CUDA Thread assignment

It’s all about data decomposition

CUDA and GPGPU in general is best for large amounts of data where many threads can execute and compute quite a bit of calculations in parallel

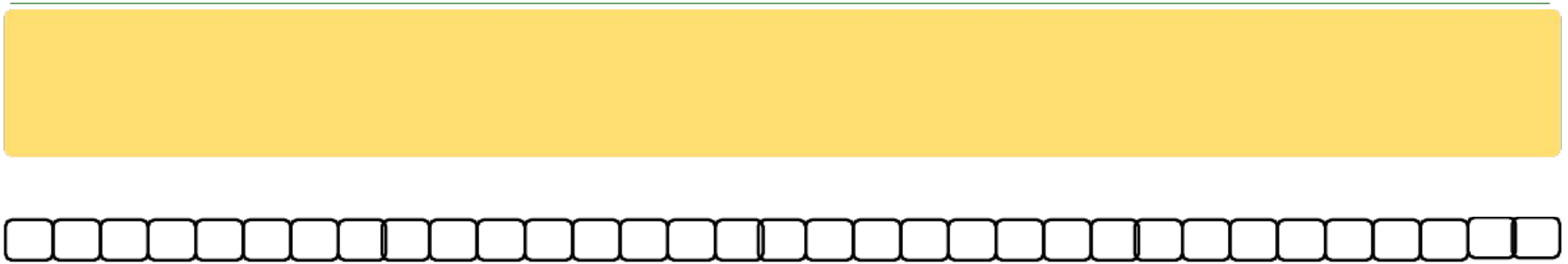
We arrange our data in 1, 2, or 3 dimensions, depending on the problem

So CUDA designers enabled a complicated, yet natural way to map the decomposition of the threads to the data

1-D Data

1-D array of data

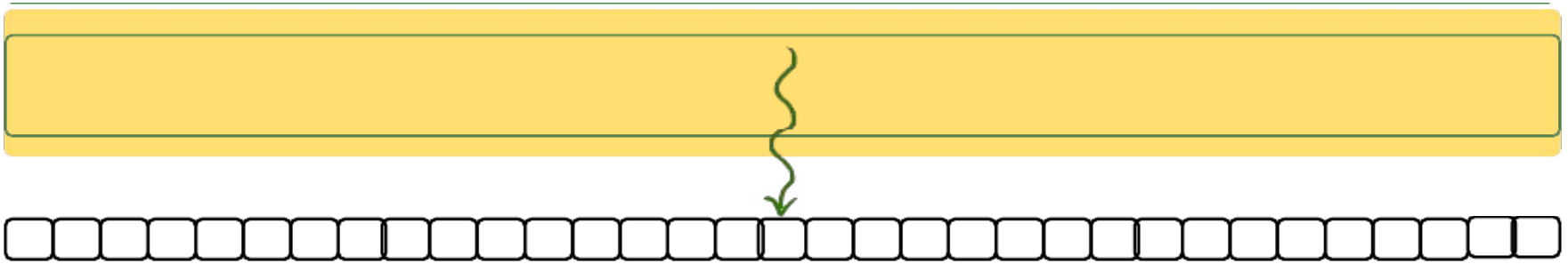
1-D Data, 1-D Grid with 1-D blocks



1-D array of data

1 x 1 Grid

1-D Data, 1-D Grid with 1-D blocks

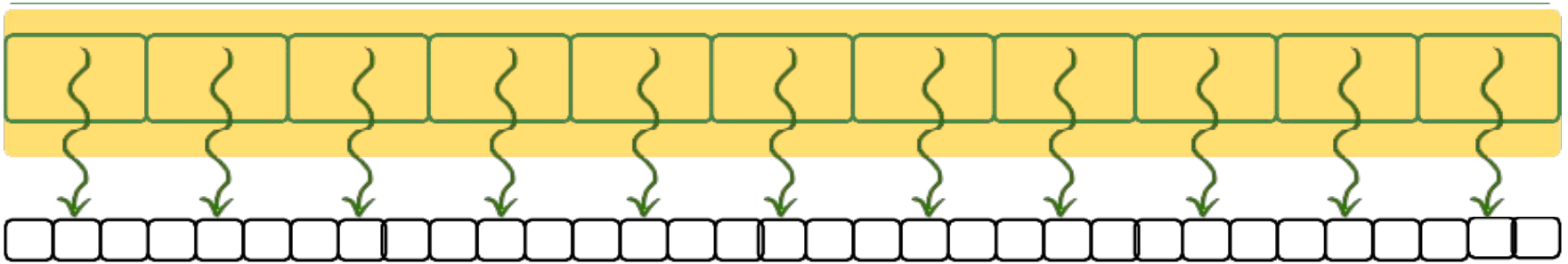


1-D array of data

1 x 1 Grid

1 x 1 Block

1-D Data, 1-D Grid with 1-D blocks

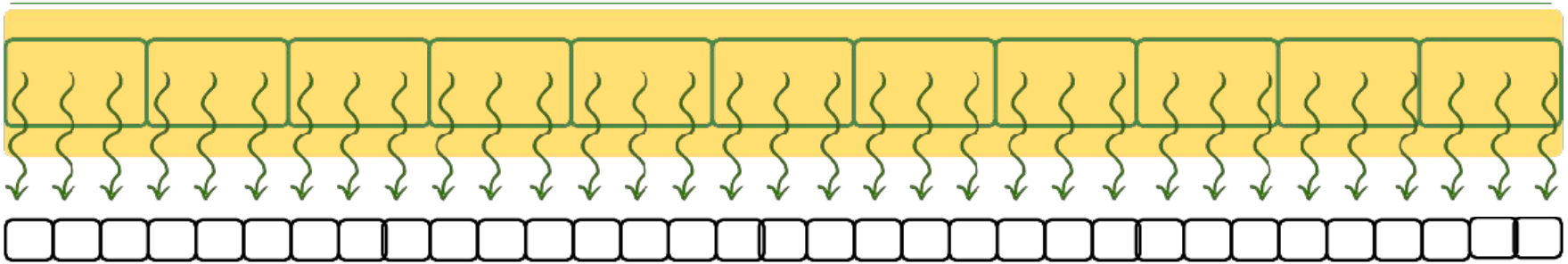


1-D array of data

1 x N Grid

1 x 1 Block

1-D Data, 1-D Grid with 1-D blocks

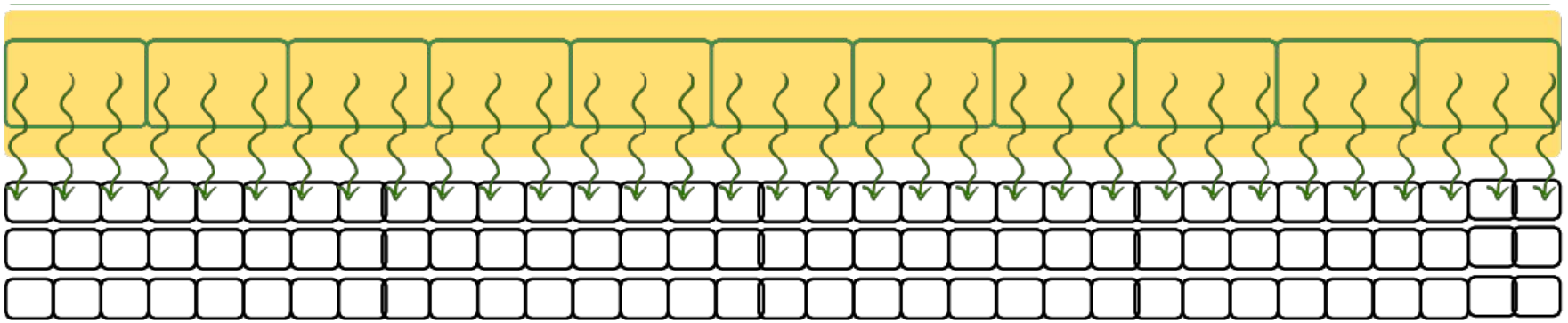


1-D array of data

1 x N Grid of Blocks

1 x T Blocks of Threads

2-D Data, 1-D Grid, 1-D Blocks

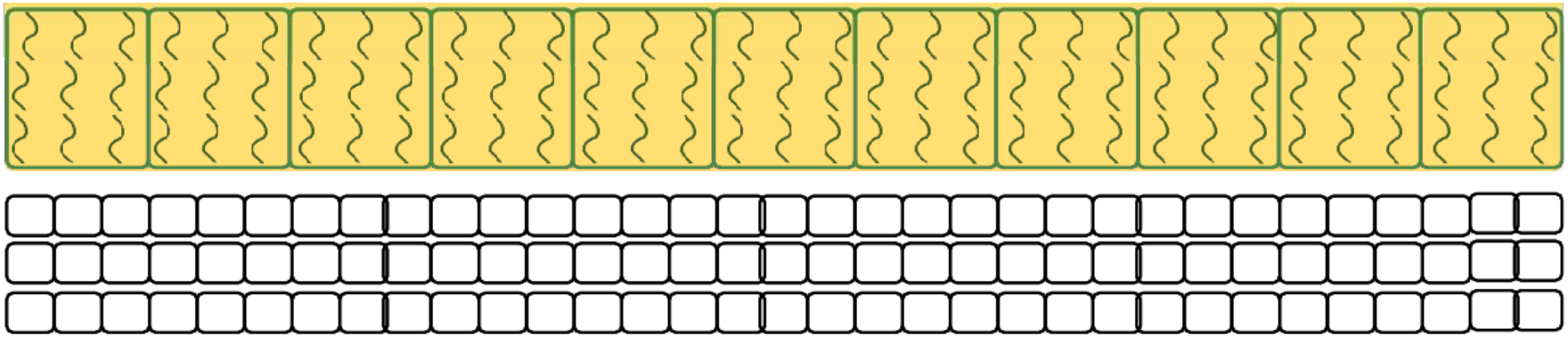


2-D array of data

1 x N Grid of Blocks

1 x T Blocks of Threads

2-D Data, 1-D Grid, 2-D Blocks

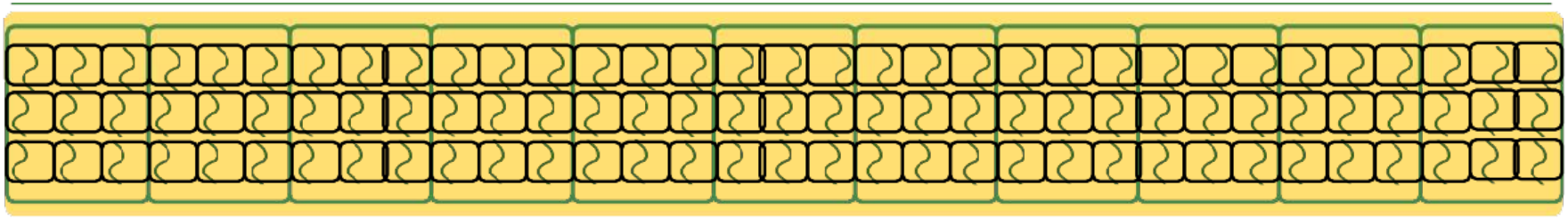


2-D array of data

1 x N Grid of Blocks

X x Y Blocks of Threads

2-D Data, 1-D Grid, 2-D Blocks

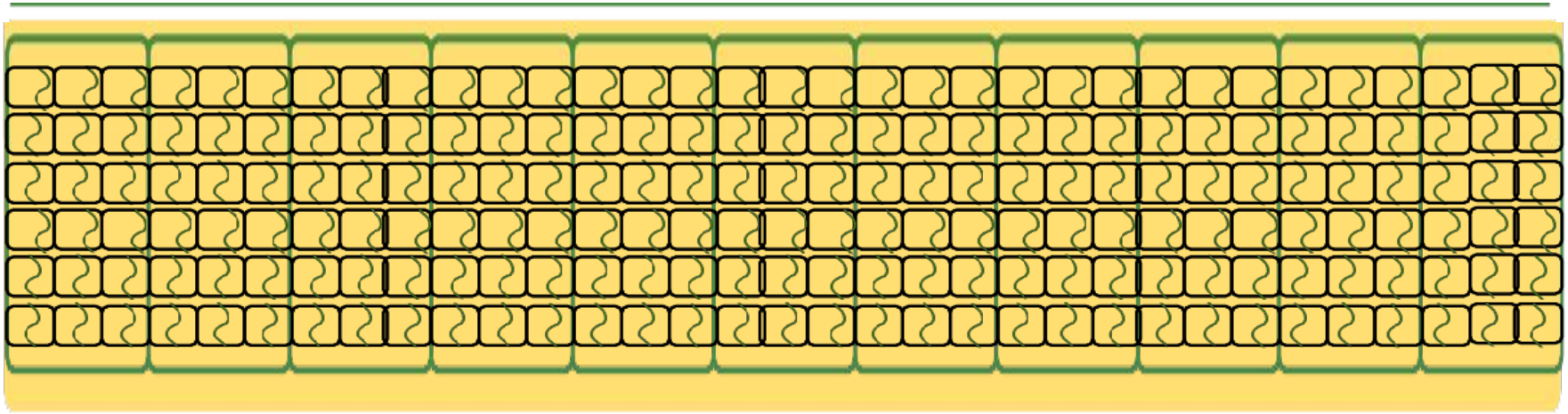


2-D array of data

1 x N Grid of Blocks

X x Y Blocks of Threads

2-D Data, 1-D Grid, 2-D Blocks

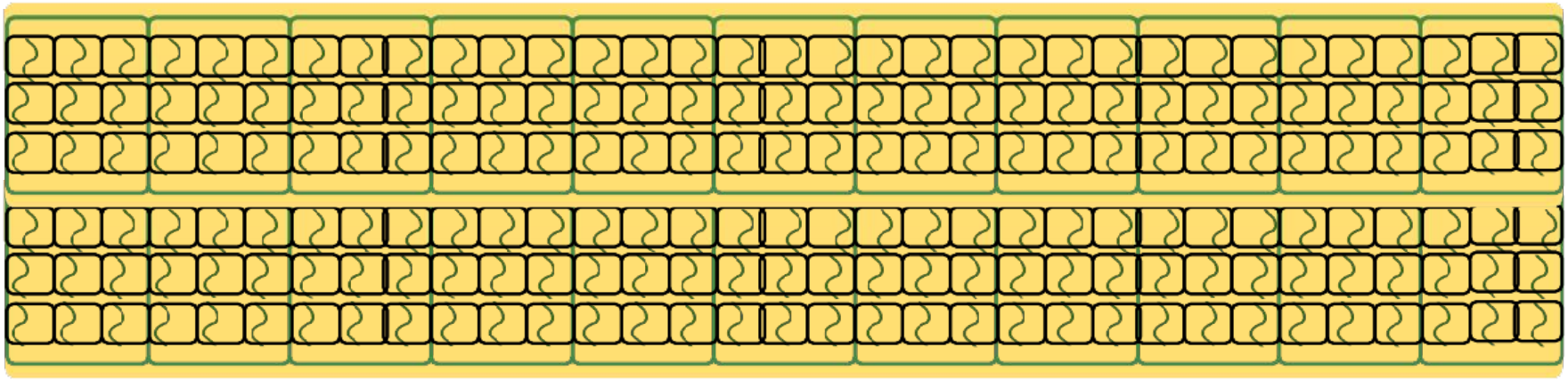


2-D array of data

1 x N Grid of Blocks

X x Y Blocks of Threads

2-D Data, 2-D Grid, 2-D Blocks

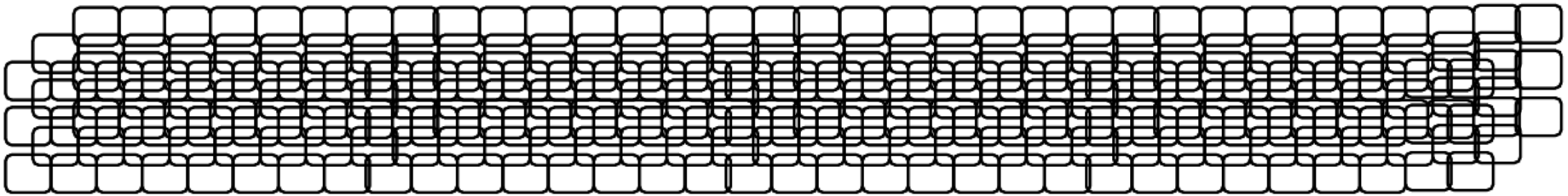


2-D array of data

2 x N Grid of Blocks

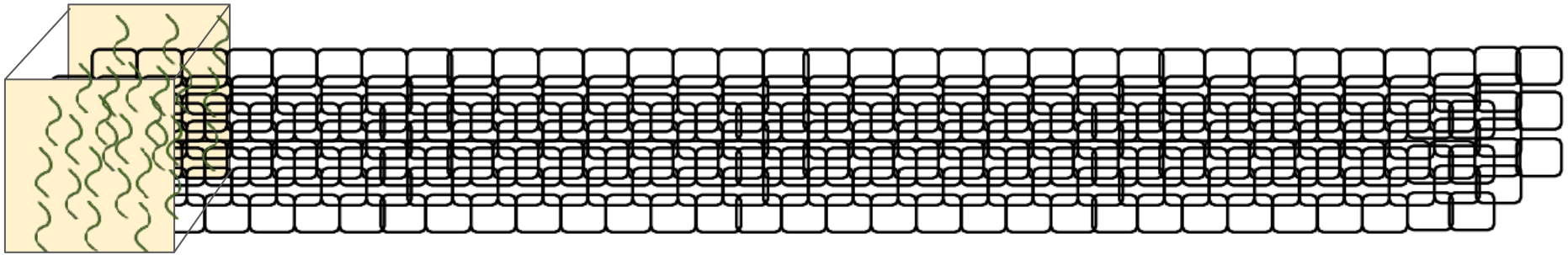
X x Y Blocks of Threads

3-D Data, 1,2,3-D Grid, 1,2,3-D Blocks



3-D array of data

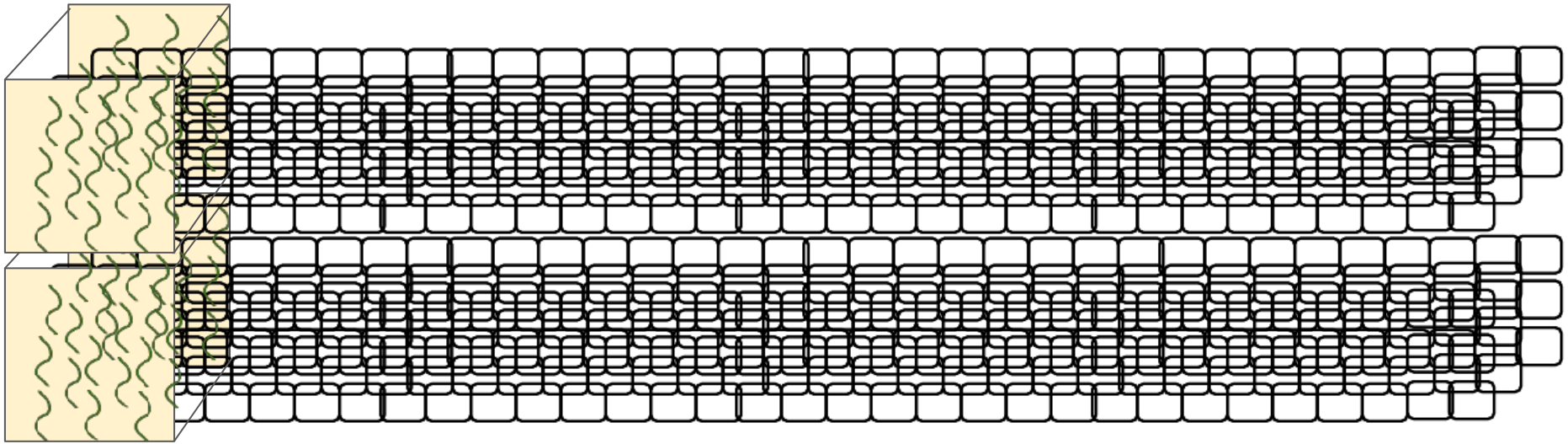
3-D Data, 1,2,3-D Grid, 1,2,3-D Blocks



3-D array of data

1-D Grid of 3-D blocks is one possibility

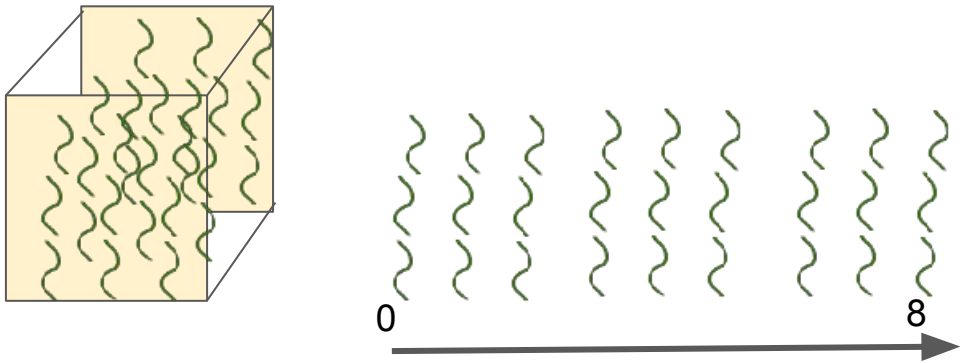
3-D Data, 1,2,3-D Grid, 1,2,3-D Blocks



3-D array of data

2-D Grid of 3-D blocks is one possibility

Thread id calculation

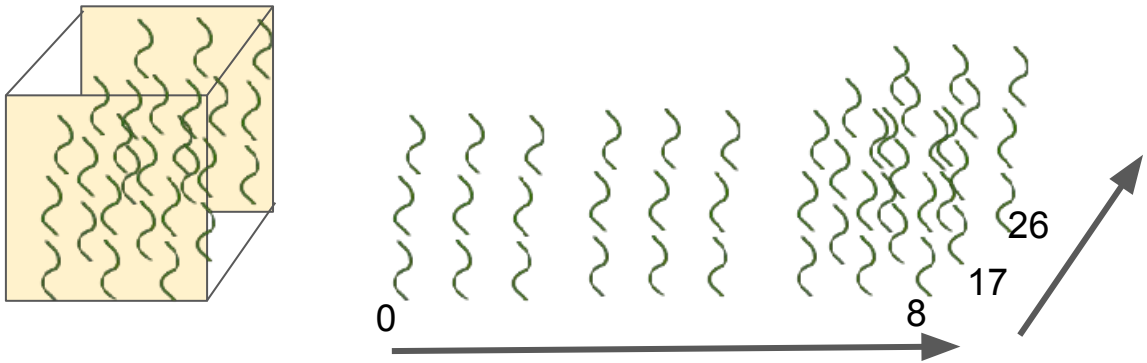


3-D array of data

1-D Grid of 3-D blocks is one possibility

Threads numbered along X direction first

Thread id calculation

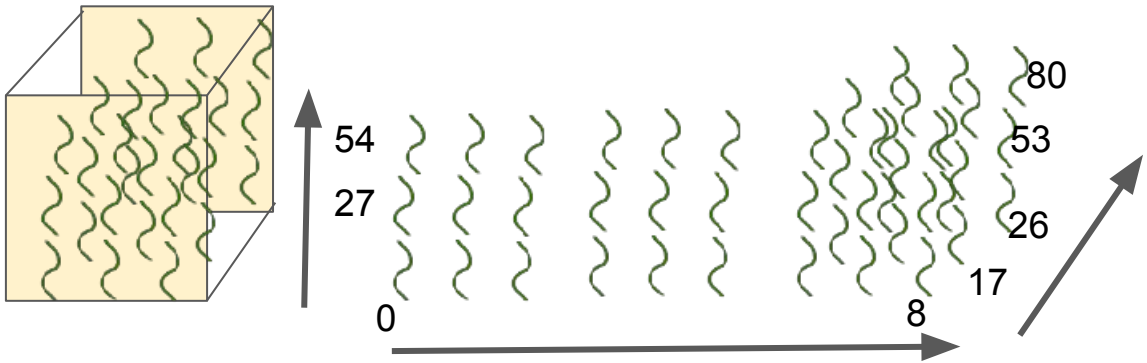


3-D array of data

1-D Grid of 3-D blocks is one possibility

Threads numbered along X direction first, then Y

Thread id calculation



3-D array of data

1-D Grid of 3-D blocks is one possibility

Threads numbered along X direction first, then Y, then Z

Setting the Grid of Blocks

```
__global__ void Func(float* parameter)
```

must be called like this:

```
Func<<< Dg, Db, Ns >>>(parameter)
```

Where *Dg, Db, Ns* are :

Dg is of type *dim3*, dimension and size of the grid [up to 3 dimensions]

Dg.x * *Dg.y* = number of blocks being launched if 2 dimensions

Db is of type dim3, dimension and size of each block

Db.x * *Db.y* * *Db.z* = number of threads per block;

Ns is of type *size_t*, number of bytes in shared memory that is dynamically allocated in addition to the statically allocated memory
Ns is an optional argument which defaults to 0.

use of dim3

For 2-D data, we might decide on something like this:

```
dim3 grid( 512 );          // 512 x 1 x 1

dim3 block( 1024, 1024 ); // 1024 x 1024 x 1

fooKernel<<< grid, block >>>();
```

- Not all the 3 elements need to be provided. Any element not provided during initialization is initialized to 1. Please note that they are initialized to 1, not 0!