CSc 8530
Parallel Algorithms

Spring 2019

March 5th, 2019

CPUs vs. GPUs

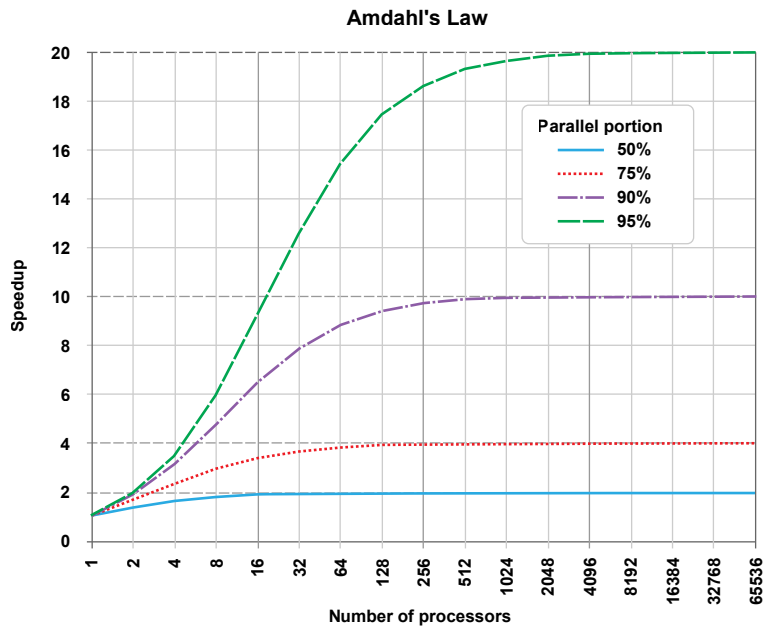
- CPUs minimize single-thread latency and thus have sophisticated control logic and caches
- GPUs maximize throughput (especially of floating-point operations), so they have simple control logic and many arithmetic units

Parallel speedup – Amdahl's law

- How much using a GPU vs. a CPU will speed up an algorithm depends on how much of it we can parallelize:

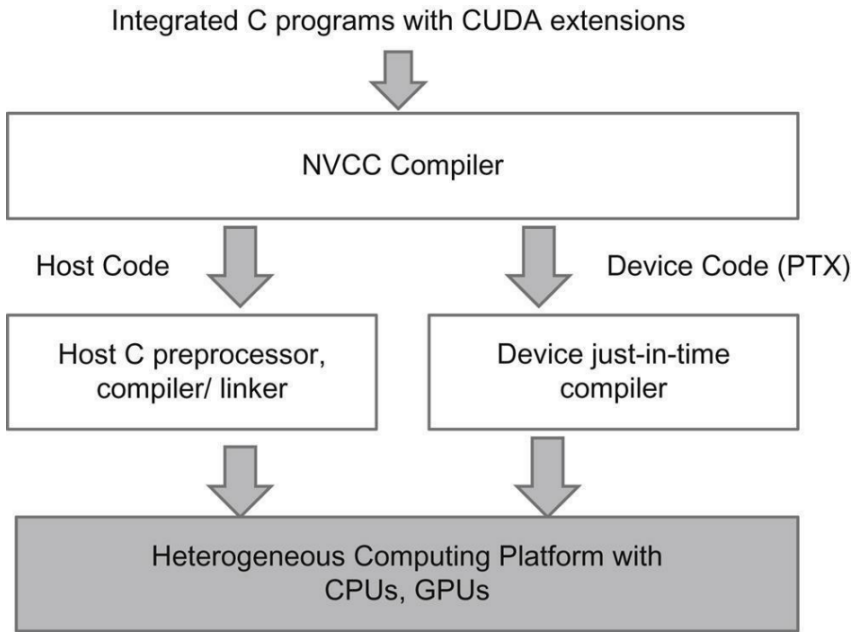
$$S_{latency}(s) = \frac{1}{(1-p) + \frac{p}{s}}$$

  - $S_{latency}$ is the theoretical speedup
  - p is the percentage of the work we can parallelize
  - s is how much parallelization improves the running time



By Daniels220 at English Wikipedia, CC BY-SA 3.0

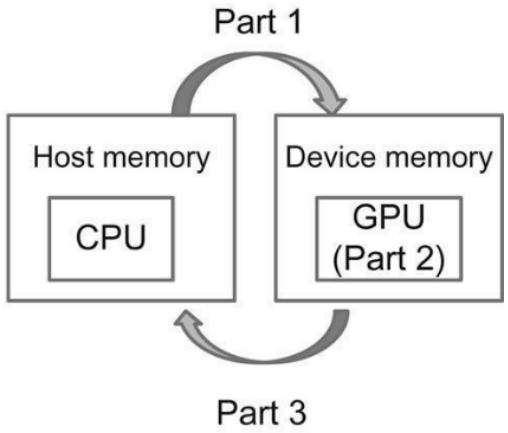
CUDA compilation process



- The NVCC compiler splits code into two parts:
  - Host code (CPU)
  - Device code (GPU)
- Regular C programs are also CUDA C compatible (why?)
  - They only have host code

CUDA example: vector addition

```
#include <cuda.h>
…
void vecAdd(float* A, float* B, float* C, int n)
{
  int  size = n* sizeof(float);
  float  *d_A *d_B, *d_C;
  …
1. // Allocate device memory for A, B, and C
   // copy A and B to device memory

2. // Kernel launch code – to have the device
   // to perform the actual vector addition

3. // copy C from the device memory
   // Free device vectors
}
```



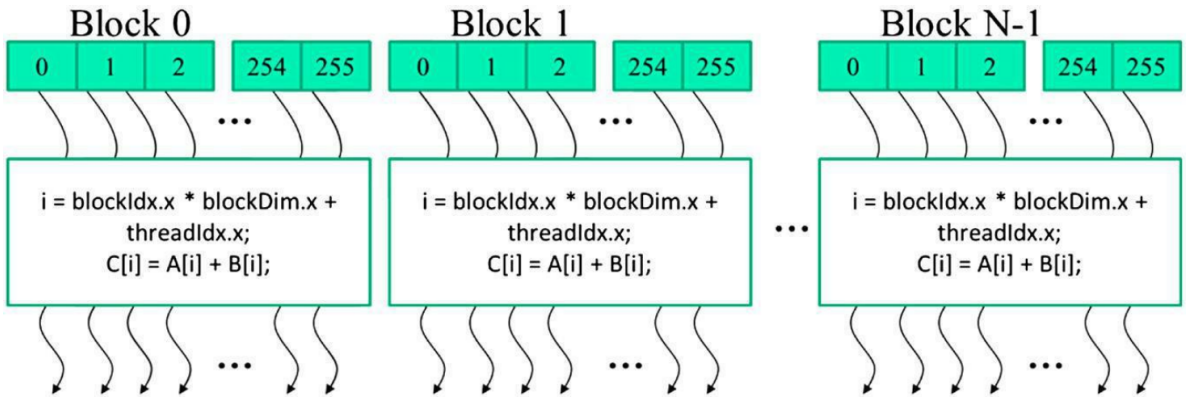
- Above is a sketch of the CUDA version
- We have the additional steps of **transferring the data to and from the GPU**

CUDA example: vector addition

```
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
    int size = n * sizeof(float);
    float *d_A, *d_B, *d_C;

    cudaMalloc((void **) &d_A, size);
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &d_B, size);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    cudaMalloc((void **) &d_C, size);

    // Kernel invocation code - to be shown later
    ...

    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Free device memory for A, B, C
    cudaFree(d_A); cudaFree(d_B); cudaFree (d_C);
}
```

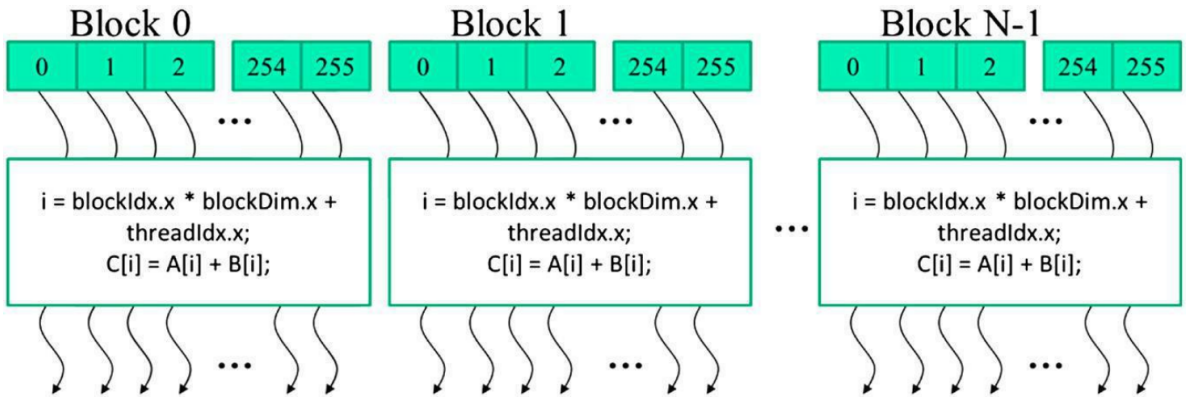
- Here is a more filled-in version of the CUDA code
- The built-in functions allow the CPU and GPU to talk to each other

Grid organization



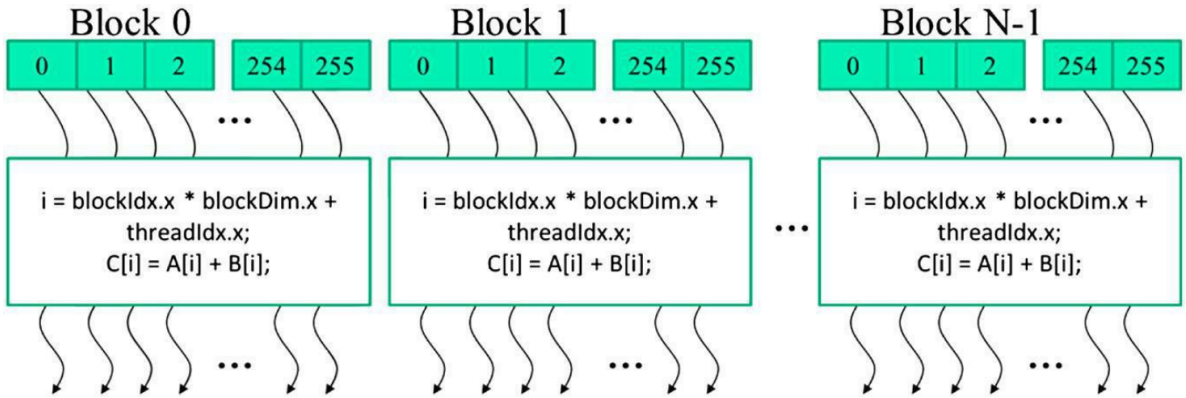
- Threads are organized in a two-level hierarchy
  - N **blocks**, each with k **threads** (usually a multiple of 32)
- You access individual threads using two indices, akin to a phone number
  - Block IDs are like area codes
  - Thread IDs like the local number

Grid organization



- **blockIdx**, **blockDim**, and **threadIdx** are *global, built-in* variables that allow you to index any thread
  - CUDA initializes the necessary values when you call a kernel function
  - Global variables are accessible to all functions in a program
  - You should never modify built-in variables yourself

Grid organization



- Blocks and threads can have up to 3 dimensions
  - $[x, y, z]$ coordinates in 3D graphics
- The number of dimensions should reflect the nature of the data
  - 1D for text, 2D for images, 3D for video, etc.
- Higher-dimensional data has to be handled indirectly
  - e.g., 4D data can be represented by w 3D blocks

A kernel function

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__
void vecAddKernel(float* A, float* B, float* C, int n)
{
   int i = blockDim.x*blockIdx.x + threadIdx.x;
   if(i<n) C[i] = A[i] + B[i];
}
```

- You define a kernel function for a generic processor
  - Similar to the pseudocode from the beginning of the course
  - **Not** the parallel for-loop style
- This kernel function receives **pointers** to the three arrays A, B, and C
- It uses the built-in variables to find the correct index
- The **if**-statement ensures that only processors with valid data will execute this function

Keyword extensions and grid sizes

	Executed on the:	Only callable from the:
`__device__ float DeviceFunc()`	device	device
`__global__ void KernelFunc()`	device	host
`__host__ float HostFunc()`	host	host

- CUDA C extends regular C with three keywords
  - They specify who can call which functions

```
int vectAdd(float* A, float* B, float* C, int n)
{
//  d_A, d_B, d_C allocations and copies omitted
// Run ceil(n/256) blocks of 256 threads each
    vecAddKernel<<<ceil(n/256.0), 256>>>(d_A, d_B, d_C, n);
}
```

- We specify the number of blocks and threads-per-block by the values within the three <> brackets
- Note that we specify these values when we *call* the function, not when we *declare* it

Final vector addition code

```
void vecAdd(float* A, float* B, float* C, int n)
{

  int size = n * sizeof(float);
  float *d_A, *d_B, *d_C;

  cudaMalloc((void **) &d_A, size);
  cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
  cudaMalloc((void **) &d_B, size);
  cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice);

  cudaMalloc((void **) &d_C, size);

  vecAddKernel<<<ceil(n/256.0), 256>>>(d_A, d_B, d_C, n);

  cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);

        // Free device memory for A, B, C
  cudaFree(d_A); cudaFree(d_B); cudaFree (d_C);
}
```

- The number of threads-per-block is fixed
- The number of blocks is a function of the vector length n
- Blocks can be executed *in any order*
  - The GPU's specs define how many blocks can run in parallel

Multidimensional threads

- Blocks and threads can have up to three dimensions
- In the most general case, we declare the block and thread sizes as **dim3** structs
  - **dim3** is a CUDA-defined type
- For example:
  - `dim3 dimGrid(32,28,64); dim3 dimBlock(8,16,4);`
  - `vecAddKernel<<<dimGrid,dimBlock>>>(...)`
- The three dimensions can vary in size
- If our data is one dimensional, we can either use ones (`dim3 dimGrid(32,1,1)`) or a scalar (`int dimGrid 32`)

Grid and block organization

- The number of blocks along a dimension is limited to 65,536
- All threads in a block share the same **blockIdx.x**, **blockIdx.y**, and **blockIdx.z** values
- The total number of threads in a block is currently limited to 1024
  - The threads can be distributed in any way
  - As long as the total number does not exceed 1024
  - e.g., `dimBlock(512,1,1)` is valid; `dimBlock(32,32,2)` is not
- The CUDA built-in variables **gridDim** and **blockDim** store the numbers of blocks and threads, per dimension. For example:
  - `vecAddKernel<<<dim3 dimGrid(32,16,4),dim3 dimBlock(128,2,2)>>>(...)`

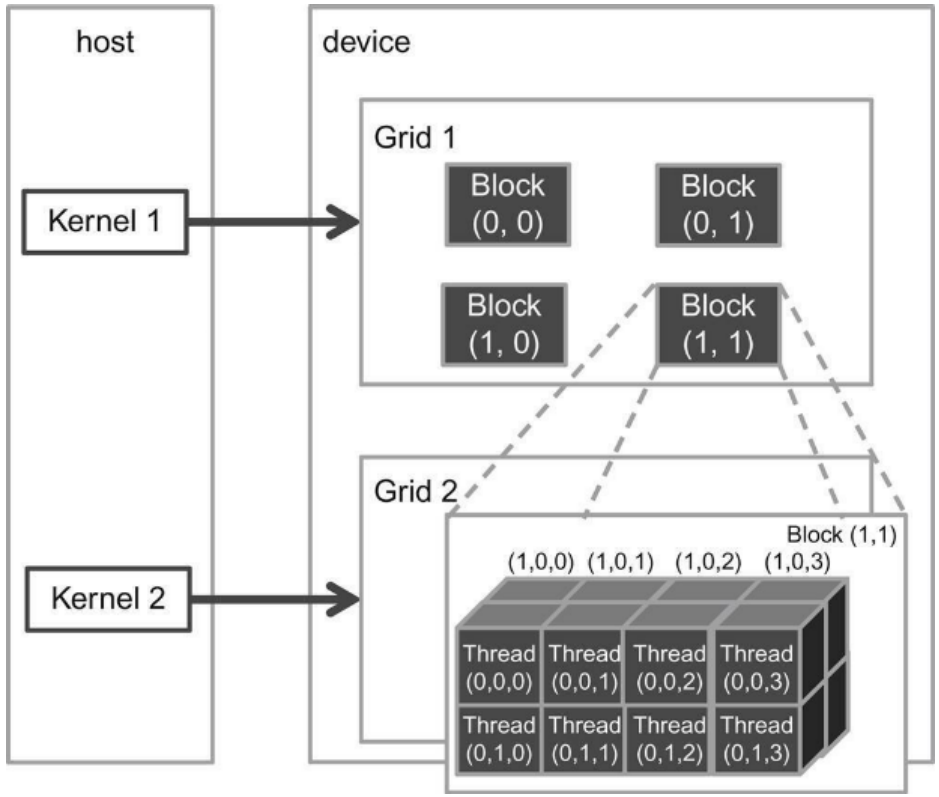
Grid and block organization

- The number of blocks along a dimension is limited to 65,536
- All threads in a block share the same **blockIdx.x**, **blockIdx.y**, and **blockIdx.z** values
- The total number of threads in a block is currently limited to 1024
  - The threads can be distributed in any way
  - As long as the total number does not exceed 1024
  - e.g., `dimBlock(512,1,1)` is valid; `dimBlock(32,32,2)` is not
- The CUDA built-in variables **gridDim** and **blockDim** store the numbers of blocks and threads, per dimension. For example:
  - `vecAddKernel<<<dim3 dimGrid(32,16,4),dim3 dimBlock(128,2,2)>>>(...)`
  - **gridDim.x** = 32, **gridDim.y** = 16, and **gridDim.z** = 4

Grid and block organization

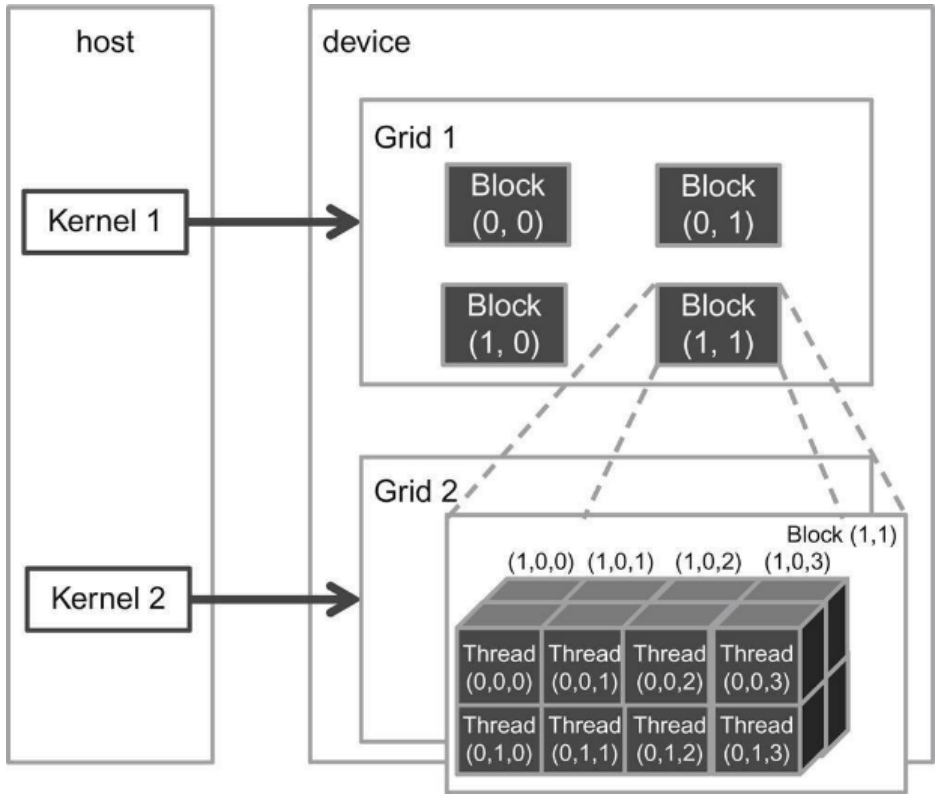
- The number of blocks along a dimension is limited to 65,536
- All threads in a block share the same **blockIdx.x**, **blockIdx.y**, and **blockIdx.z** values
- The total number of threads in a block is currently limited to 1024
  - The threads can be distributed in any way
  - As long as the total number does not exceed 1024
  - e.g., `dimBlock(512,1,1)` is valid; `dimBlock(32,32,2)` is not
- The CUDA built-in variables **gridDim** and **blockDim** store the numbers of blocks and threads, per dimension. For example:
  - `vecAddKernel<<<dim3 dimGrid(32,16,4),dim3 dimBlock(128,2,2)>>>(...)`
  - **gridDim.x** = 32, **gridDim.y** = 16, and **gridDim.z** = 4
  - **blockDim.x** = 128, **blockDim.y** = 2, and **blockDim.z** = 2

Grid and block organization



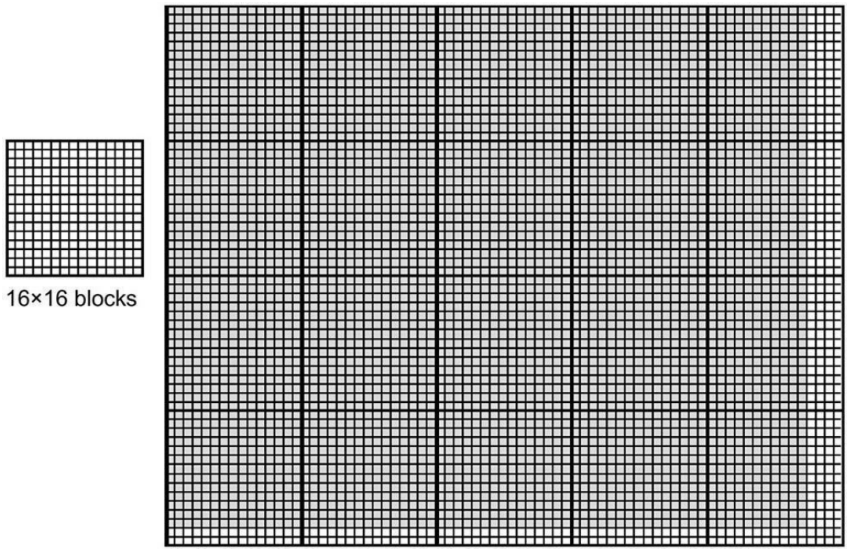
- Grid and blocks need not have the same dimensionality
- Above, we have 2D grids with 3D blocks

Grid and block organization



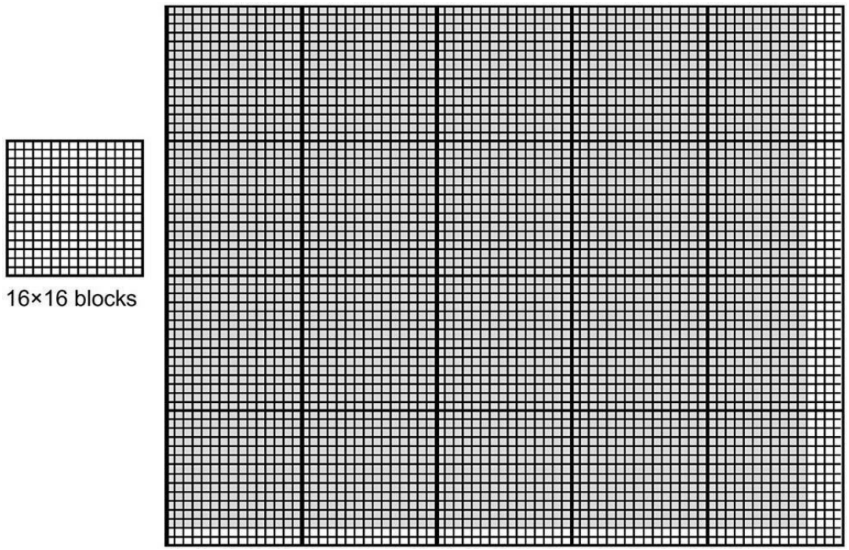
- Grid and blocks need not have the same dimensionality
- Above, we have 2D grids with 3D blocks
- The code is `dim3 gridDim(2,2,1)` and `blockDim(4,2,2)`

Mapping grids to data



- The choice of dimensionality is driven by the data
- e.g., images are naturally 2D
- Above, we are using 5×4, 16×16 blocks for 76×62 pixels
- The total number of threads is 80×64
- Some of the threads will have NULL data

Mapping grids to data



- The choice of dimensionality is driven by the data
- e.g., images are naturally 2D
- Above, we are using 5×4, 16×16 blocks for 76×62 pixels
- The total number of threads is 80×64
- Some of the threads will have NULL data
- `dim3 gridDim(5,4,1)` and `blockDim(16,16,1)`

Example: color-to-grayscale conversion

- Recall our initial example of converting a color image to grayscale
  - A color image is $n \times m \times 3$
  - Colors are stored as [red,green,blue] vectors
- If variables n and m are known, we can launch a kernel function as:
  - `dim3 dimGrid(ceil(m/16.0),ceil(n/16.0),1);`
  - `dim3 dimBlock(16,16,1);`
  - `colorToGrayscaleConversion<<<dimGrid, dimBlock>>>(d_Pin,d_Pout,m,n)`
- `d_Pin` and `d_Pout` are the pointers to the input and output arrays in the GPU