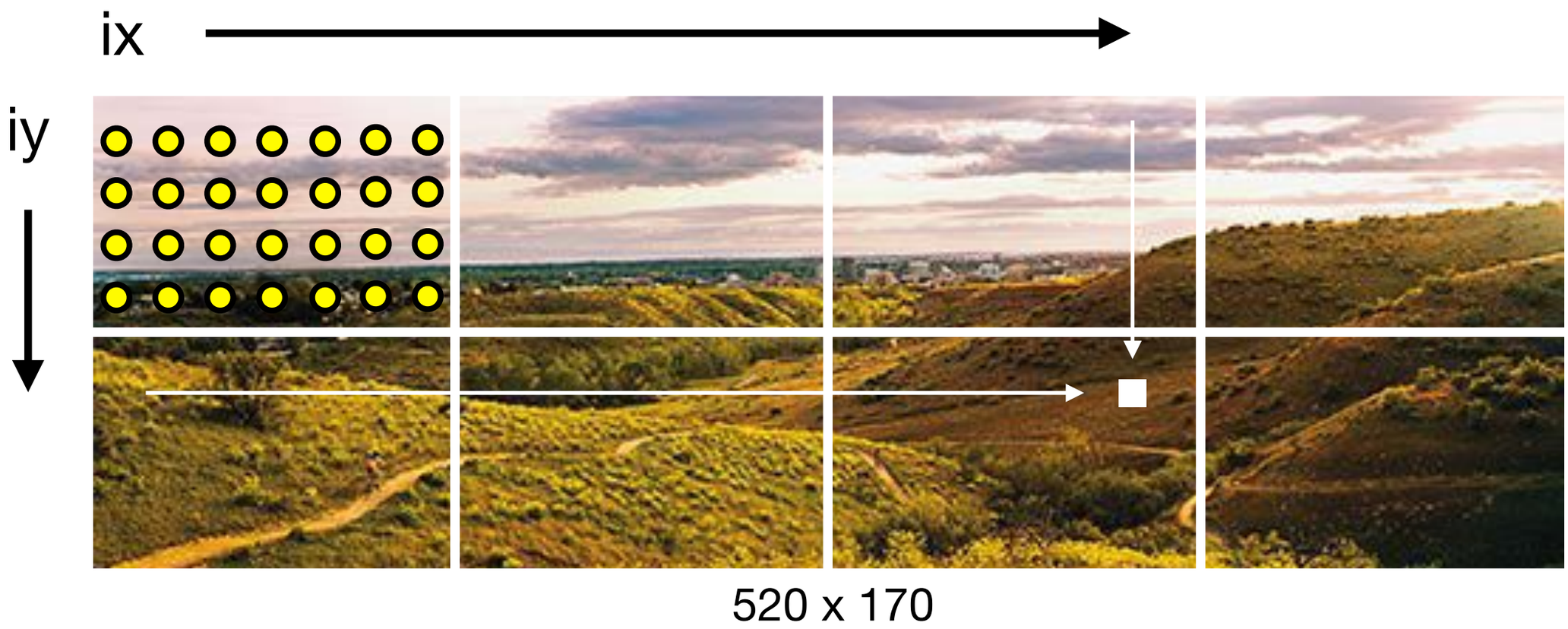
Images



520 x 170

It turns out to be more efficient to process image data in *blocks*.

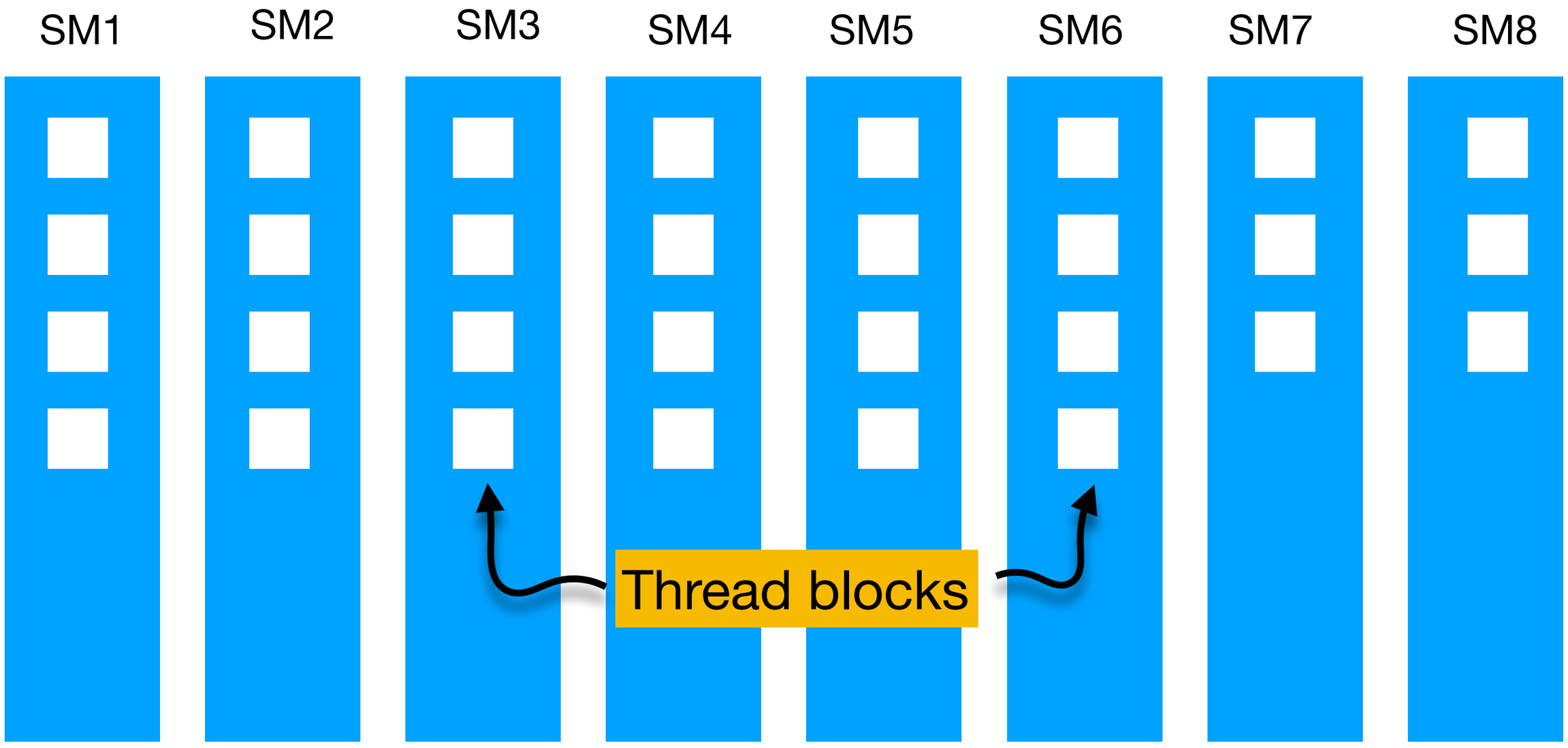
Threads within each block process one or more pixels (also arranged in blocks (e.g. *thread blocks).*

Individual pixels are associated with an index (ix,iy).

@Donna Calhoun, BSU 2018-2020

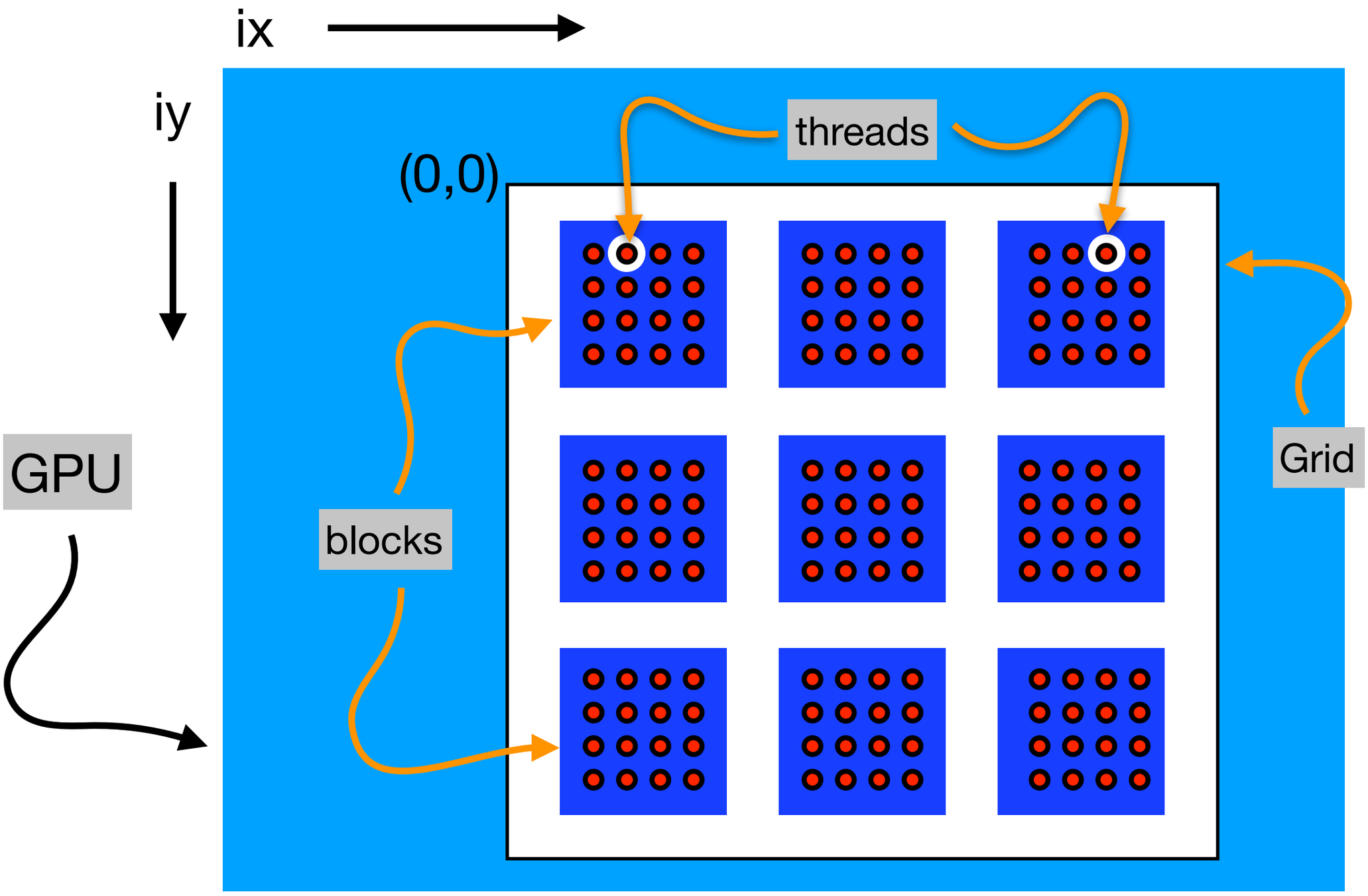
Streaming Multiprocessor

The heart of the GPU is the “Streaming Multiprocessor” or SM.



@Donna Calhoun, BSU 2018-2020

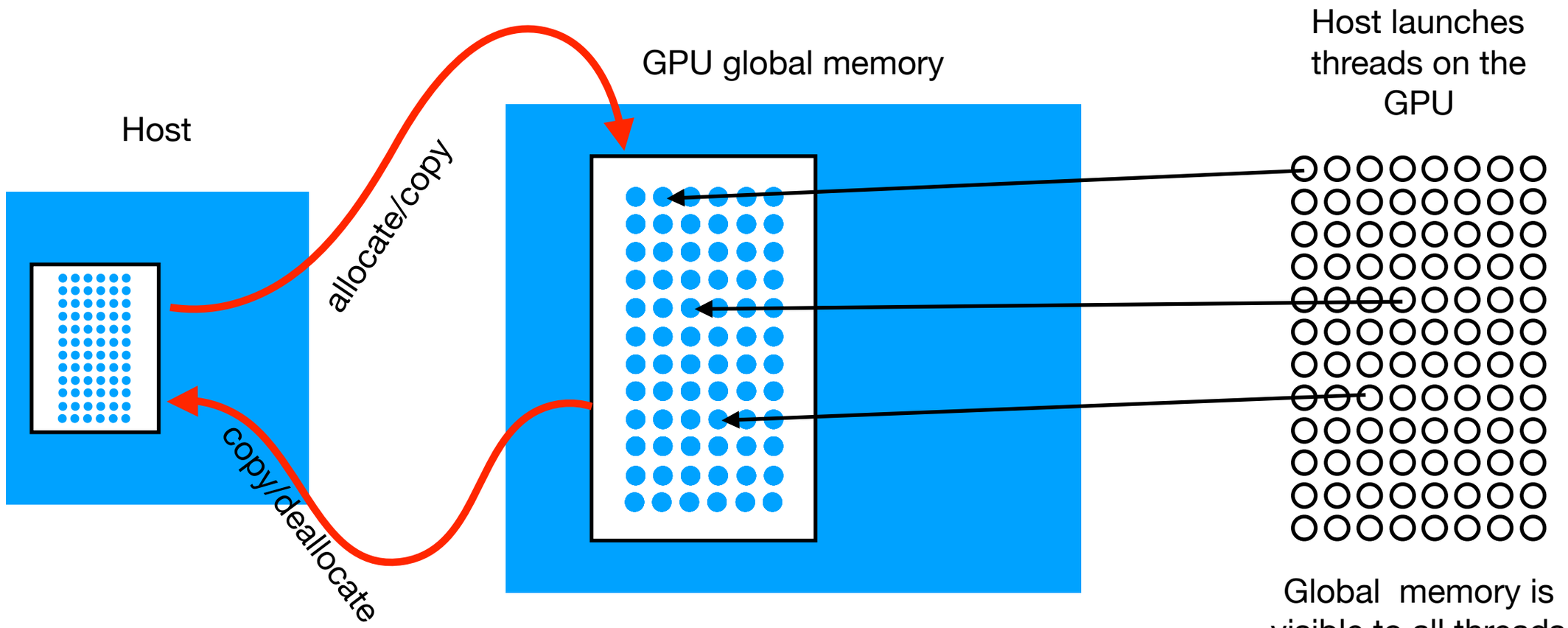
Grids, Blocks, Threads



Threads are launched on a GPU as grids, blocks and threads

CUDA Programming Model

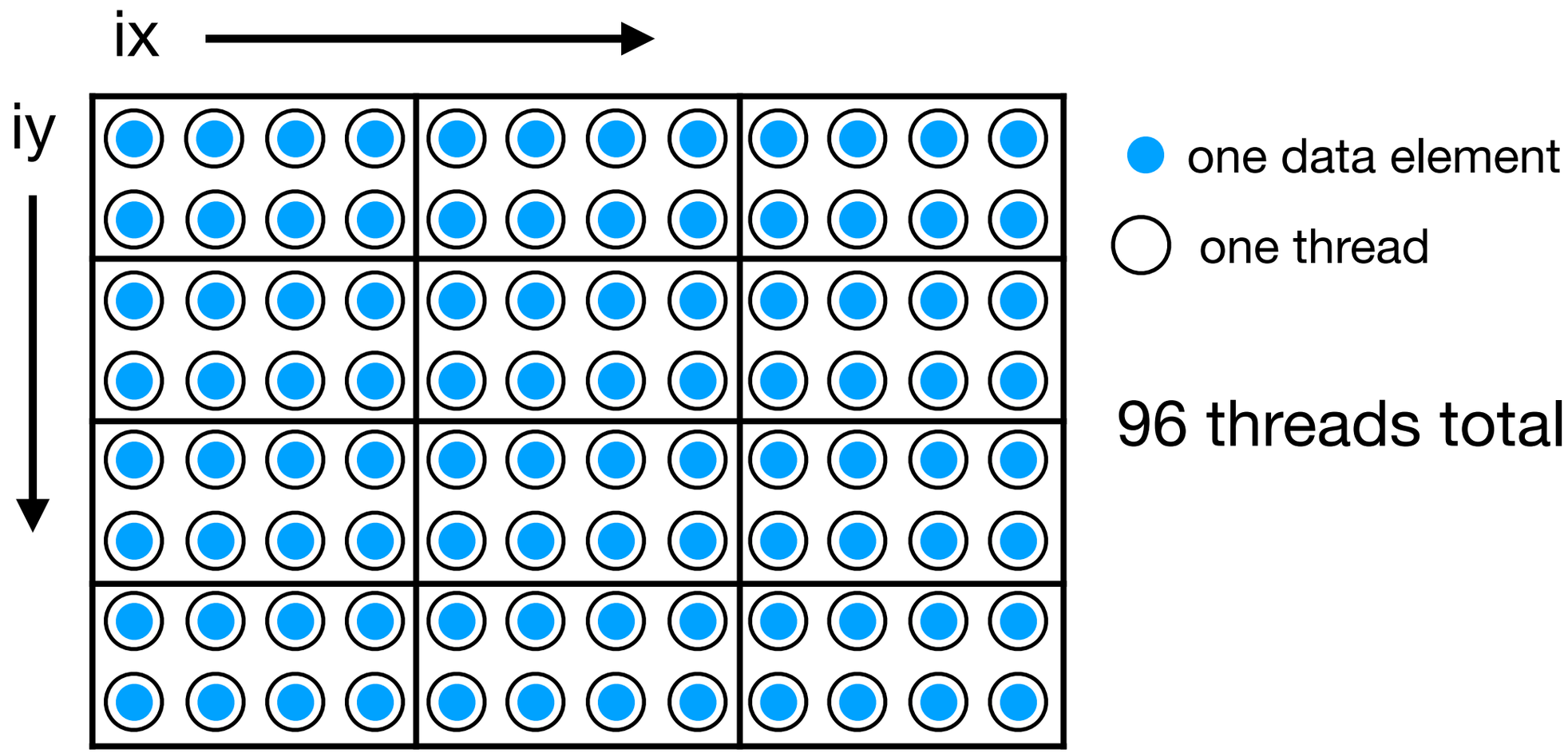
1. Allocate memory in the *global memory space* on the GPU
2. Copy any data from the host CPU to the global device memory
3. Launch an *array of threads* to work on this global memory.
4. Copy results back to host memory.



@Donna Calhoun, BSU 2018-2020

Mapping Blocks/Threads to Data

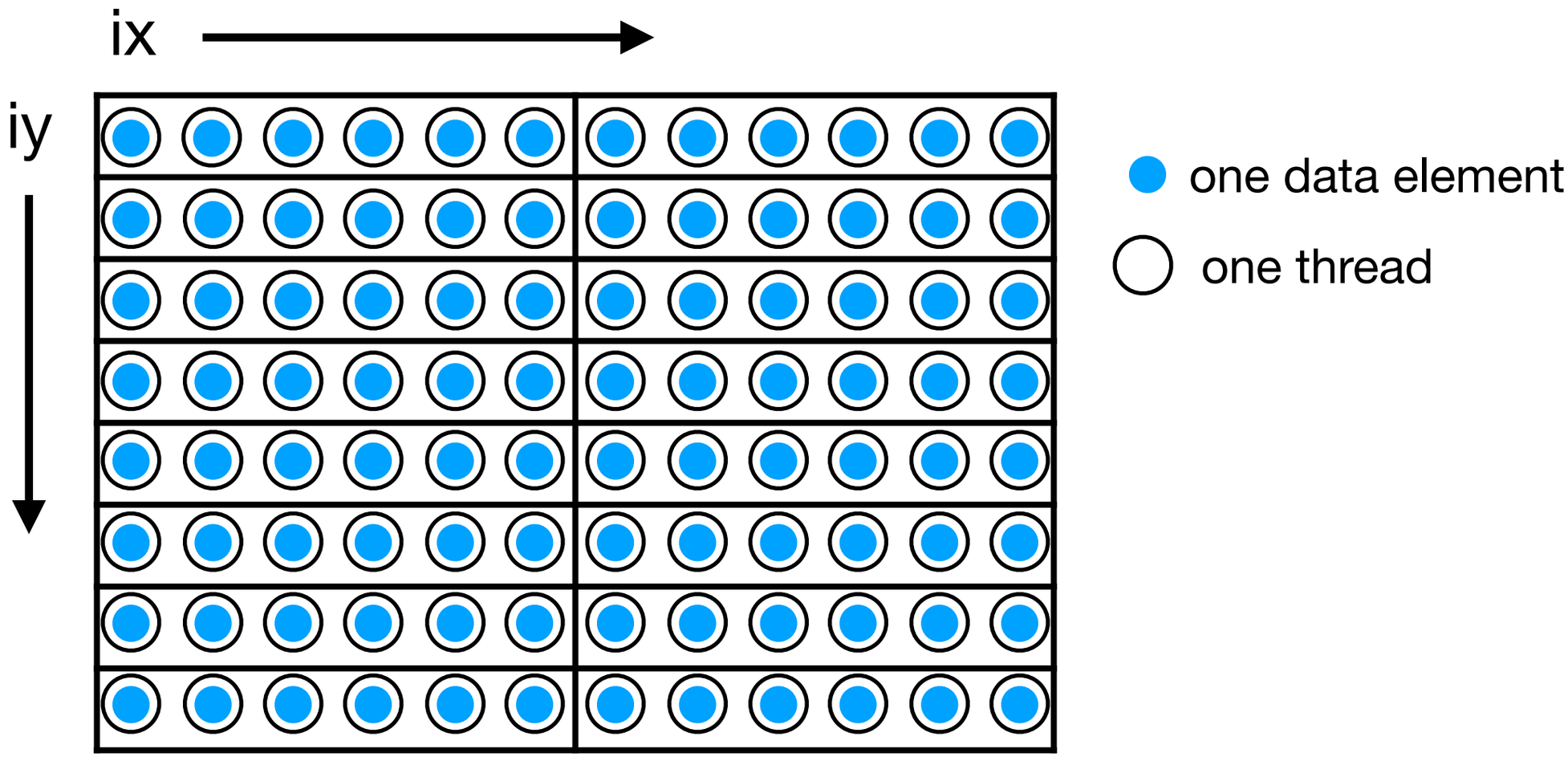
Launch exactly as many threads as there are data elements



Grid of 3 blocks in the x direction and 4 in the y direction

Blocks of 4 threads in the x direction and 2 in the y direction

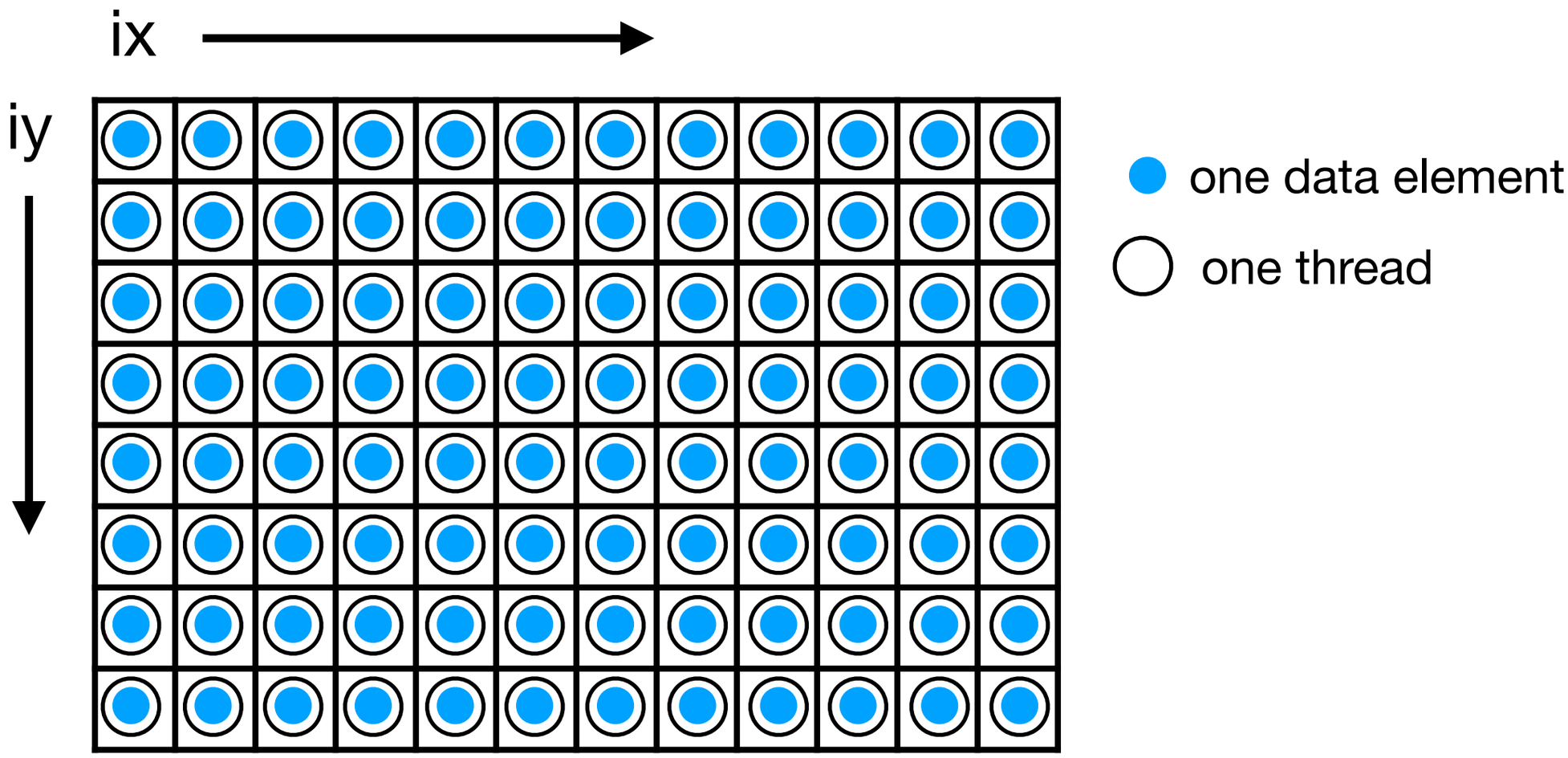
Mapping Data to Blocks/Threads



Grid of 2 blocks in the x direction and 8 in the y direction

Blocks of 6 threads in the x direction and 1 in the y direction

Mapping Blocks/Threads to Data



Grid of 12 blocks in the x direction and 8 in the y direction

Blocks of 1 thread in the x direction and 1 in the y direction

One thread per block

@Donna Calhoun, BSU 2018-2020

Matrix example

i ⟶ m=12

$$
\begin{array}{c}
\text{j} \\
\downarrow \\
\\
\\
\\
\text{n=8}
\end{array}
\begin{bmatrix}
a_{00} & a_{10} & a_{20} & \cdots & a_{11,0} \\
a_{01} & a_{11} & a_{21} & \cdots & a_{11,1} \\
a_{02} & a_{12} & a_{22} & \cdots & a_{11,2} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
a_{07} & a_{17} & a_{27} & \cdots & a_{11,7}
\end{bmatrix}
$$

Row and columns are swapped for consistency with GPU layout.

Use GPU to multiply entries of A by 2.

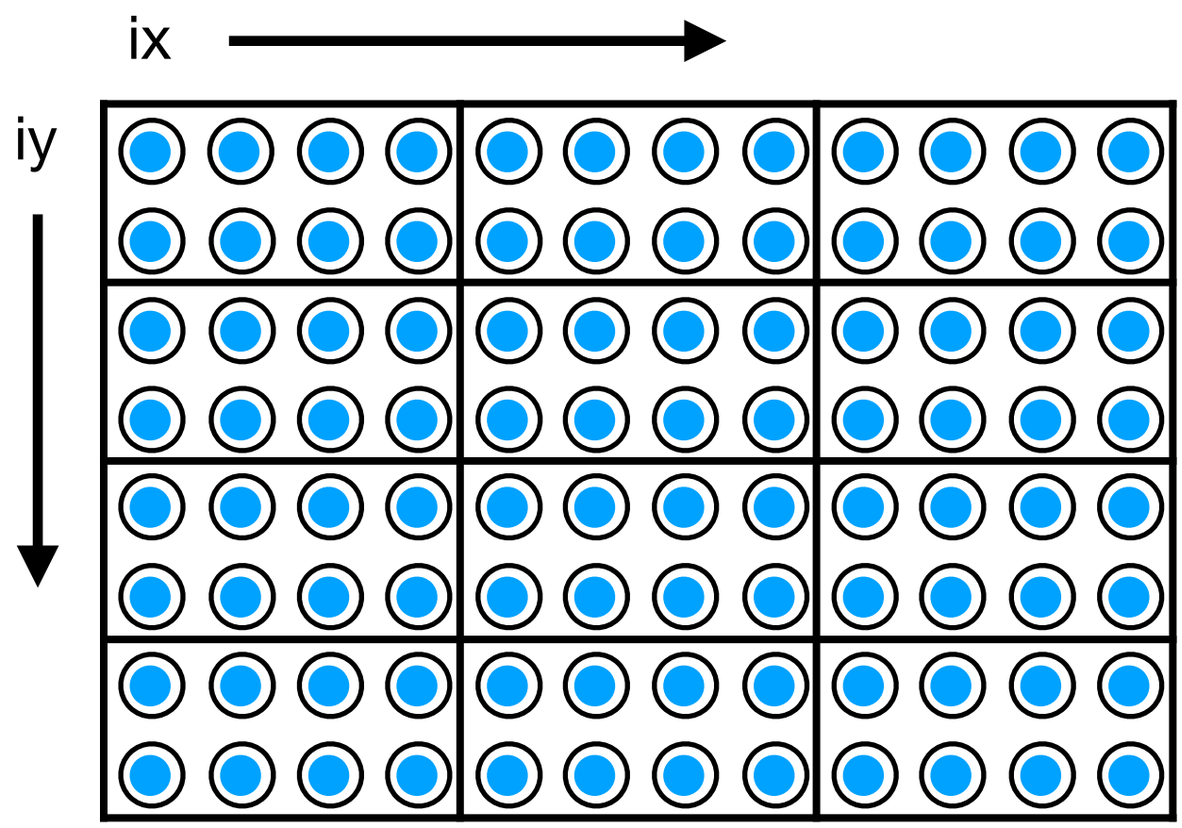
```
void main(void) {
    int *A, *dev_A;
    int m = 12,n = 8;
    int nbytes = m*n*sizeof(int);

    A = malloc(nbytes);
    for(int k=0; k < m*n; k++)
       A[k] = 1;

    cudaMalloc((void**) &dev_A, nbytes);
    cudaMemcpy(dev_A,A,nbytes,cudaMemcpyHostToDevice);
    ....
```

@Donna Calhoun, BSU 2018-2020

Matrix example



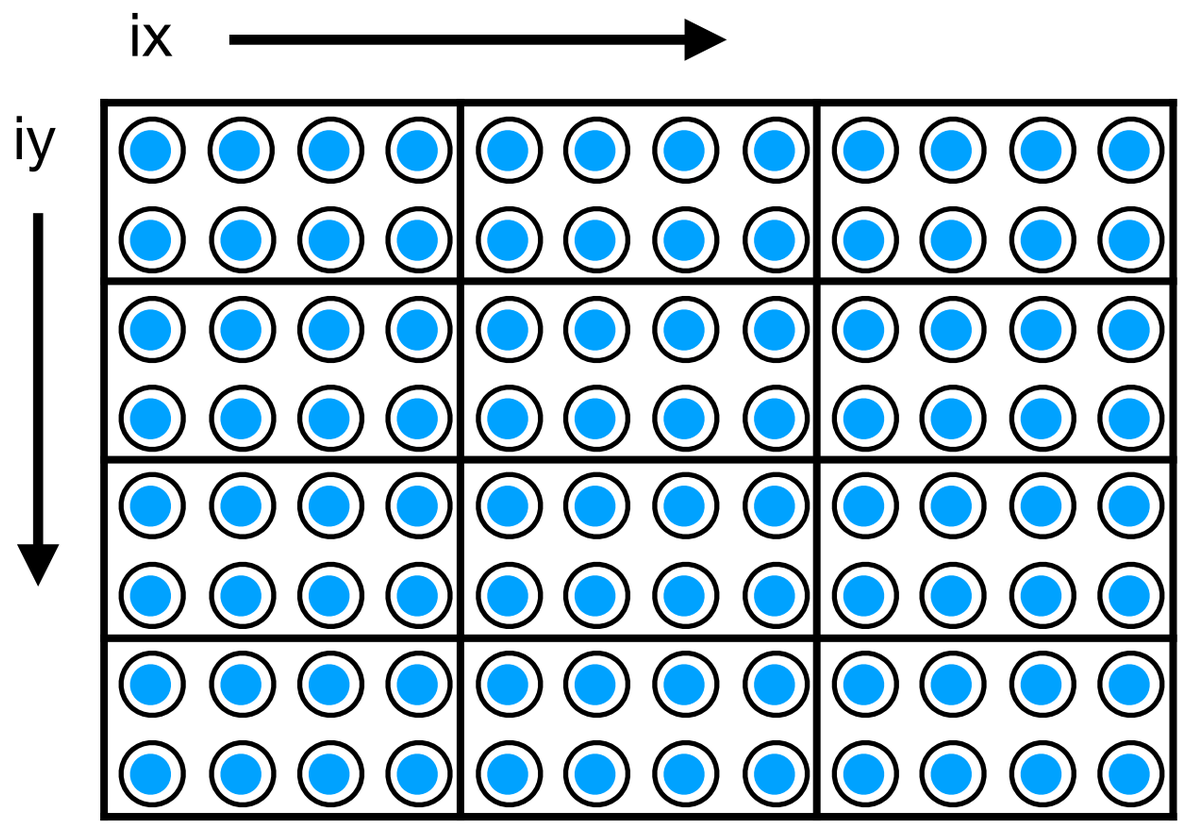
8x12 = 96 threads launched

```
....
dim3 block(4,2);  /* 4x2 block of threads */
dim3 grid(3,4);   /* 3x4 grid of blocks   */

/* Kernel call */
multby2<<<grid,block>>>(m,n,A_dev);
....
```

@Donna Calhoun, BSU 2018-2020

Kernel : multby2



blockDim = (4,2)

gridDim = (3,4)

```
__global__ void multby2(int m, int n, int* A)
{
    int ix = blockDim.x*blockIdx.x + threadIdx.x;
    int iy = blockDim.y*blockIdx.y + threadIdx.y;
    int idx = iy*m + ix;   /* linear address */
    A[idx] = 2*A[idx];
}
```

Only one data element processed

General block/grid sizes

To get exactly **one thread per data element** for an m x n matrix

1. Choose block dimensions (b_x, b_y)
2. From block dimensions and m and n, determine grid dimensions (g_x, g_y)

$$g_x = \left\lceil \frac{m}{b_x} \right\rceil, \qquad g_y = \left\lceil \frac{n}{b_y} \right\rceil$$

```
dim3 block(bx, by);
dim3 grid((m+block.x-1)/block.x, (n+block.y-1)/block.y);
```

@Donna Calhoun, BSU 2018-2020

Available on Redhawk

GPUs	Redhawk Nodes	Number of GPUs	Compute Capability	Series	Year
GeForce GTX Titan	5	1	3.5	Kepler	Mar. 2012
Tesla K20c	6	1	3.5	Kepler	Nov. 2012
Tesla K40c	6	1	3.5	Kepler	Oct. 2013
GeForce GTX Titan X	1-4	2	5.2	Maxwell	Mar. 2015

Grid, block, thread hardware limits

	Compute capability 3.5, and 5.2
Maximum threads per block	1024
Maximum thread dimensions (x,y,z)	(1024, 1024, 64)
Maximum grid dimension (x,y,z)	(2147483647, 65535, 65535)
Global memory	6GB, 11.2GB, 12GB

Execution configurations

So far, we have looked at 2d grids and 2d blocks, and have generally launched one thread per data element.

Different configurations lead to different execution results

Test different execution configurations for $m = 2^{14},\ n = 2^{14}$

```
$ mapping_demo --dimx 32 --dimy 32
```

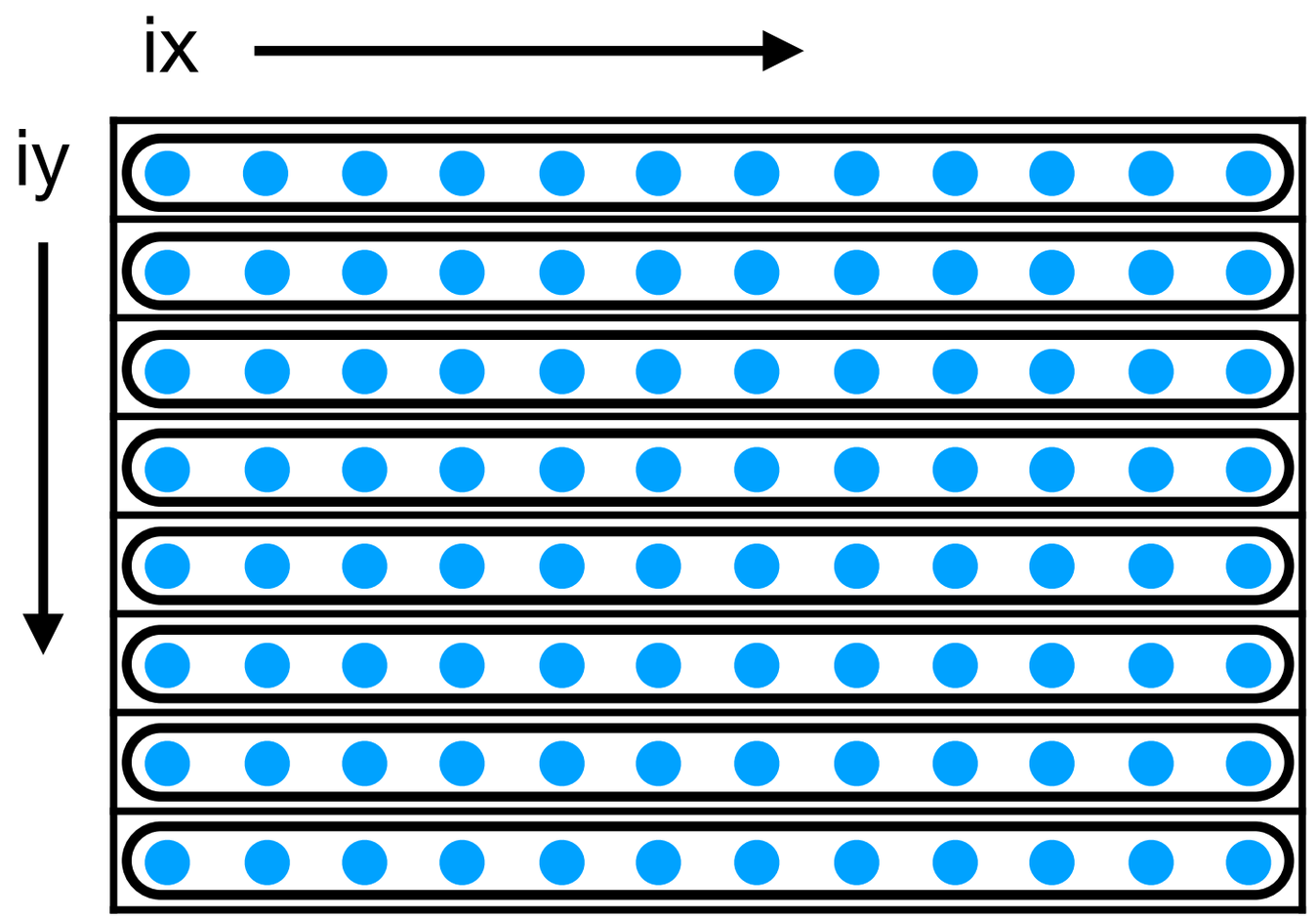
or

```
$ nvprof --metrics achieved_occupancy mapping_demo --dimx 32 --dimy 32
```

Execution Configuration

GPU	grid.x	block.x	grid.y	block.y	Time (ms)	Occupancy
Titan	512	32	512	32	12.29	76%

Multiple elements per thread

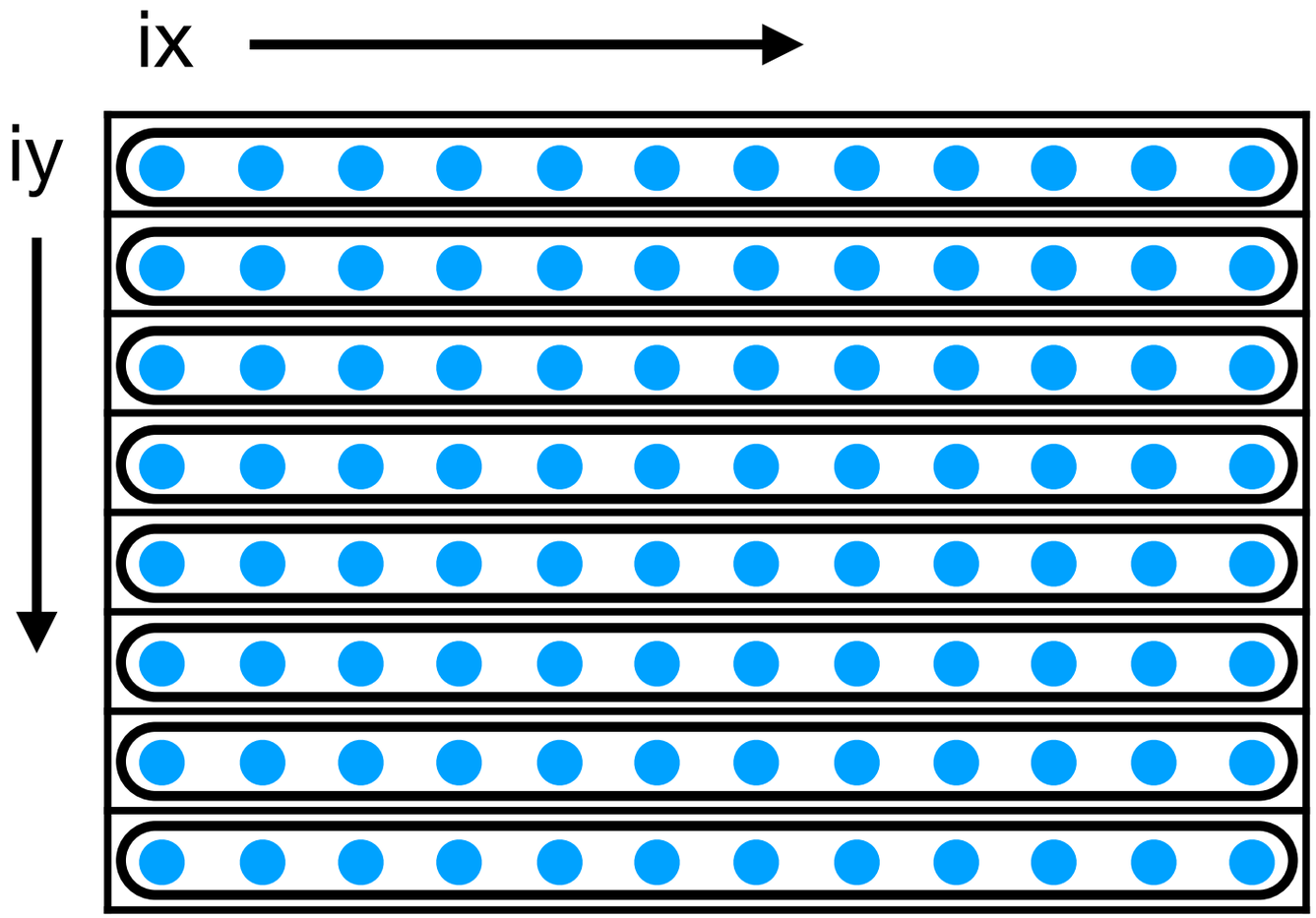


one data element

one thread

```
dim3 block(1,1);   /* 1 thread per block */
dim3 grid(1,(n + block.y - 1)/block.y);
multby2<<<grid,block>>>(m,n,dev_A);
```

Kernel : multby2

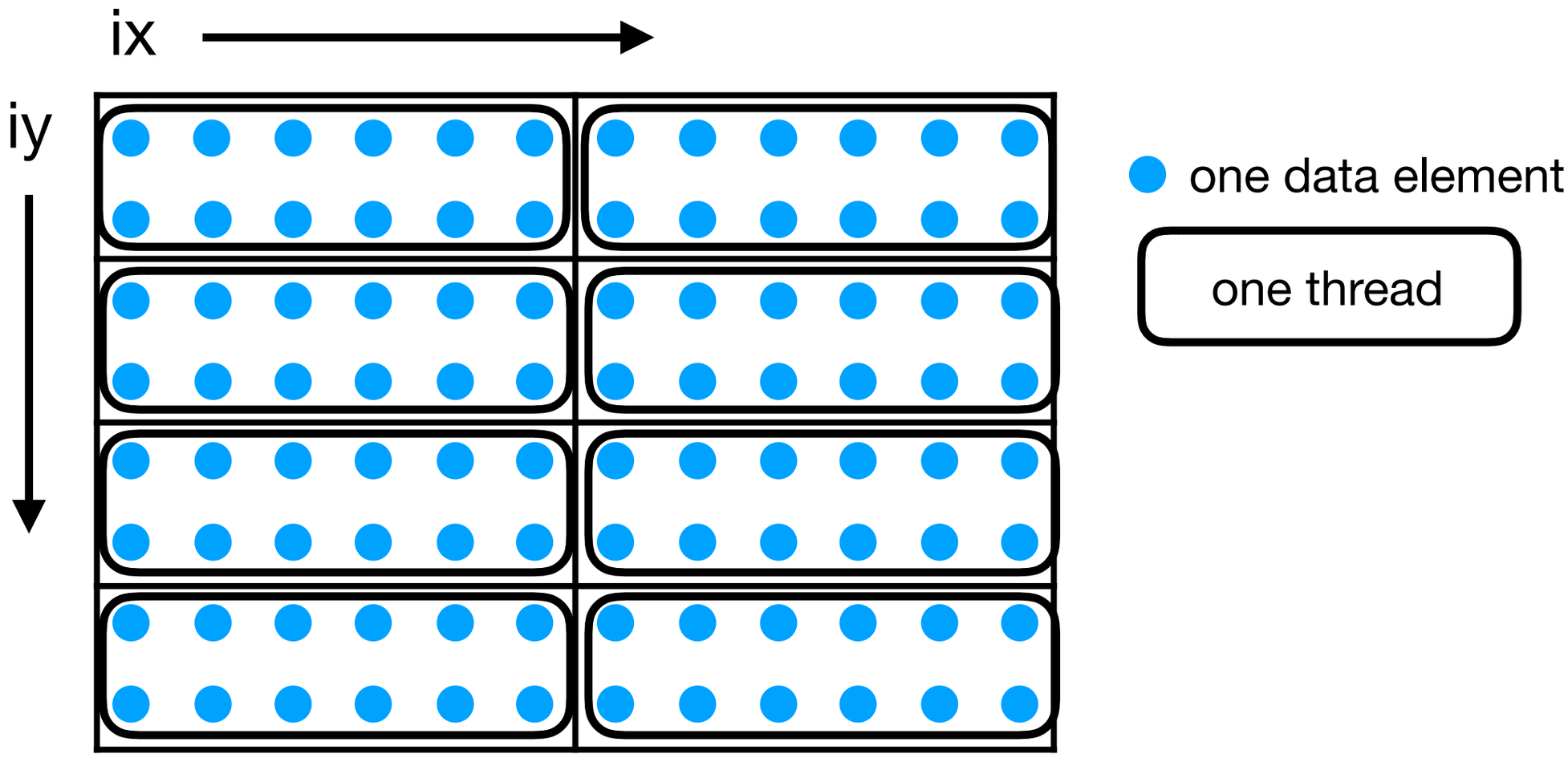



```
blockDim = (1,1)
gridDim = (1,8)
```

```
__global__ void multby2(int m, int n, int* A) {
    int iy = blockIdx.y;
    for(int ix = 0; ix < m; ix++) {
        int idx = iy*m + ix;
        A[idx] = 2*A[idx];
    }
}
```

@Donna Calhoun, BSU 2018-2020

Mapping Data to Blocks/Threads



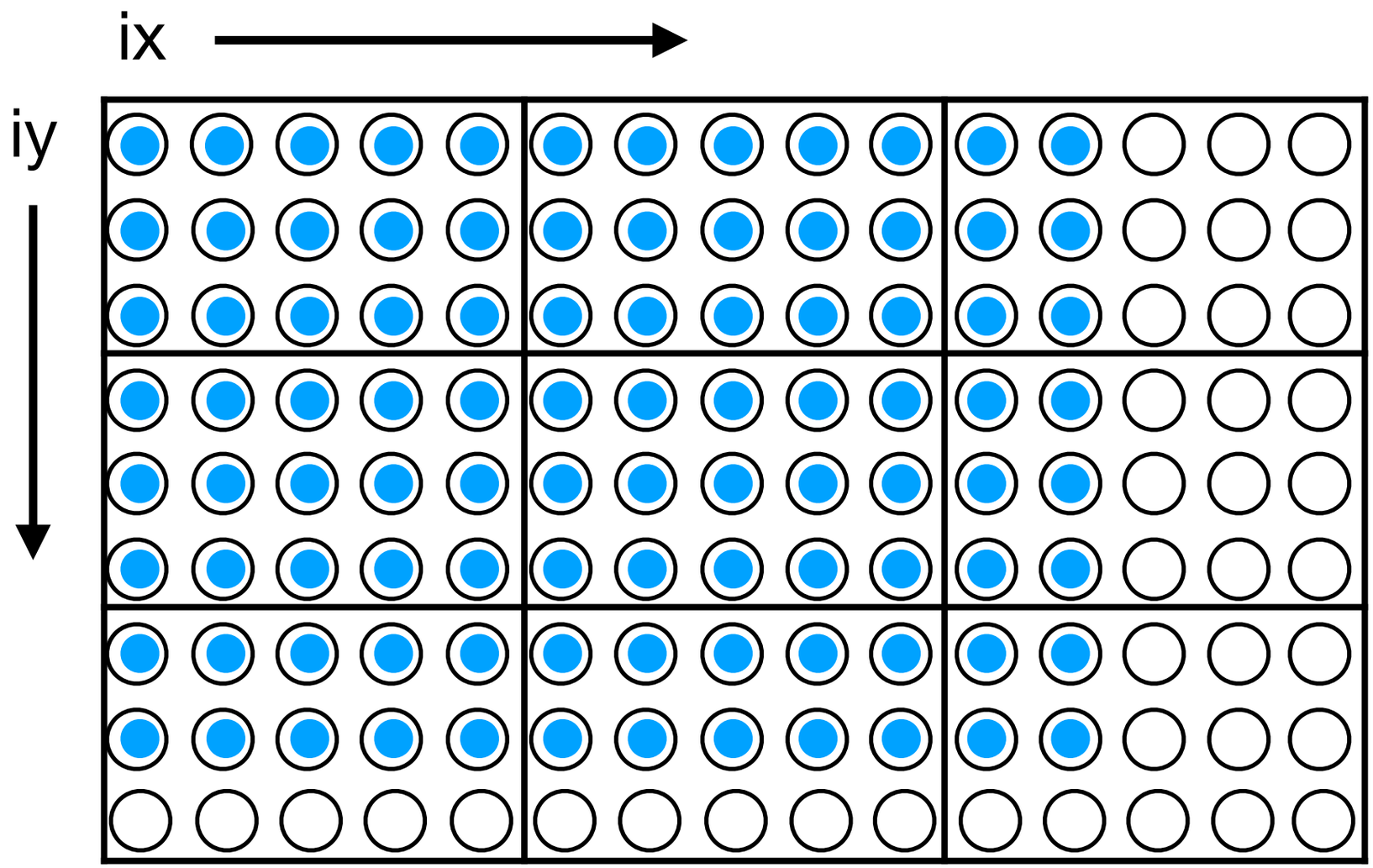

```
dim3 block(1,1);     /* 1 thread per block */
dim3 grid(2,4);
multby2<<<grid,block>>>(m,n,dev_A);
```

Kernel : multby2

```
__global__ void multby2(int m, int n, int* A)
{
    int ix,iy,idx, i, j;
    int msub = m/gridDim.x;
    int nsub = n/gridDim.y;
    for(i = 0; i < msub; i++)
    {
        for(j = 0; j < nsub; j++)
        {
            ix = blockIdx.x*msub + i;
            iy = blockIdx.y*nsub + j;
            idx = iy*m + ix;
           A[idx] = 2*A[idx];
        }
    }
}
```

@Donna Calhoun, BSU 2018-2020

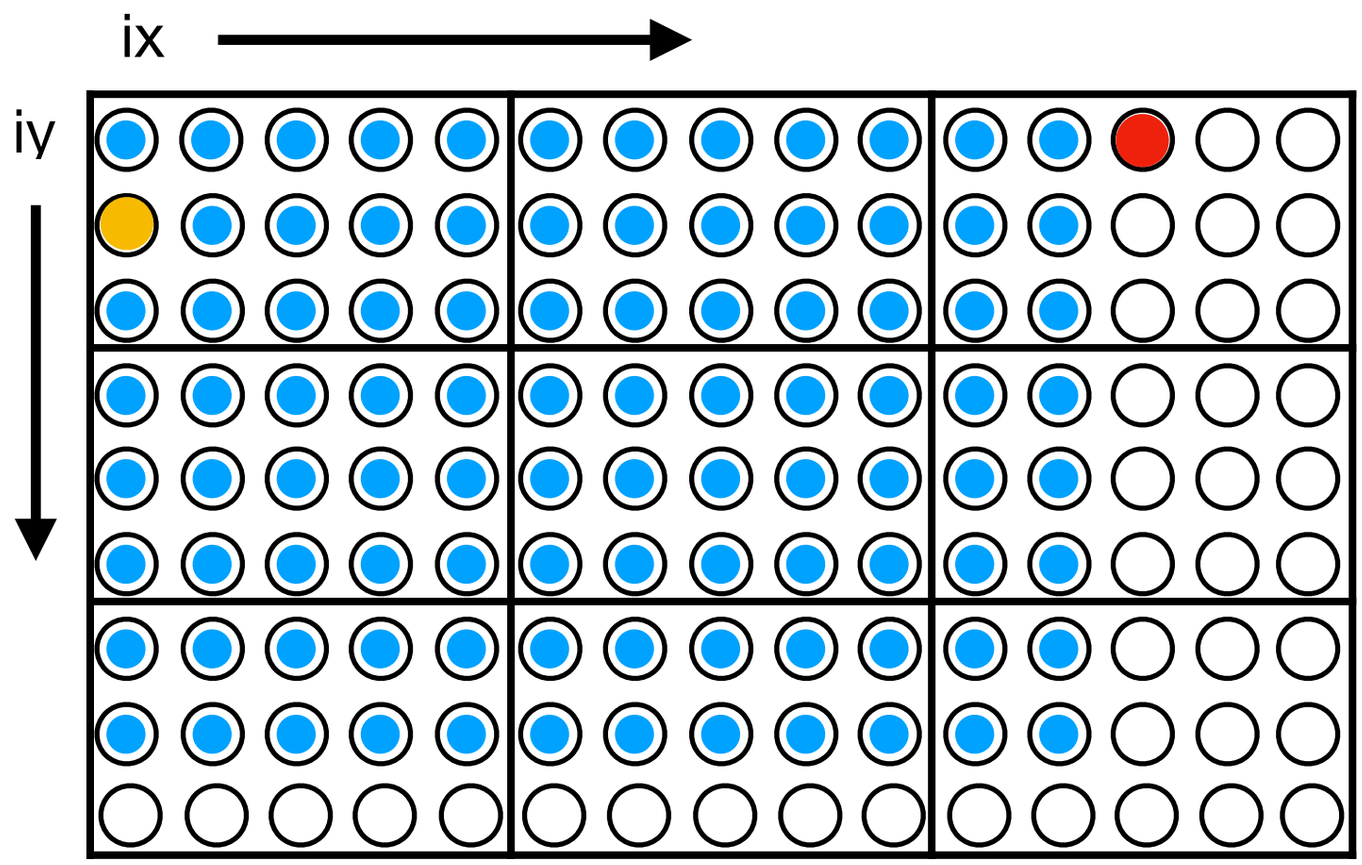
More threads than data elements?




```
dim3 block(5,3);
dim3 grid(3,3);
multby2<<<grid,block>>>(12,8,dev_A);
```

195 threads are launched. The kernel is called for each thread

More threads than data elements?



```
ix = 5*2 + 2 = 12
iy = 3*0 + 0 = 0
idx = 0*12 + 12 = 12
```

```
ix = 5*0 + 0 = 0
iy = 3*0 + 1 = 1
idx = 1*12 + 0 = 12
```

Which thread should process A[12] ??

```
__global__ void multby2(int m, int n, int* A) {
    int ix = blockDim.x*blockIdx.x + threadIdx.x;
    int iy = blockDim.y*blockIdx.y + threadIdx.y;
    int idx = iy*m + ix;
    if (ix < m && iy < n)          /* Make sure only one thread */
        A[idx] = 2*A[idx];         /* operates on A[12]         */
}
```

@Donna Calhoun, BSU 2018-2020