A TYPICAL GPU PROGRAM

1. CPU ALLOCATES STORAGE ON GPU — cuda Malloc
2. CPU COPIES INPUT DATA FROM CPU → GPU — cuda Memcpy
3. CPU LAUNCHES KERNEL(S) ON GPU TO PROCESS THE DATA — kernel launch
4. CPU COPIES RESULTS BACK TO CPU FROM GPU — cuda Memcpy

CONFIGURING THE KERNEL LAUNCH

```
square<<<1, 64>>>(d_out, d_in)
```

- `1` → NUMBER OF BLOCKS
- `64` → THREADS PER BLOCK

128 THREADS? `square<<<1, 128>>>( ... )`

Alejar

DEFINING THE GPU COMPUTATION

BIG IDEA

KERNELS LOOK LIKE SERIAL PROGRAMS

WRITE YOUR PROGRAM AS IF IT WILL RUN ON ONE THREAD

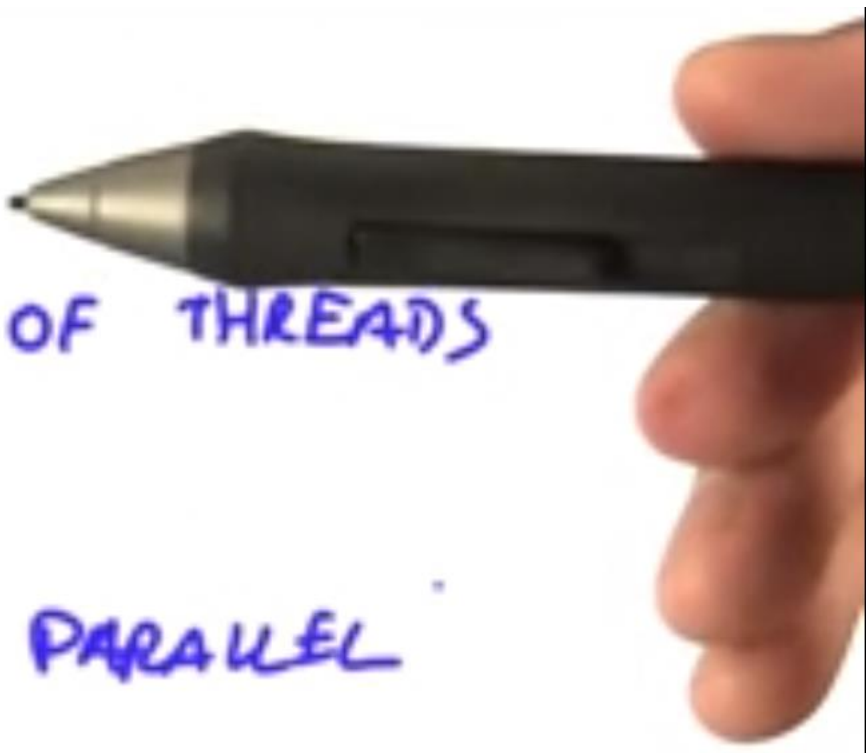
THE GPU WILL RUN THAT PROGRAM ON MANY THREADS

MAKE SURE YOU UNDERSTAND THIS

WHAT IS THE GPU GOOD AT ?

- [ ] LAUNCHING A SMALL NUMBER OF THREADS EFFICIENTLY
- [x] LAUNCHING A LARGE NUMBER OF THREADS EFFICIENTLY
- [ ] RUNNING ONE THREAD VERY QUICKLY
- [ ] RUNNING ONE THREAD THAT DOES LOTS OF WORK IN PARALLEL
- [x] RUNNING A LARGE NUMBER OF THREADS IN PARALLEL

WHAT IS THE GPU GOOD AT?



1. EFFICIENTLY LAUNCHING LOTS OF THREADS
2. RUNNING LOTS OF THREADS IN PARALLEL

You may be used to other programming environments where launching threads is an expensive process.

SIMPLE EXAMPLE:

IN: FLOAT ARRAY [0 1 2 ... 63]

OUT: FLOAT ARRAY [0 1^2 2^2 ... 63^2]

[0 1 4 9 ...]

KERNEL: SQUARE

1. CPU
2. GPU (theory, no code)
3. GPU (code)

We're going to start by looking at how we'd run this code on the CPU.

CPU CODE: SQUARE EACH ELEMENT OF AN ARRAY

```
for (i=0; i< 64; i++) {
    out[i] = in[i] * in[i];
}
```

CPU CODE: SQUARE EACH ELEMENT OF AN ARRAY

```
for (i=0; i<64; i++) {
    out[i] = in[i] * in[i];
}
```

(1) ONLY ONE THREAD OF EXECUTION
("thread" = "one independent path of execution through the code")

(2) NO EXPLICIT PARALLELISM

CPU CODE: SQUARE EACH ELEMENT OF AN ARRAY

```
for (i=0; i<64; i++) {
    out[i] = in[i] * in[i];
}
```

(1) ONLY ONE THREAD OF EXECUTION ("thread" = "one independent path of execution through the code")

(2) NO EXPLICIT PARALLELISM

QUIZ:

HOW MANY MULTIPLICATIONS? 64

1 * TAKES 2 ns.
HOW LONG TO EXECUTE? 128

GPU CODE: A HIGH-LEVEL VIEW

CPU

ALLOCATE MEMORY

COPY DATA TO/FROM GPU

LAUNCH KERNEL

↑

SPECIFIES DEGREE OF PARALLELISM

GPU

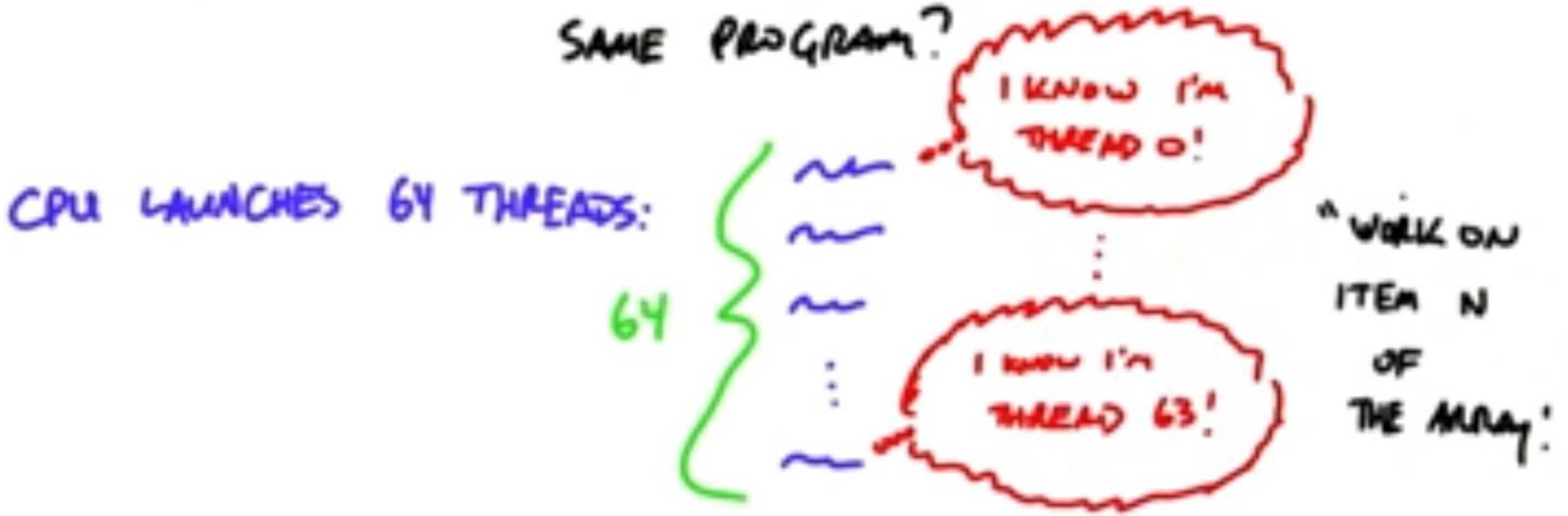
EXPRESS OUT = IN · IN

↑

SAYS NOTHING ABOUT THE DEGREE OF PARALLELISM.

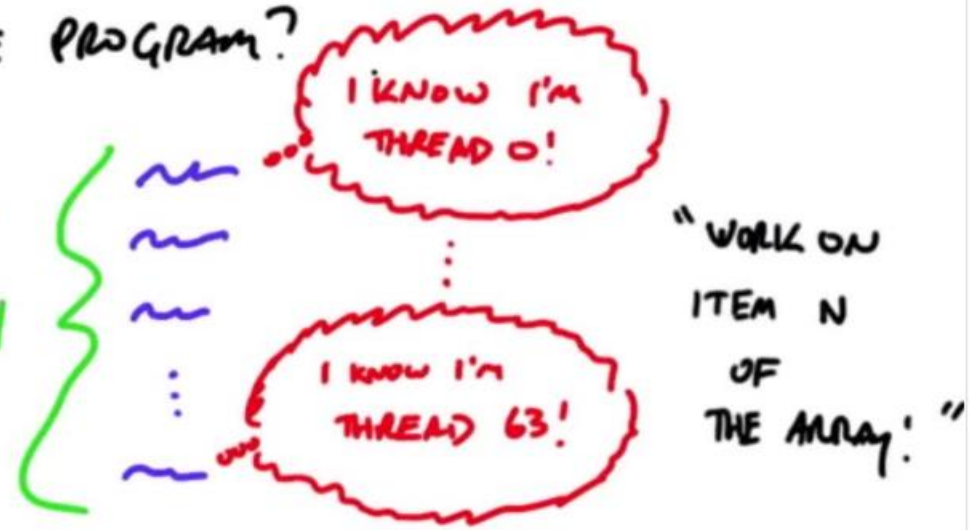
CPU code: square kernel <<< 64 >>> (outArray, inArray)

BUT HOW DOES IT WORK IF I LAUNCH 64 INSTANCES OF THE SAME PROGRAM?



BUT HOW DOES IT WORK IF I LAUNCH 64 INSTANCES OF THE SAME PROGRAM?

CPU LAUNCHES 64 THREADS:



QUIZ:

HOW MANY MULTIPLICATIONS? 64

IF EACH MULT. TAKES 10 ns, HOW LONG FOR THE ENTIRE COMPUTATION? 10

```
8
9  int main(int argc, char ** argv) {
10   const int ARRAY_SIZE = 64;
11   const int ARRAY_BYTES = ARRAY_SIZE * sizeof(float);
12
13   // generate the input array on the host
14   float h_in[ARRAY_SIZE];
15   for (int i = 0; i < ARRAY_SIZE; i++) {
16     h_in[i] = float(i);
17   }
18   float h_out[ARRAY_SIZE];
19
20   // declare GPU memory pointers
21   float * d_in;
22   float * d_out;
23
24   // allocate GPU memory
25   cudaMalloc((void **) &d_in, ARRAY_BYTES);
26   cudaMalloc((void **) &d_out, ARRAY_BYTES);
27
28   // transfer the array to the GPU
29   cudaMemcpy(d_in, h_in, ARRAY_BYTES, cudaMemcpyHostToDevice);
30
31   // launch the kernel
```

Data on the CPU, the host, starts with h underscore. Data on the GPU, the device, starts with d underscore.

```
27
28  // transfer the array to the GPU
29  cudaMemcpy(d_in, h_in, ARRAY_BYTES, ???);
30
31  // launch the kernel
32  cube<<<1, ARRAY_SIZE>>>(d_out, d_in);
33
34  // copy back the result array to the CPU
35  cudaMemcpy(h_out, d_out, ARRAY_BYTES, ???);
36
```

1

cudaMemcpyHostToDevice

2

cudaMemcpyDeviceToHost

CONFIGURING THE KERNEL LAUNCH

```
square<<<1, 64>>>(d_out, d_in)
```

- `1` → NUMBER OF BLOCKS
- `64` → THREADS PER BLOCK

(1) CAN RUN MANY BLOCKS AT ONCE

(2) MAXIMUM NUMBER OF THREADS/BLOCK < 512 (OLDER GPUS) / 1024 (NEWER GPUS)

Two, each block has a maximum number of threads that it can support.

CONFIGURING THE KERNEL LAUNCH

```
square<<<1, 64>>>(d_out, d_in)
```

NUMBER OF BLOCKS

THREADS PER BLOCK

128 THREADS?

```
square<<<1, 128>>>( ... )
```

1280 THREADS?

```
square<<<10, 128>>>( ... )
square<<<5, 256>>>( ... )
~~square<<<1, 1280>>>( ... )~~
```

But we can't call square 1,1280 because that's too many threads per block.

CONFIGURING THE KERNEL LAUNCH

```
square<<< 1, 64 >>> (d_out, d_in)
```

NUMBER OF BLOCKS

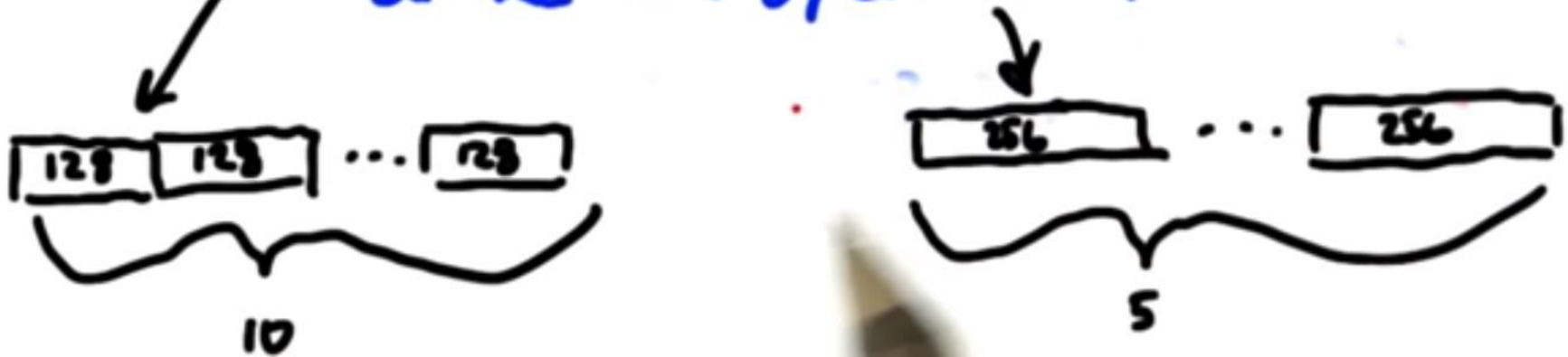
THREADS PER BLOCK

128 THREADS?

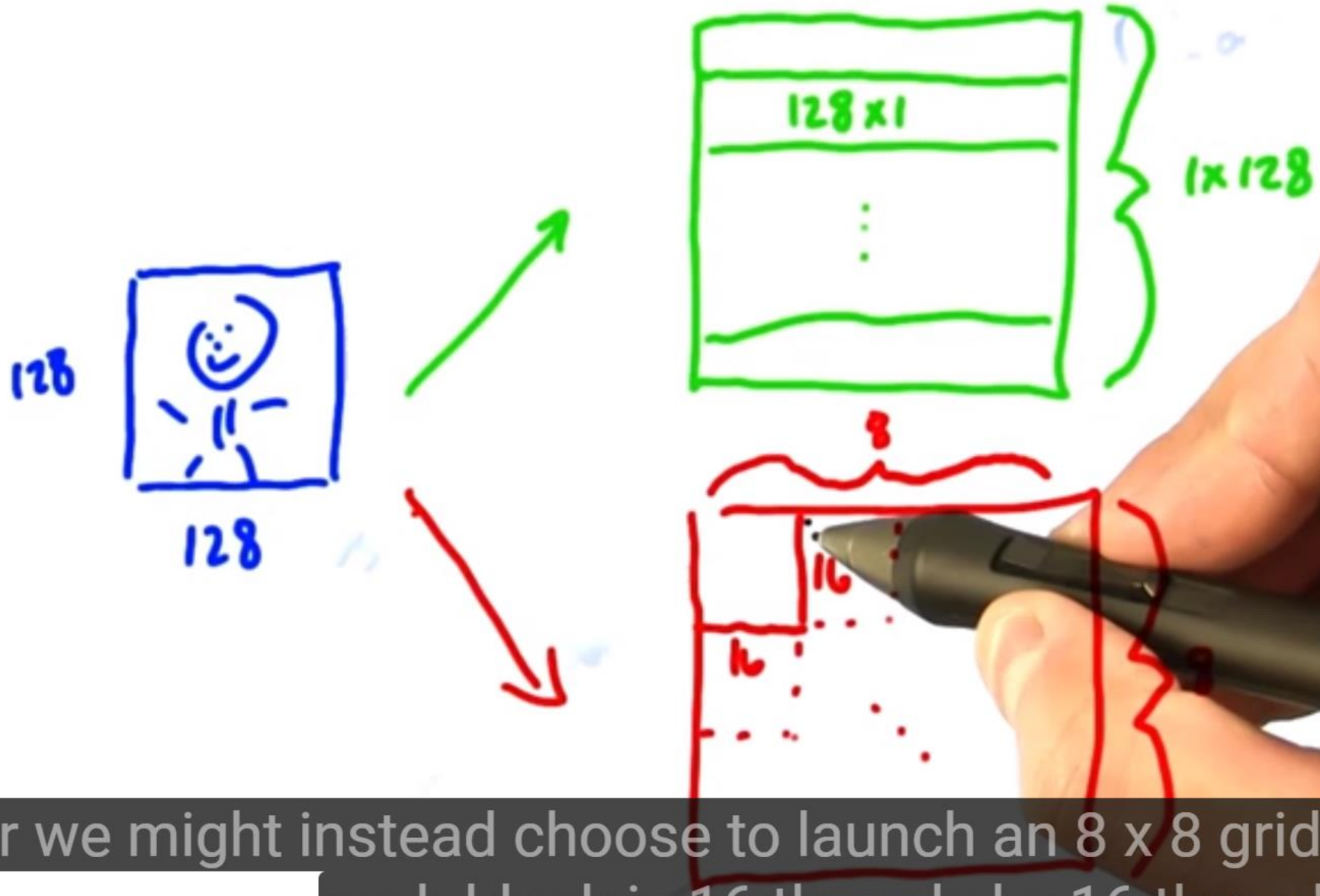
```
square <<< 1, 128 >>> ( ... )
```

1280 THREADS?

```
square <<< 10, 128 >>> ( ... )
square <<< 5, 256 >>> ( ... )
```



CONFIGURING THE KERNEL LAUNCH



Or we might instead choose to launch an 8 x 8 grid of blocks where each block is 16 threads by 16 threads.

CONFIGURING THE KERNEL LAUNCH

KERNEL <<< GRID OF BLOCKS, BLOCK OF THREADS >>> (...)

↓

1, 2, or 3D

So CUDA supports 1, 2, or 3 dimensional thread blocks.

CONFIGURING THE KERNEL LAUNCH

KERNEL <<< GRID OF BLOCKS, BLOCK OF THREADS >>> (...)

1, 2, or 3D

1, 2, or 3D

We can also arrange thread blocks into 1, 2, or 3 dimensional grids.

CONFIGURING THE KERNEL LAUNCH

KERNEL <<< GRID OF BLOCKS, BLOCK OF THREADS >>> (...)

GRID OF BLOCKS: 1, 2, or 3D

BLOCK OF THREADS: 1, 2, or 3D

dim3(x, y, z)

dim3(w, 1, 1) == dim3(w) == w

CONFIGURING THE KERNEL LAUNCH

KERNEL <<< GRID OF BLOCKS, BLOCK OF THREADS >>> (...)

square <<< dim3(bx, by, bz), dim3(tx, ty, tz), shmem >>> (...)

- grid of blocks: bx · by · bz
- block of threads: tx · ty · tz
- shared memory per block in bytes

The most general kernel launch we can do looks like thi:, square of 3 parameters.

CONFIGURING THE KERNEL LAUNCH

KERNEL <<< GRID OF BLOCKS, BLOCK OF THREADS >>> (...)

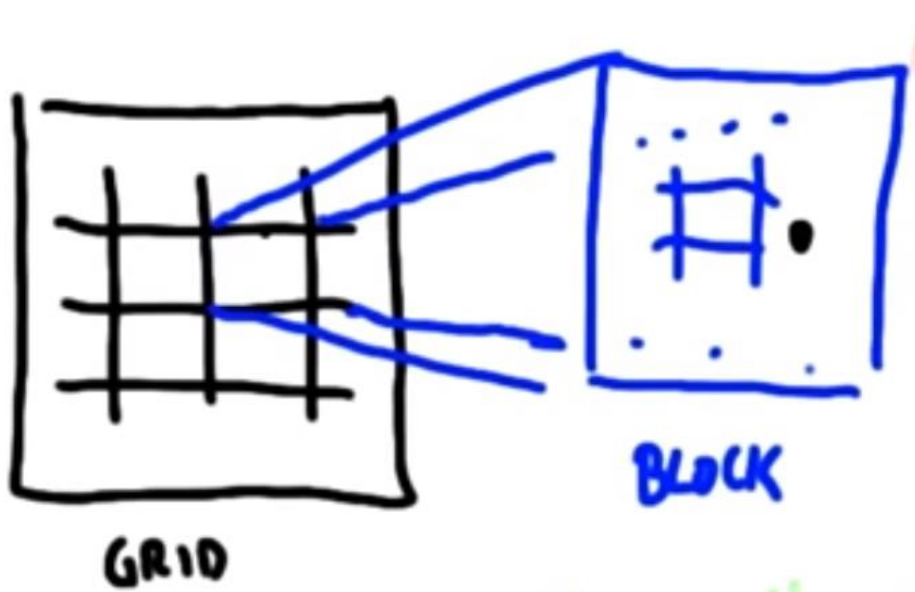
square <<< dim3(bx, by, bz), dim3(tx, ty, tz), shmem >>> (...)

- grid of blocks: bx · by · bz
- block of threads: tx · ty · tz
- shared memory per block in bytes

CONFIGURING THE KERNEL LAUNCH

KERNEL <<< GRID OF BLOCKS, BLOCK OF THREADS >>> (...)

square <<< dim3(bx, by, bz), dim3(tx, ty, tz), shmem >>> (...)



threadIdx : thread within block
threadIdx.x threadIdx.y

blockDim : size of a block

blockIdx : block within grid

gridDim : size of grid

CONFIGURING THE KERNEL LAUNCH

KERNEL <<< GRID OF BLOCKS, BLOCK OF THREADS >>> (...)

square <<< dim3(bx, by, bz), dim3(tx, ty, tz), shmem >>> (...)

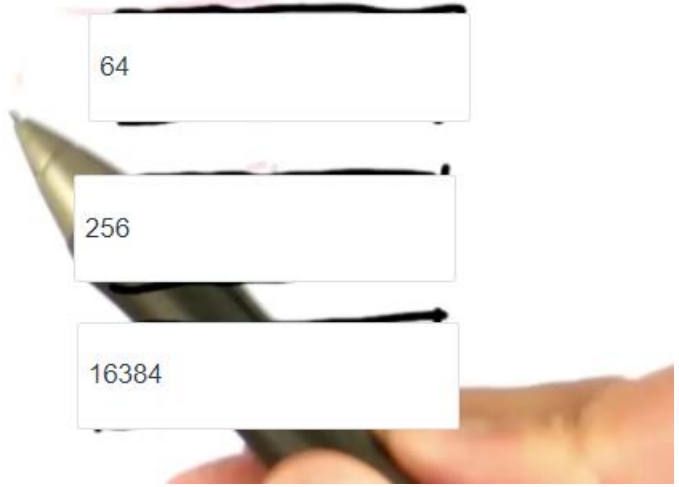
Quiz

kernel <<< dim3(8,4,2), dim3(16,16) >>> (...)

How many blocks? 64

How many threads/block? 256

How many total threads? 16384



LESSONS FOR TODAY: WHAT WE KNOW

- WE WRITE A PROGRAM THAT LOOKS LIKE IT RUNS ON ONE THREAD

- WE CAN LAUNCH THAT PROGRAM ON ANY NUMBER OF THREADS

- EACH THREAD KNOWS ITS OWN INDEX IN THE BLOCK + THE GRID

MAP

- SET OF ELEMENTS TO PROCESS [64 FLOATS]
- FUNCTION TO RUN ON EACH ELEMENT ["SQUARE"]

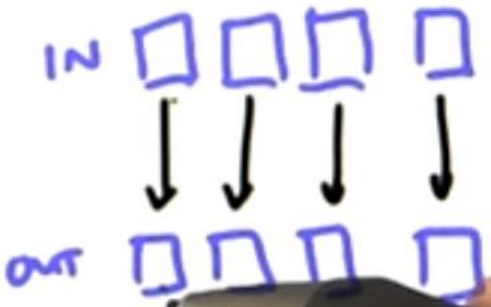
MAP(ELEMENTS, FUNCTION)

GPUS ARE GOOD AT MAP

— GPUS HAVE MANY PARALLEL PROCESSORS

— GPUS OPTIMIZE FOR THROUGHPUT

MAP'S COMMUNICATION PATTERN



Quiz CHECK THE PROBLEMS THAT CAN BE SOLVED USING MAP.

- [ ] SORT AN INPUT ARRAY
- [x] ADD ONE TO EACH ELEMENT IN AN INPUT ARRAY
- [ ] SUM UP ALL ELEMENTS IN AN INPUT ARRAY
- [ ] COMPUTE THE AVERAGE OF AN INPUT ARRAY

WHAT WE LEARNED

- TECHNOLOGY TRENDS
- THROUGHPUT VS LATENCY
- GPU DESIGN GOALS
- GPU PROGRAMMING MODEL
  - WITH EXAMPLE!
- MAP