That was Blocks in Parallel. What about Threads in Parallel

C Host

GPU Kernel

Function Call Change

Cuda

```
// add<<<1, 1>>>(da, db, dc); // single thread GPU
// add<<<N, 1>>>(da, db, dc); // N blocks on GPU
   add<<<1, N>>>(da, db, dc); // N threads on GPU
```

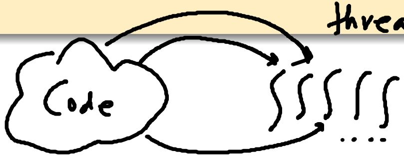
Changes in Kernel Code

```
__global__ void add(int *a, int *b, int *c)
{
    c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];
}
```

Single instruction multiple threads

- Rest of Host code would be the same

~~for~~ body id scheme.

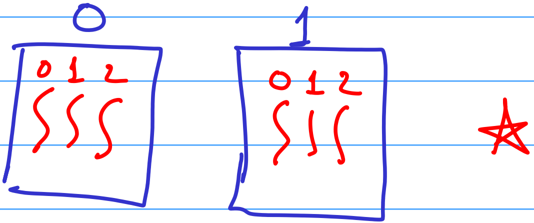


3x2 = 6 threads.

```
fnName <<<2, 3>>>( ——— );
```

blocks

threads.



1	1	1	1	1	1	1	1

A[8]

+

1	1	1	1	1	1	1	1

B[8]

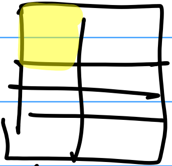
=

2	2	2	2	2	2	2	2

c[8]

GPU RAM

8 cores



$\frac{1}{8} = 12.5\%$

add<<< 8 , 1 >>>(A, B, C);

0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7

blockId

threadId

c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x]

1	1	1	1	1	1	1	1

A[8]

+

1	1	1	1	1	1	1	1

B[8]

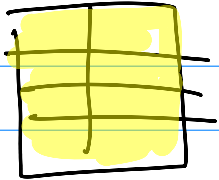
=

2	2	2	2	2	2	2	2

c[8]

GPU RAM

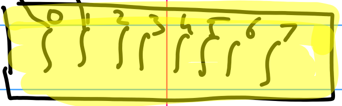
8 cores



$8/8 = 100\%$

add<<< 1 , 8 >>>(A, B, C);

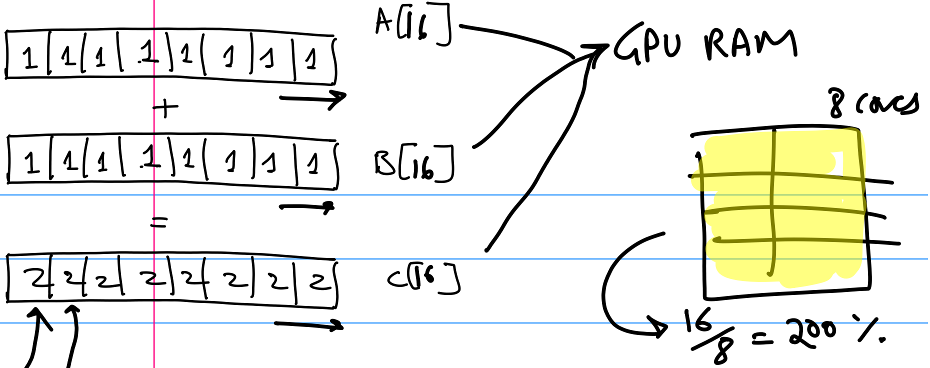
0



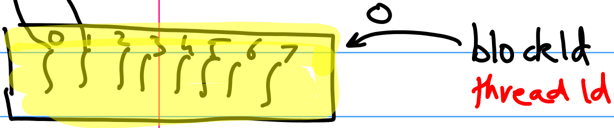
blockId

threadId

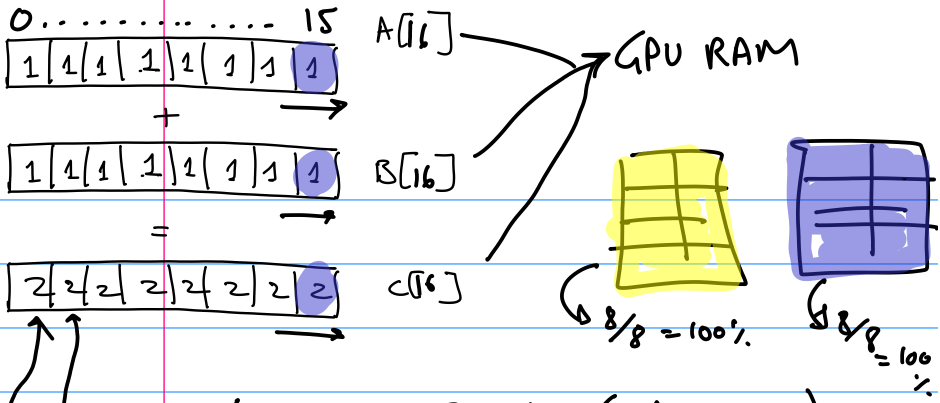
c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x]



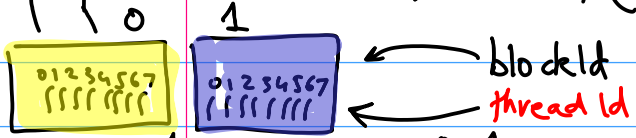
add<<< 1 , 16 >>>(A, B, C);



c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x]

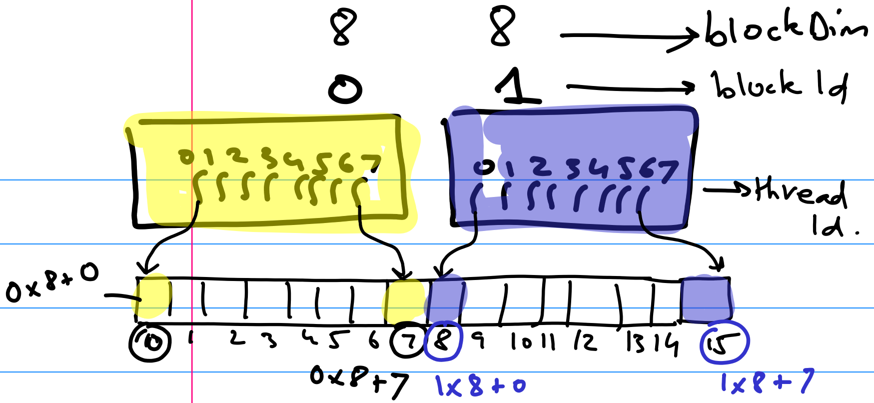


add<<< 2 , 8 >>>(A, B, C);



c[index] = a[index] + b[index]

0...1 / 0...7 (at each index)



blockId
threadId $\Longrightarrow$ Inde

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

blockDim	4	4	4	4
blockIdx.x	0	1	2	3

$\boxed{\text{index} = \text{blockIdx.x} * \text{blockDim.x} + \text{threadIdx.x}}$

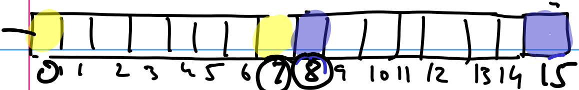
~~<<<2, 8>>>~~

$\boxed{\lll 4, 4 \ggg}$

X, Y

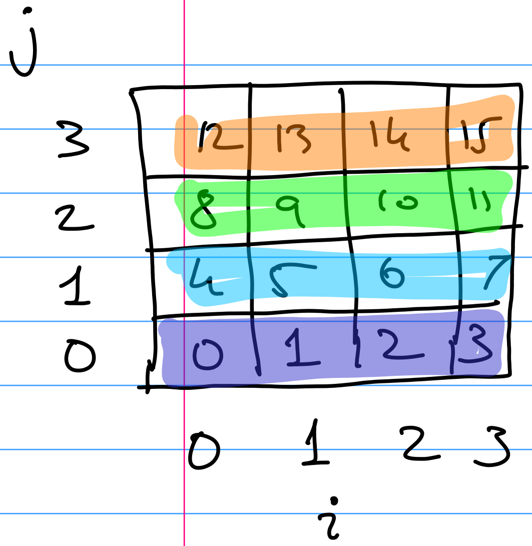
0x4+0

~~0x8+0~~



~~0x8+7~~ ~~1x8+0~~ ~~1x8+7~~

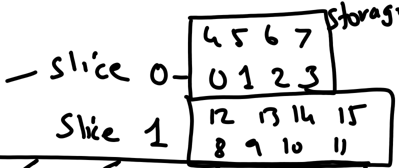
1x4+3 2x4+0 3x4+3




```
for(j=0; j<4; j++)
    for(i=0; i<4; i++)
```

index = $\underbrace{j}_{\text{block}} * 4 + \underbrace{i}_{\text{thread}}$

$k \times 4 \times 2$
$+ j \times 4$
$+ i$



storage 1D 0-16

threads 1D

16x1

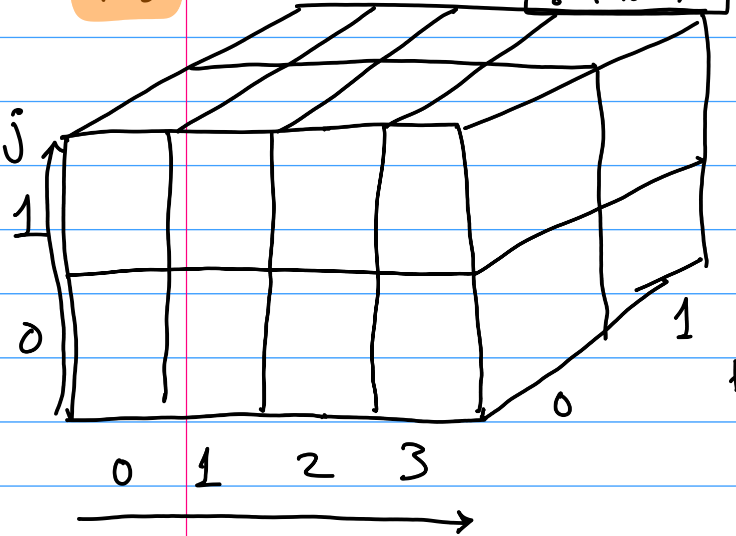
1x16

threads 2D

4x4

threads 3D

2x4x2



$k \times 4 \times 2$
$+ j \times 4$
$+ i$ → Cuda

$\boxed{\text{Index}}$

blockIdx.z * blockDim.x * blockDim.y

+ blockIdx.y * blockDim.x

+ threadIdx.x

3D

2D

1D

Data — Thread

1D → 1D

1D → 2D

1D → 3D

What if

2D

array of size 8

8x1 = 12%

1x8 = 50%

4x2 = 100%

2x2x2 = 200%

Thread Dimensions

load balance

Blocks or Threads. Whatever. Code is running in Parallel anyways

- Let's look at sample specs for NVIDIA Kepler K40
  - Cores = 2,880
  - Multiprocessors = 15
  - Cores / multi-processor = 192
- If we make parallel blocks, 1 block will be assigned to 1 multi-processor. This means all multi-processors will be busy, but within the multi-processor, 1 core has work to do, the remaining others are free.
- If we make parallel threads and 1 block, only 1 multi-processor will be busy, the remaining will be free.
- In either case, the GPU is **under-utilized**. For maximum utilization, use both (blocks + threads)
- **Note: Above is programmer's interpretation. The actual execution model loosely follows this interpretation but has other restrictions also.**

Case: Higher Dimensions

1D = <<< N, 1 >>>

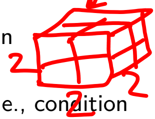
<<< 1, N >>>

3D

- So far, we have passed single values to kernel function call.
- For higher dimensions, use:
  - dimGrid(g_x, g_y, g_z) for dimensions of the grid
  - dimBlock(t_x, t_y, t_z) for dimensions of the thread block

dimGrid(1,1,1)
dimBlock(2,2,2)

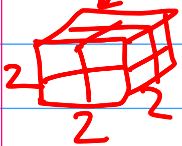
- Total threads in thread-block cannot exceed 512 (or 1024 for some GPU's), i.e. condition $t_x \times t_y \times t_z \leq 512$ must be satisfied.
- Grid dimensions cannot exceed 32,768 in either dimension, i.e., condition $\max(g_x, g_y) \leq 32,768$
- dimGrid component must be evenly divisible by dimBlock component, i.e., condition $mod(g_n, t_n) = 0$ must be satisfied.



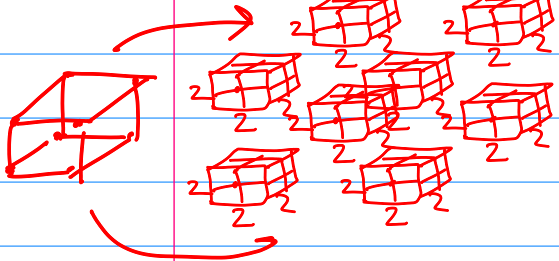

```
CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS:3
CL_DEVICE_MAX_WORK_ITEM_SIZES:       1024 / 1024 / 64 (512/512/32)
CL_DEVICE_MAX_WORK_GROUP_SIZE:       1024 (512)
```

2D = <<< M, N >>>

dimGrid(2,2,2)
dimBlock(2,2,2)



3D threads



6D

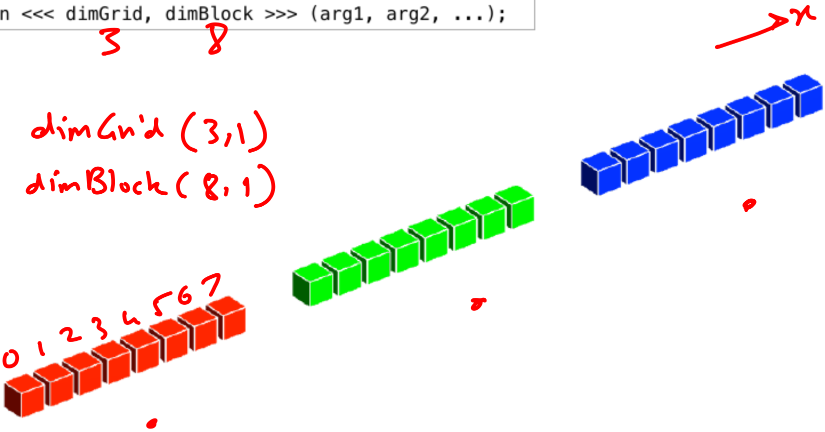
Case: Higher Dimensions (cont.)

```
dim3 dimGrid(3,1,1);
dim3 dimBlock(8,1,1);

kernelfn <<< dimGrid, dimBlock >>> (arg1, arg2, ...);
```

2D

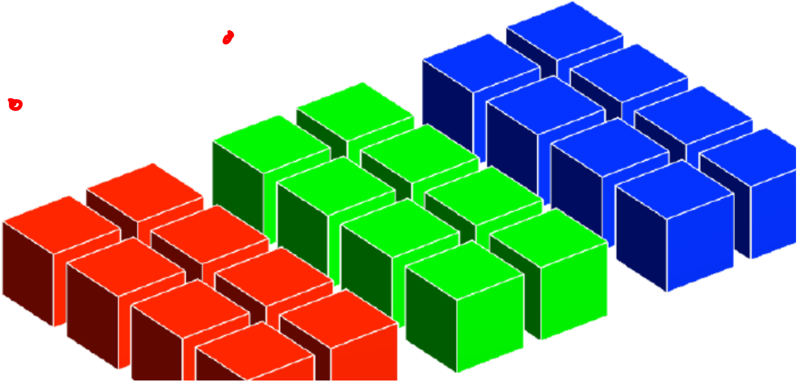
3 8

dim2 dimGrid (3,1)
dim2 dimBlock(8,1)



Case: Higher Dimensions (cont.)

```
dim3 dimGrid(3,1,1);
dim3 dimBlock(2,4,1);

kernelfn <<< dimGrid, dimBlock >>> (arg1, arg2, ...);
```



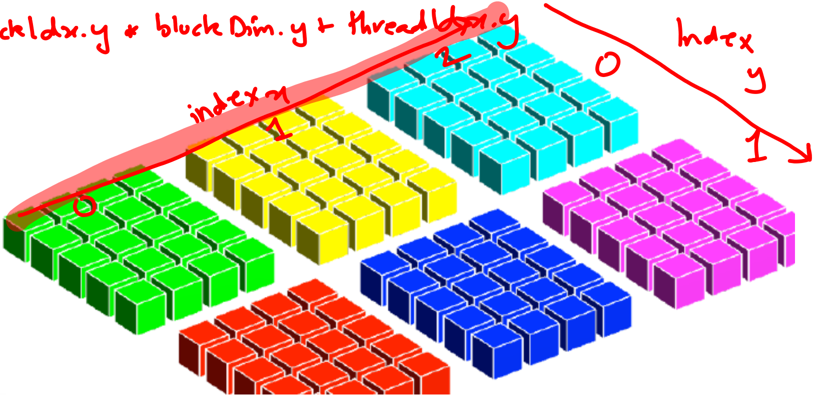
Case: Higher Dimensions (cont.)

$I = I_x * gridDim.x * blockDim.x + I_y$

```
dim3 dimGrid(3,2,1);
dim3 dimBlock(4,5,1);

kernelfn <<< dimGrid, dimBlock >>> (arg1, arg2, ...);
```

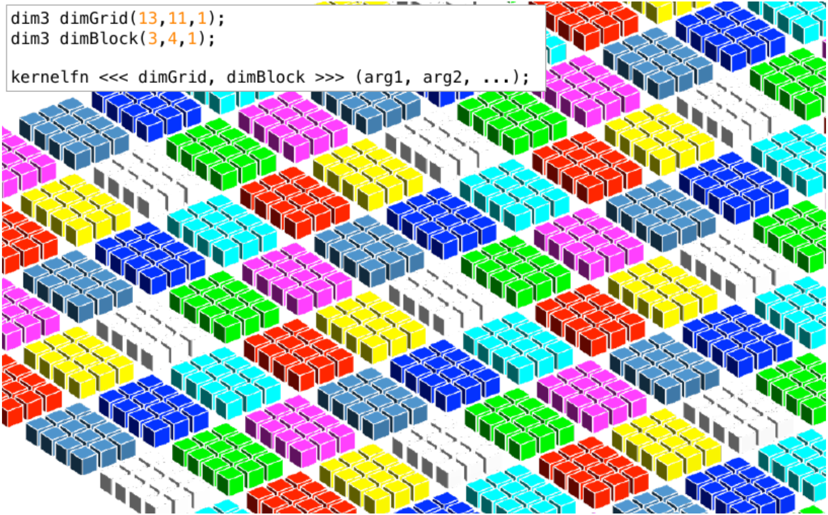
$I_x = blockIdx.x * blockDim.x + threadIdx.x$

$I_y = blockIdx.y * blockDim.y + threadIdx.y$



Case: Higher Dimensions (cont.)

```
dim3 dimGrid(13,11,1);
dim3 dimBlock(3,4,1);

kernelfn <<< dimGrid, dimBlock >>> (arg1, arg2, ...);
```

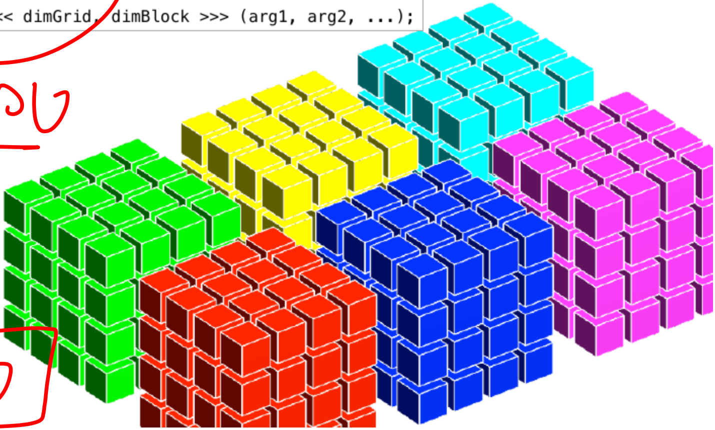


Case: Higher Dimensions (cont.)

```
dim3 dimGrid(3,2,1);
dim3 dimBlock(4,4,4);

kernelfn <<< dimGrid, dimBlock >>> (arg1, arg2, ...);
```

threads

1 GPU



M GPU

2 Host Code → Kernel

1 Cuda Drivers.

↳ NVIDIA

↳ Proprietary.

Open Source

Black list. ~~Nouveau~~

White list

NVIDIA

Combining both Concepts of Blocks and Threads

Formula to identify a point

```
int index = blockIdx.x * blockDim.x + threadIdx.x;
```

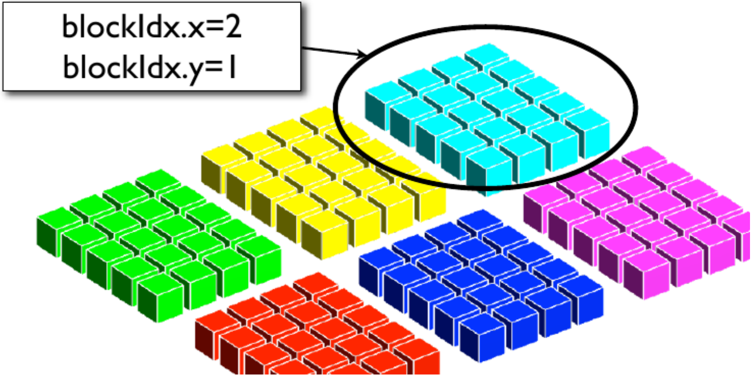
- blockIdx.x = Index of thread block in grid
- blockDim.x = Number of threads in 1D thread-block
- threadIdx.x = Index of thread in 1D thread block

- Each thread execute kernel code is automatically provided the following variables
  - threadIdx.x, threadIdx.y, threadIdx.z
  - blockDim.x, blockDim.y, blockDim.z
  - blockIdx.x, blockIdx.y

* gridDim.x , gridDim.y , gridDim.z

Combining both Concepts of Blocks and Threads (cont.)

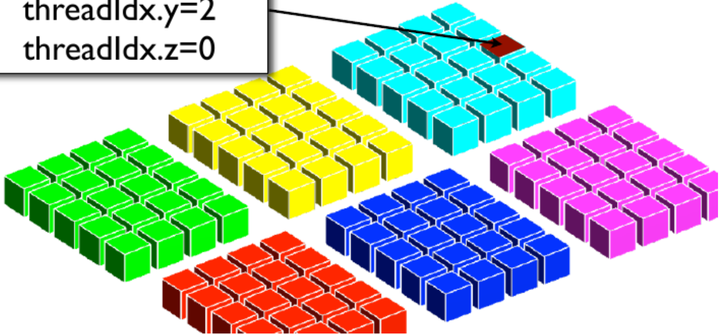
```
dim3 dimGrid(3,2,1);
dim3 dimBlock(4,5,1);

kernelfn <<< dimGrid, dimBlock >>> (arg1, arg2, ...);
```



Combining both Concepts of Blocks and Threads (cont.)

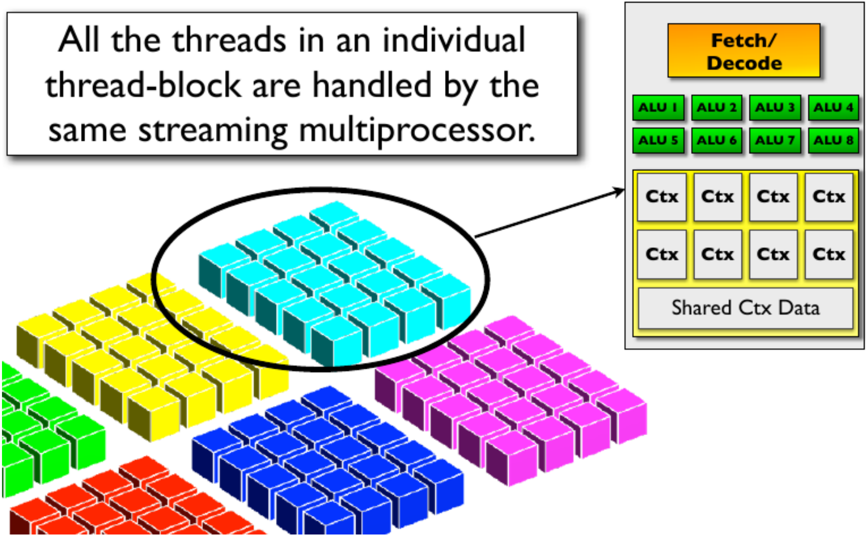
```
dim3 dimGrid(3,2,1);
dim3 dimBlock(4,5,1);

kernelfn <<< dimGrid, dimBlock >>> (arg1, arg2, ...);
```

threadIdx.x=3
threadIdx.y=2
threadIdx.z=0



Higher Dimensions: Execution Model Revisited

All the threads in an individual thread-block are handled by the same streaming multiprocessor.



Higher Dimensions: Execution Model Revisited (cont.)