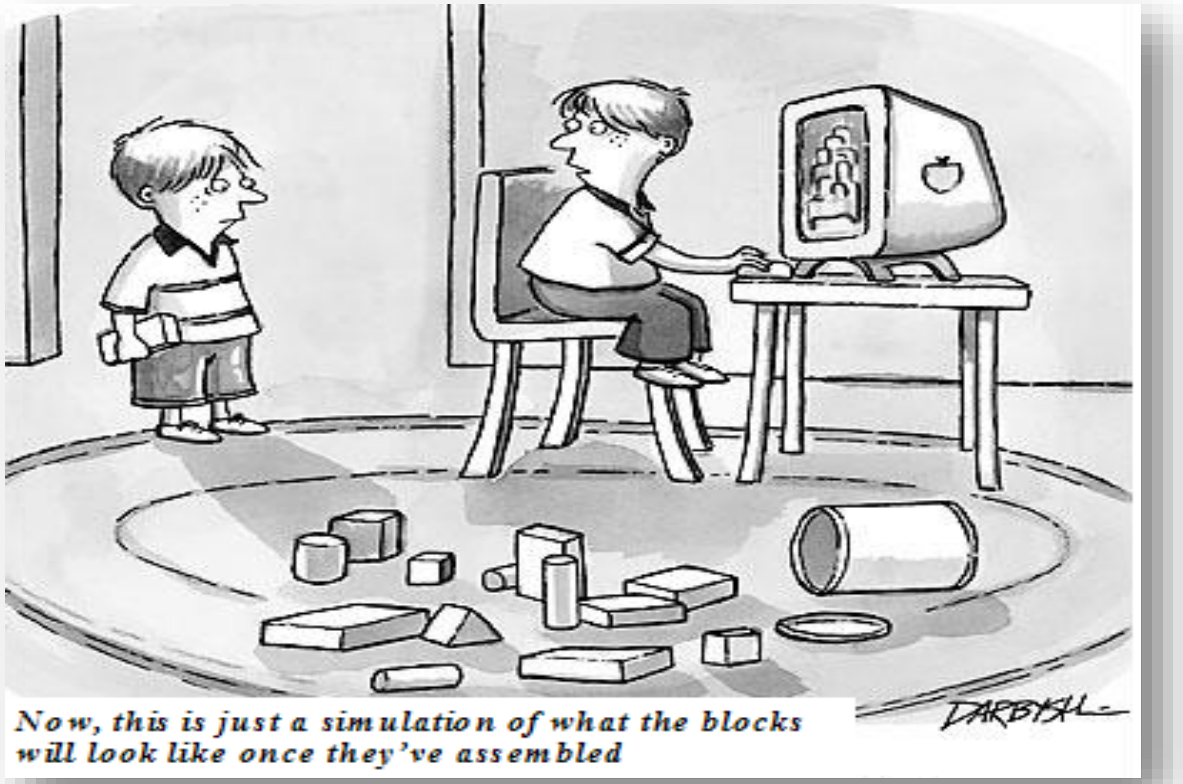
ECE569
Module 12

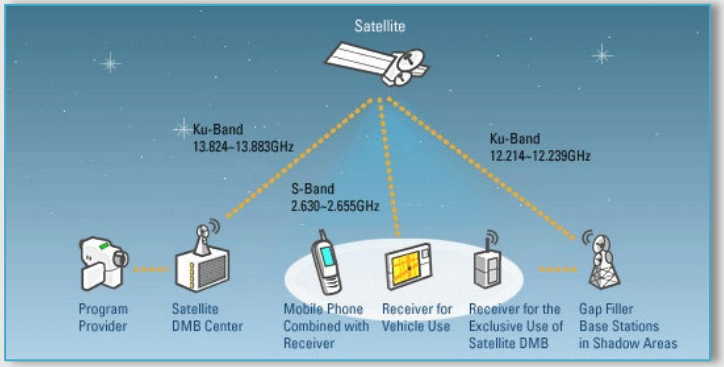


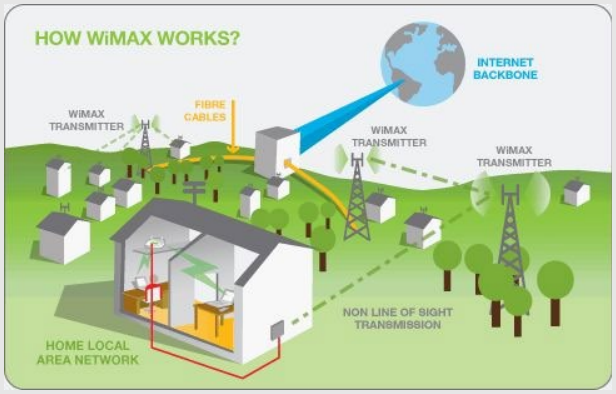
- Thread block organization

Accelerating Error Characterization for Communication Systems

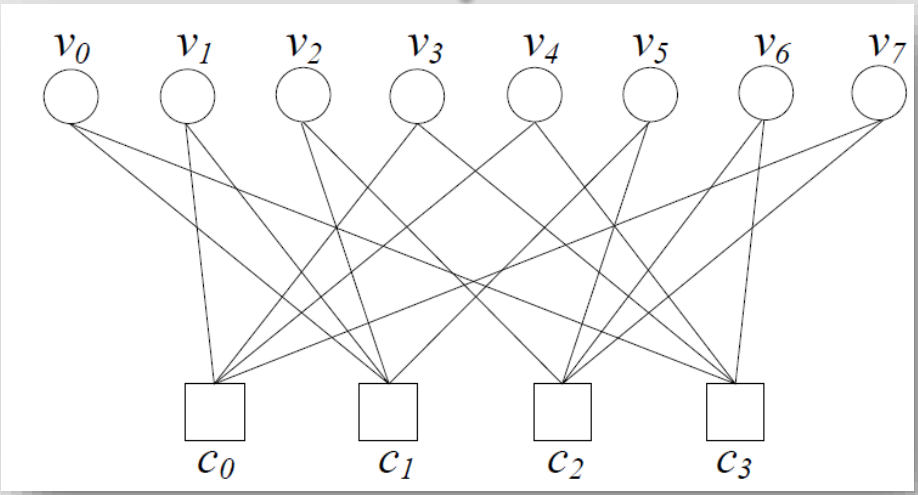
Error Correction

THE UNIVERSITY OF ARIZONA®





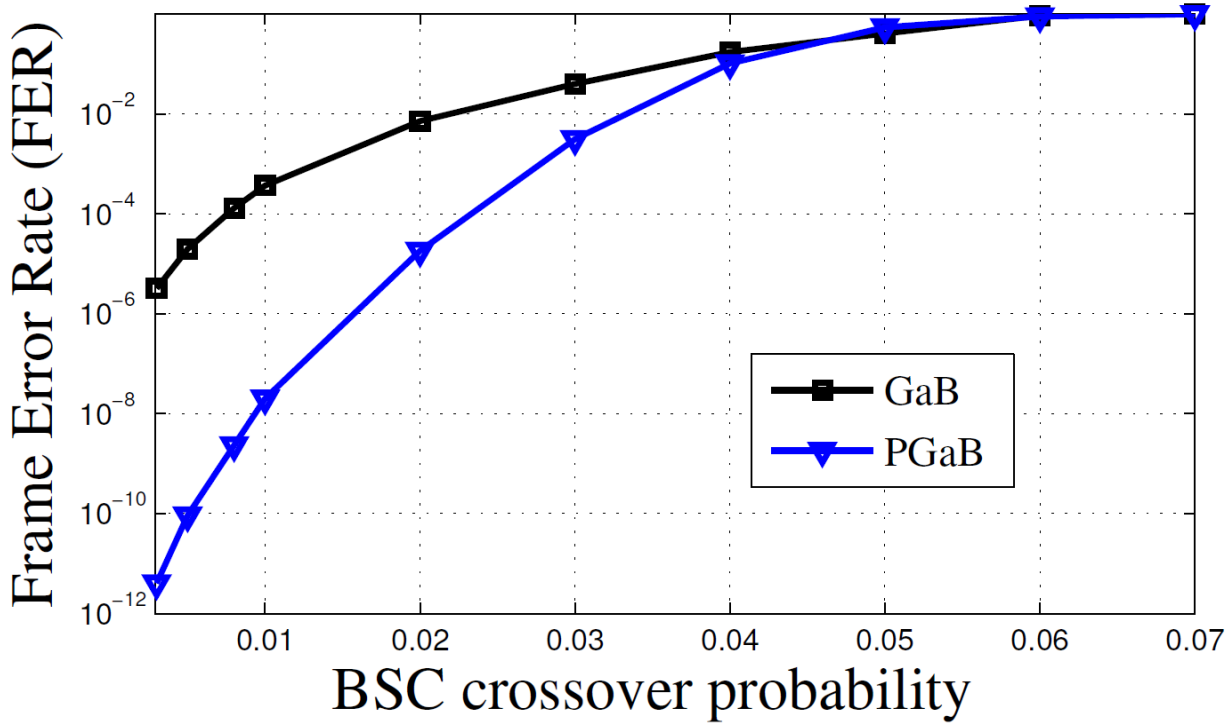
Simulation time
Scalability and
Hardware complexity

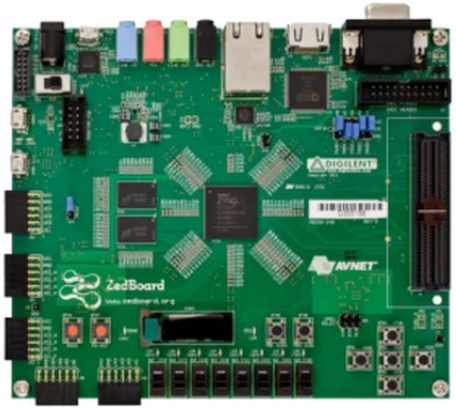


$$H = \begin{bmatrix} 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \end{bmatrix}$$

Processing all codewords

THE UNIVERSITY OF ARIZONA®





Alpha	0.01		0.005	
Enviroment	FPGA	PC	FPGA	PC
GaB	1 min<	2h' 27 min	1 min<	18h' 47 min
PGaB	4 min	116 days	24 hours	199 years

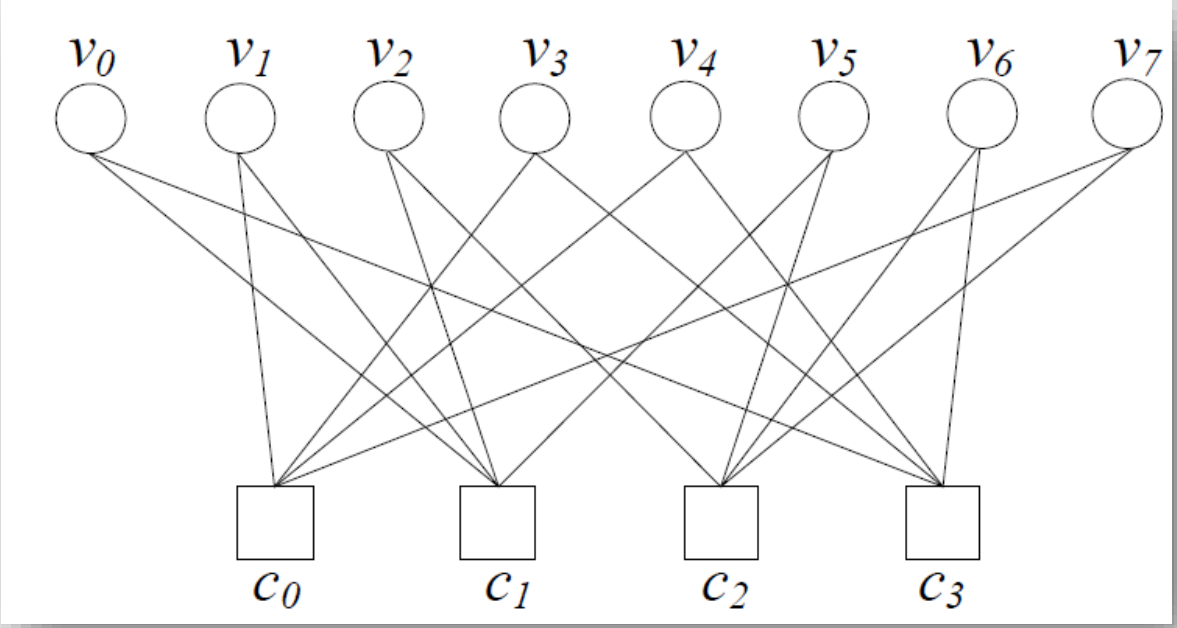
Error Pattern

- In order to evaluate all 4-bit error patterns for a codeword length of 1296, we need to test 117,002,820,060 codewords.

$$C(n,k) = \binom{n}{k} = \frac{n!}{(n-k)!k!}$$

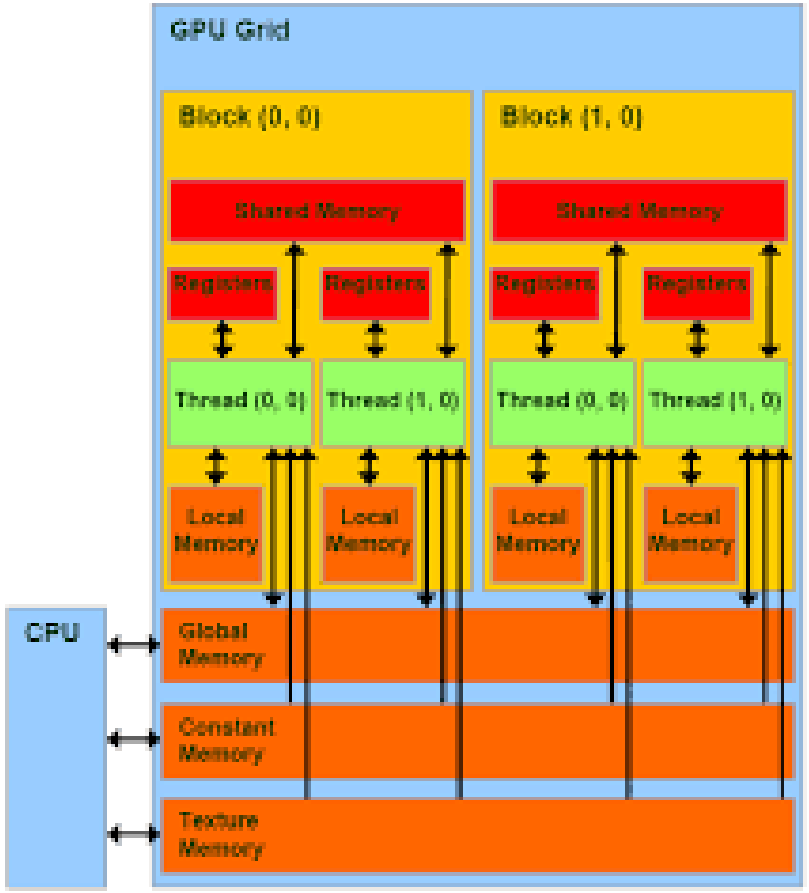
- Throughput on the Intel Xeon (2.33GHz, 8GB RAM) processor is 10,396 codewords per minute
  - Estimated to take **7,803 days.**
  - Same experiment on FPGA based testbed, and discovered 87 4-error patterns in **4.5 hours.**

- Parallelize on GPU
  - Optimization Challenge
    - Data access pattern



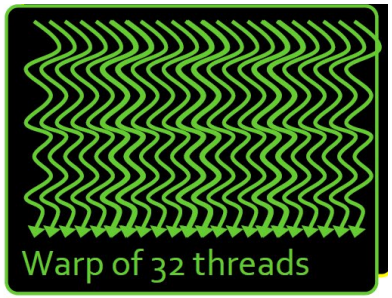
Memory Model

- **Thread**
  - Local memory
- **Threads in a Block**
  - Shared memory
- **Thread Blocks**
  - Global memory



Warps as Scheduling Units

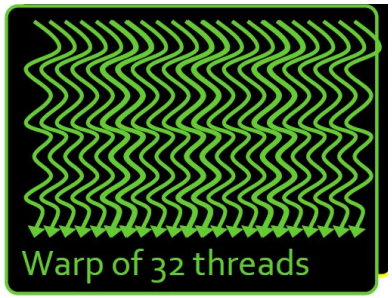
- **Each Block is executed as 32-thread Warps**
  - An implementation decision, not part of the CUDA programming model
  - Warps are scheduling units in SM
  - Threads in a warp execute in SIMD
  - Future GPUs may have different number of threads in each warp



- **If 3 blocks are assigned to an SM and each block has 256 threads, how many Warps are there in an SM?**

Warps as Scheduling Units

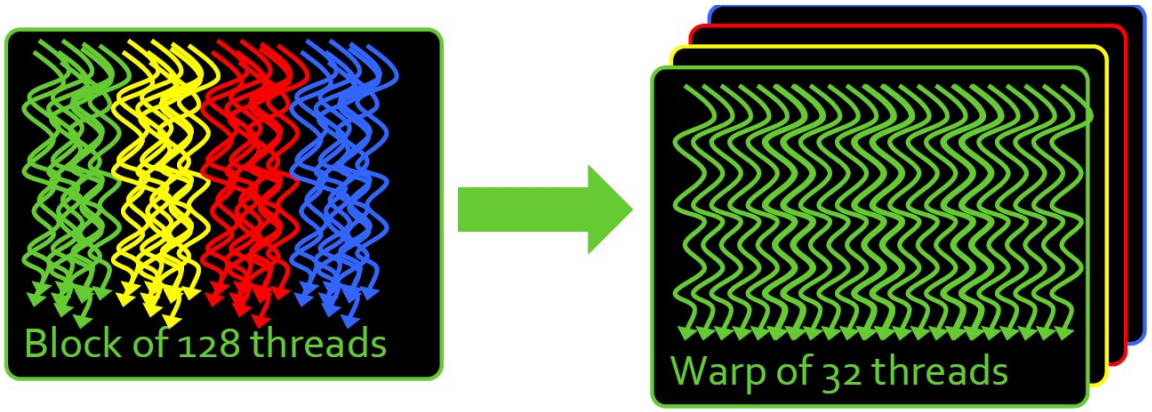
- **Each Block is executed as 32-thread Warps**
  - An implementation decision, not part of the CUDA programming model
  - Warps are scheduling units in SM
  - Threads in a warp execute in SIMD
  - Future GPUs may have different number of threads in each warp



- **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

Threads are Executed in Warps

- **Each thread block split into one or more warps**
  - When the thread block size is not a multiple of the warp size, unused threads within the last warp are disabled automatically
  - The hardware schedules each warp independently
  - Warps within a thread block can execute independently

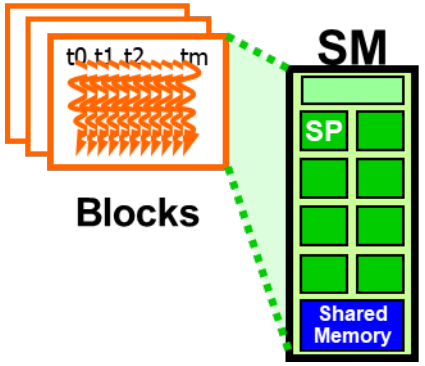


Execution Configuration

- **Prefer to have enough threads per block to provide hardware with many warps to switch between**
  - This is how the GPU hides memory access latency
- **Prefer thread block sizes that result in mostly full warps**
  - Bad: kernel<<<N, 1>>> (...)
  - Okay: kernel<<<(N+31) / 32, 32>>>(...)
  - Better: kernel<<<(N+127) / 128, 128>>>(...)

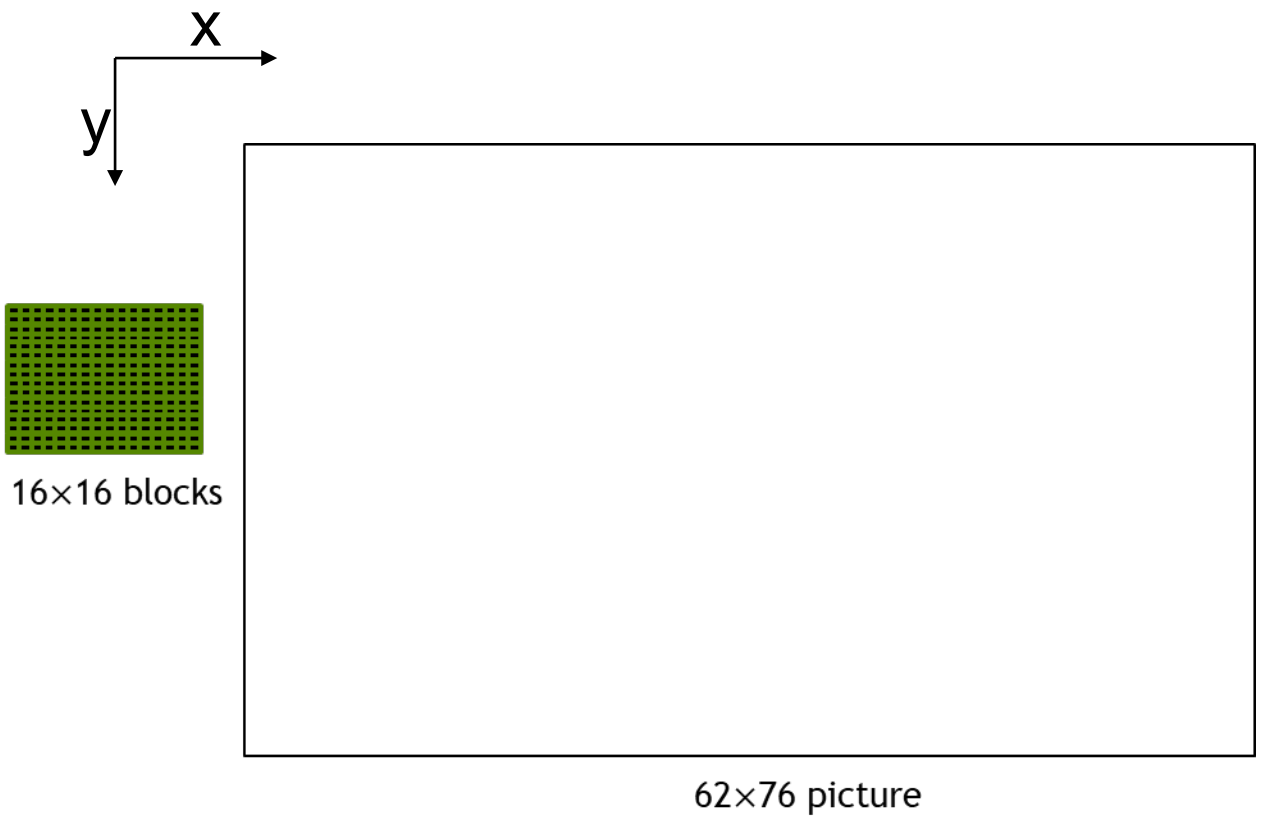
Thread Scheduling

- **SM manages/schedules thread execution**
- **SM implements zero-overhead warp scheduling**
  - Every clock cycle hardware monitors the operands of each instruction
  - 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



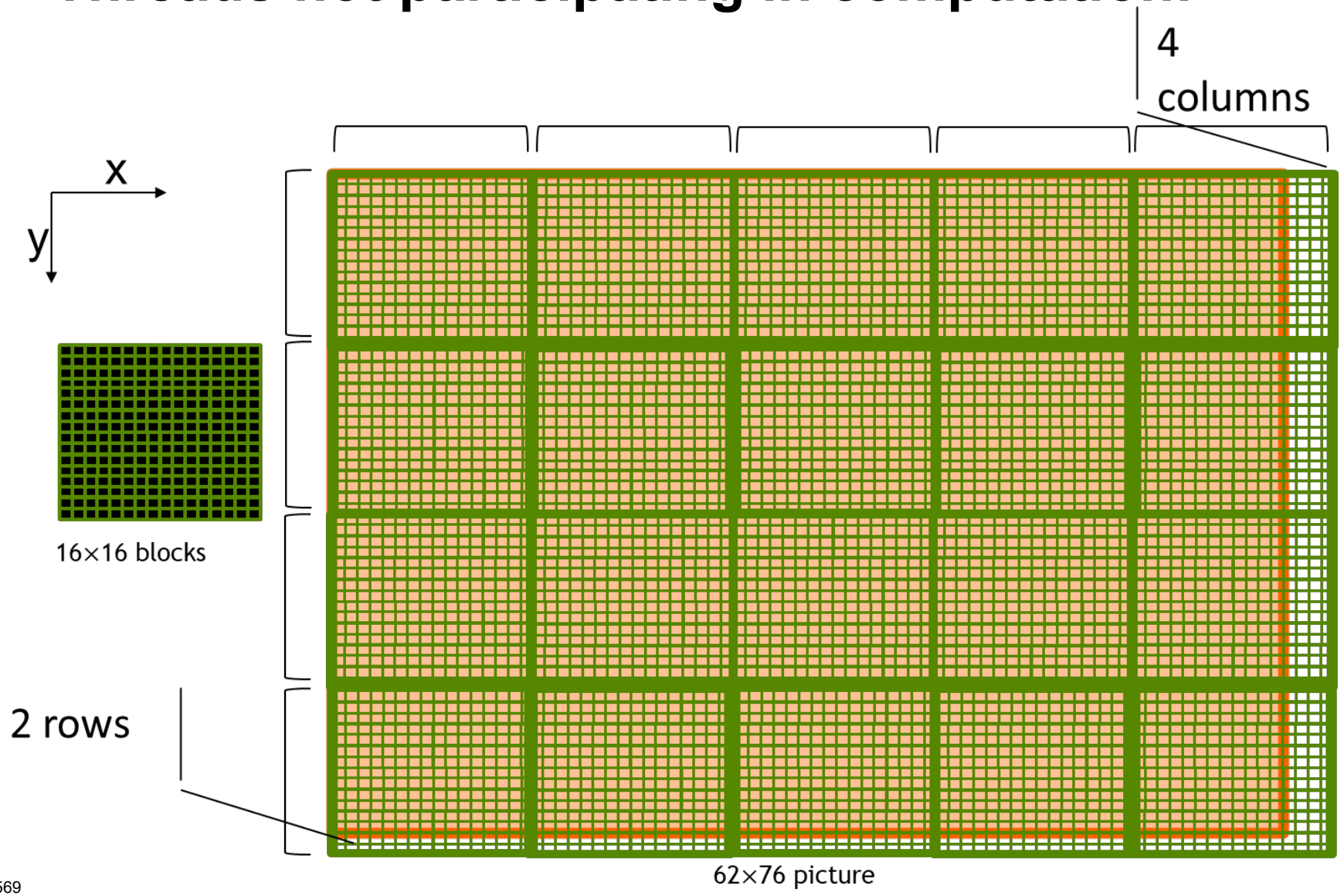
Processing a Picture with a 2D Grid

- **Block Configuration**

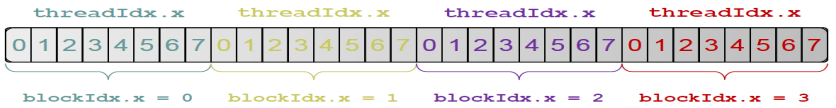


Processing a Picture with a 2D Grid

- **Threads not participating in computation!**



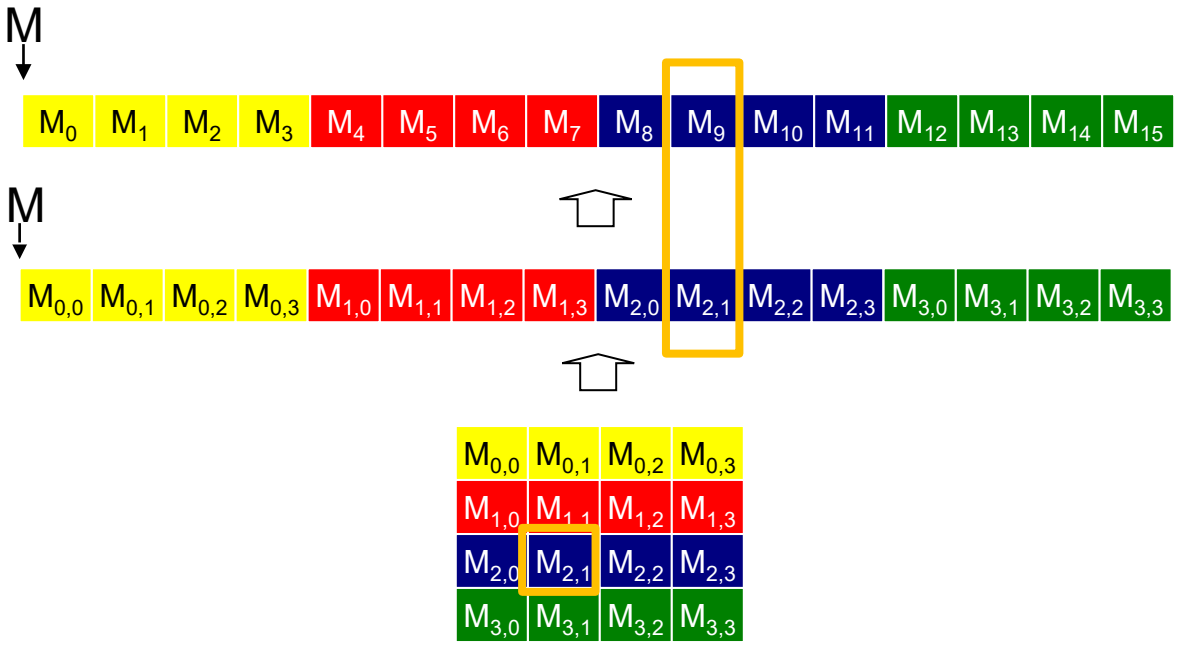
Row-Major Layout in C



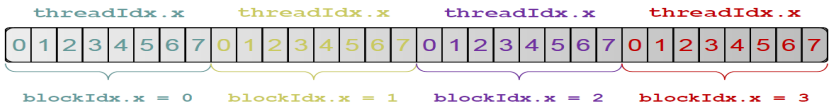
- **In 1D**
  - index = threadIdx.x + blockIdx.x * blockDim.x;
- **In 2D**

Expression for indexing?

(Image: Row, Column, Width, Height)



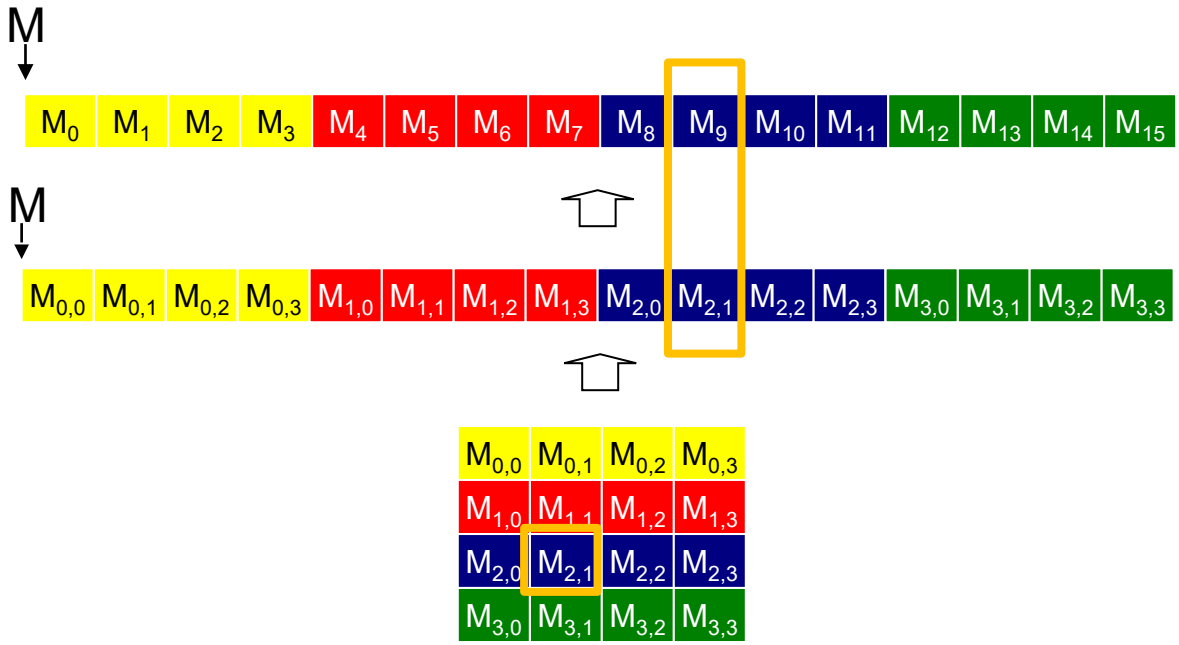
Row-Major Layout in C



- **In 1D**
  - index = threadIdx.x + blockIdx.x * blockDim.x;
- **In 2D**

Expression for indexing?

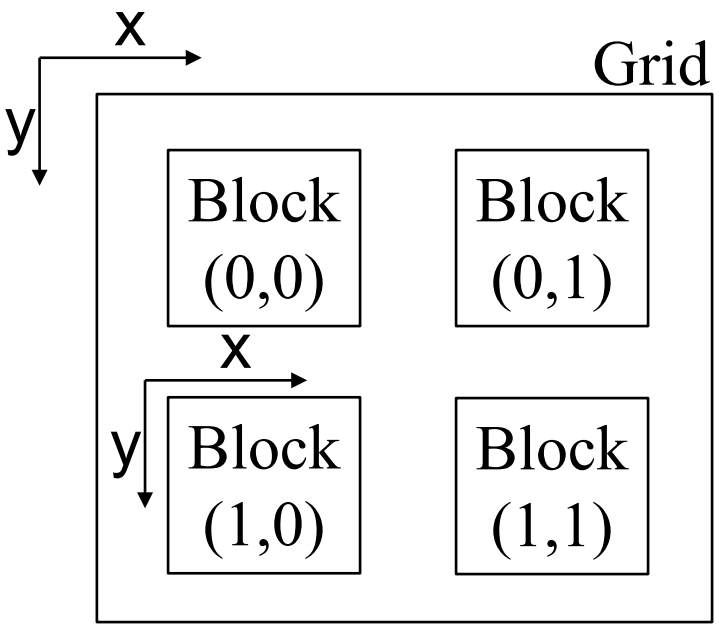
Row*Width+Col = 2*4+1 = 9



blockIdx

- grid with four blocks organized into a 2×2 array.
  - Each block is labeled with (blockIdx.y, blockIdx.x).
  - Block(1,0) has blockIdx.y=1 and blockIdx.x=0.
  - **notation is in reversed ordering of that used in the C statements**
    - **for ex: Block(0,1) is (1,0) in x,y coordinates**

Just to orient ourselves!



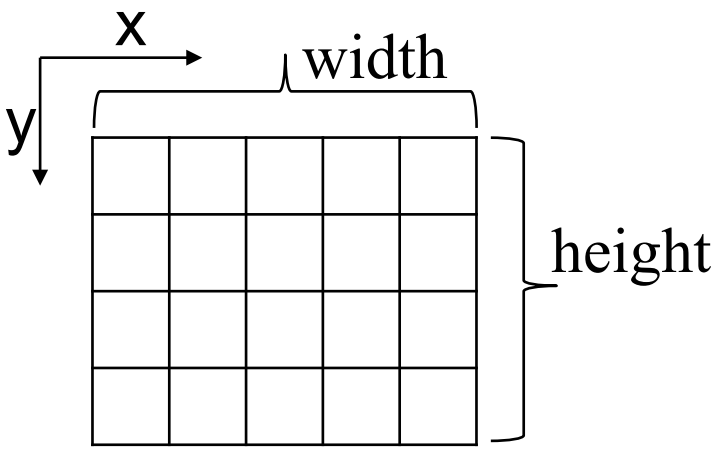
Mapping data to a single thread

- **In 1D:** `index = threadIdx.x + blockIdx.x * blockDim.x;`
- **In 2D**
  - **Row***Width+**Col**
  - X direction refers to position in a given row
    - row fixed, column number to pick
  - Y direction refers to position in a given column
    - Colum fixed, row number to pick

Iterating again

When you are changing the row number you are changing the threadIdx.y

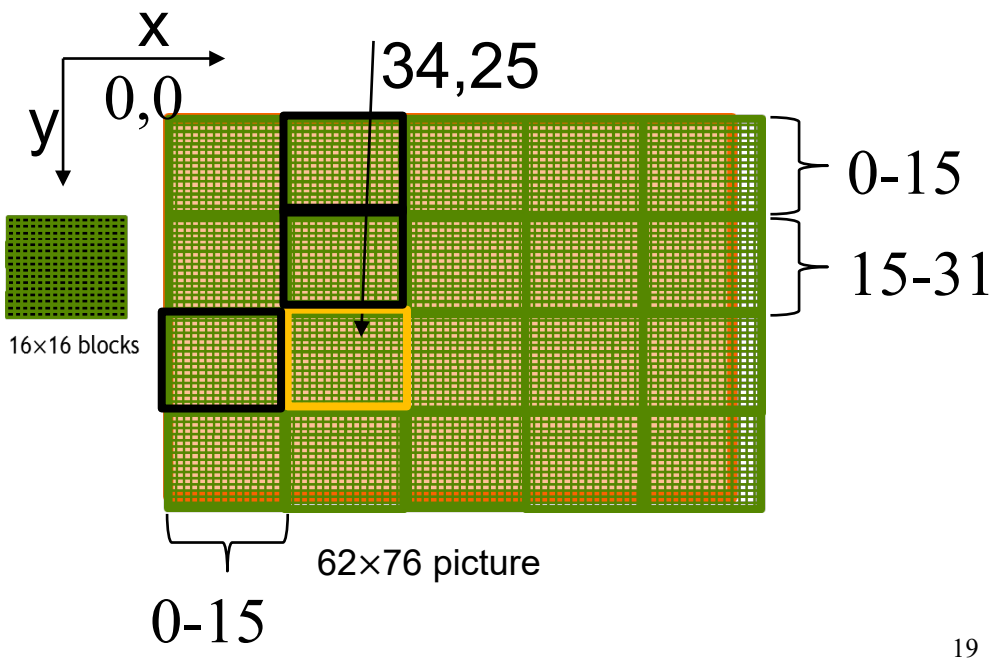
When you are moving in x direction, you are changing the column number (threadIdx.x)



PictureKernel: Row and Col mapping

//Scale every pixel value by 2.0

```
__global__ void PictureKernel(float* d_Pin, float* d_Pout,
                              int height, int width){
  //1D: index = threadIdx.x + blockIdx.x * blockDim.x;
  // Calculate the row # of the d_Pin and d_Pout element
  int Row = ????
  // Calculate the column # of the d_Pin and d_Pout element
  int Col = ????
  // each thread computes one element of d_Pout
```



}

PictureKernel: Now multiply each pixel by 2.0

```
__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



}
```

Scale every pixel value by 2.0

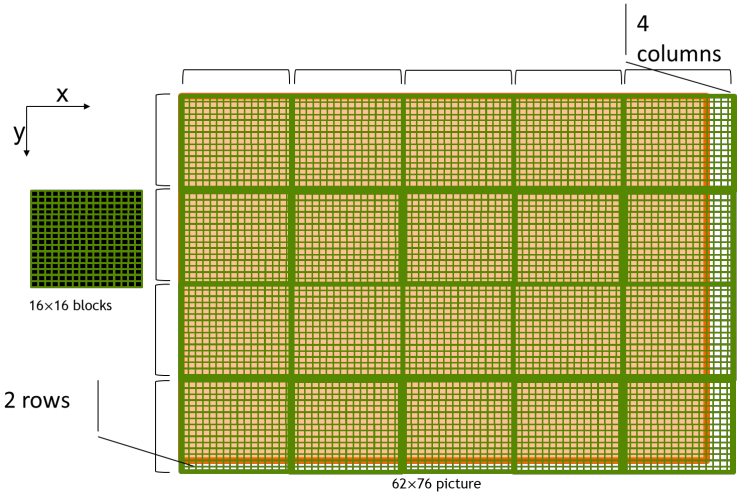
PictureKernel: Now multiply each pixel by 2.0

```
__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

    d_Pout[??????????] = 2.0*d_Pin[??????????];

}
```

Physical address as a function of Row and Col

PictureKernel: Now multiply each pixel by 2.0

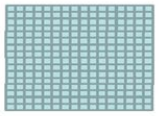
```
__global__ void PictureKernel(float* d_Pin, float* d_Pout,
                              int height, int width){

  // 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

    d_Pout[Row*width+Col] = 2.0*d_Pin[Row*width+Col];

}
```

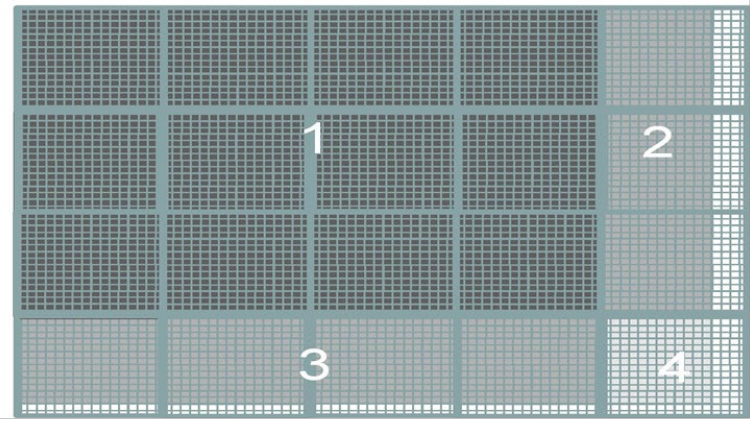


PictureKernel: How about boundary condition?

```
__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


    d_Pout[Row*width+Col] = 2.0*d_Pin[Row*width+Col];

}
```

76 pixels in x
62 pixels in y





idle threads
4*16*3

idle threads
2*16*4

idle threads
256-12*14

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

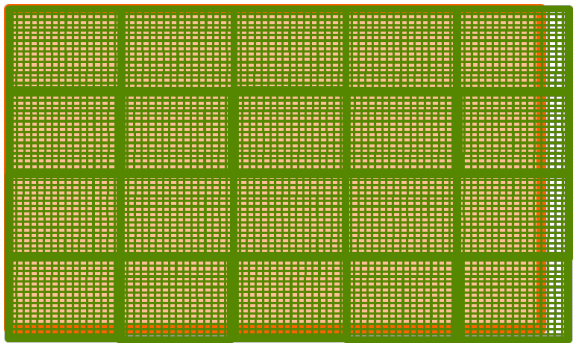
PictureKernel

```
__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 PictureKernel

```
// assume that the picture is m × n,
// m pixels in y dimension and n pixels in x dimension
// block size is 16x16
// input d_Pin has been allocated on and copied to device
// output d_Pout has been allocated on device
...
dim3 DimGrid((?-1)/16 + 1, (?-1)/16+1, 1);
dim3 DimBlock(16, 16, 1);
PictureKernel<<<DimGrid,DimBlock>>>(d_Pin, d_Pout, m, n);
...
```

Remember !
dim3 is a 3D structure (C `struct`)
with three unsigned integers (**x**,y,z)
x is first, y is second!!

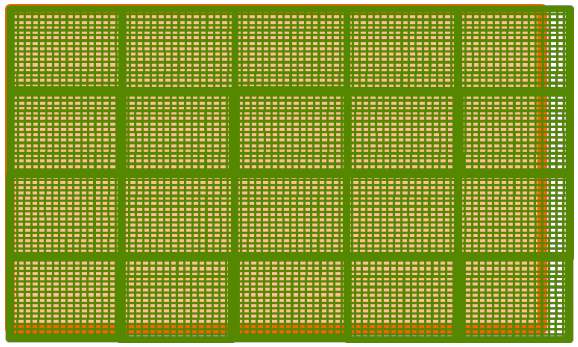


62×76 picture

Host Code for PictureKernel

```
// assume that the picture is m × n,
// m pixels in y dimension and n pixels in x dimension
// block size is 16x16
// 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);
...
```

Remember !
dim3 is a 3D structure (C `struct`)
with three unsigned integers (**x**,y,z)
x is first, y is second!!



62×76 picture

Extending to 3D arrays

- **We add planes in z direction**
  - Plane = blockIdx.z*blockDim.z +threadIdx.z
- **Linearized access to 3D array P**

Extending to 3D arrays

- **We add planes in z direction**
  - Plane = blockIdx.z*blockDim.z +threadIdx.z
- **Linearized access to 3D array P**
  - P[**Plane*m*n** + Row*m + Col]

Next

- **Image processing algorithms: Color space conversion**