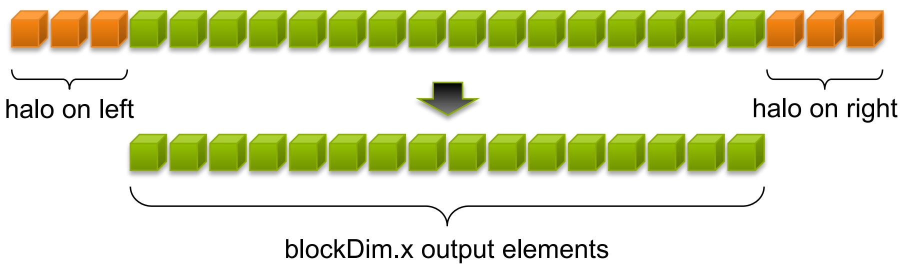


Sharing Data Between Threads

- Terminology: within a block, threads share data via shared memory
- Extremely fast on-chip memory, user-managed
- Declare using `__shared__`, allocated per block
- Data is not visible to threads in other blocks

Implementing With Shared Memory

- Cache data in shared memory
  - Read (blockDim.x + 2 * radius) input elements from global memory to shared memory
  - Compute blockDim.x output elements
  - Write blockDim.x output elements to global memory
  - Each block needs a halo of radius elements at each boundary

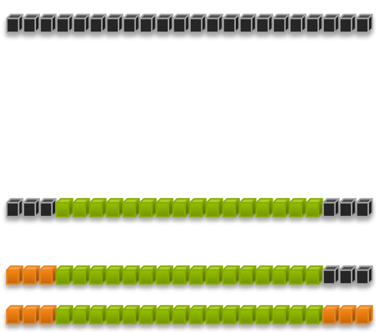


1D Stencil Computation Example , Radius = 1

```
// assume u[i] initialized to some values
for (s=1; s<T; s+=2) {
    for (i=1; i<(N-1); i++) {
       tmp[i] = 1/3 * (u[i-1] + u[i] + u[i+1]); // S1
    }

    for (j=1; j<(N-1); j++) {
       u[i] = 1/3 * (tmp[j-1] + tmp[j] + tmp[j+1]); // S2  }
}
```


Stencil Kernel

```
__global__ void stencil_1d(int *in, int *out) {
  __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
  int gindex = threadIdx.x + blockIdx.x * blockDim.x;
  int lindex = threadIdx.x + RADIUS;

  // Read input elements into shared memory
  temp[lindex] = in[gindex];
  if (threadIdx.x < RADIUS) {
    temp[lindex - RADIUS] = in[gindex - RADIUS];
    temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
  }
 // Apply the stencil
  int result = 0;
  for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
    result += temp[lindex + offset];

  // Store the result
  out[gindex] = result;
}
```



Data Race!

- The stencil example will not work…
- Suppose thread 15 reads the halo before thread 0 has fetched it…

```
temp[lindex] = in[gindex];
if (threadIdx.x < RADIUS) {
  temp[lindex – RADIUS = in[gindex – RADIUS];
  temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
}

int result = 0;
result += temp[lindex + 1];
```

Store at temp[18]

Skipped, threadIdx > RADIUS

Load from temp[19]

__syncthreads()

- `void __syncthreads();`

- Synchronizes all threads within a block
  - Used to prevent RAW / WAR / WAW hazards

- All threads must reach the barrier
  - In conditional code, the condition must be uniform across the block

```
__global__ void stencil_1d(int *in, int *out) {
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + radius;

    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }
    // Synchronize (ensure all the data is available)
    __syncthreads();
    // Apply the stencil
    int result = 0;
    for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
        result += temp[lindex + offset];
    // Store the result
    out[gindex] = result;
}
```

Stencil Kernel