Parallel Programming

Software Project, Spring 2018

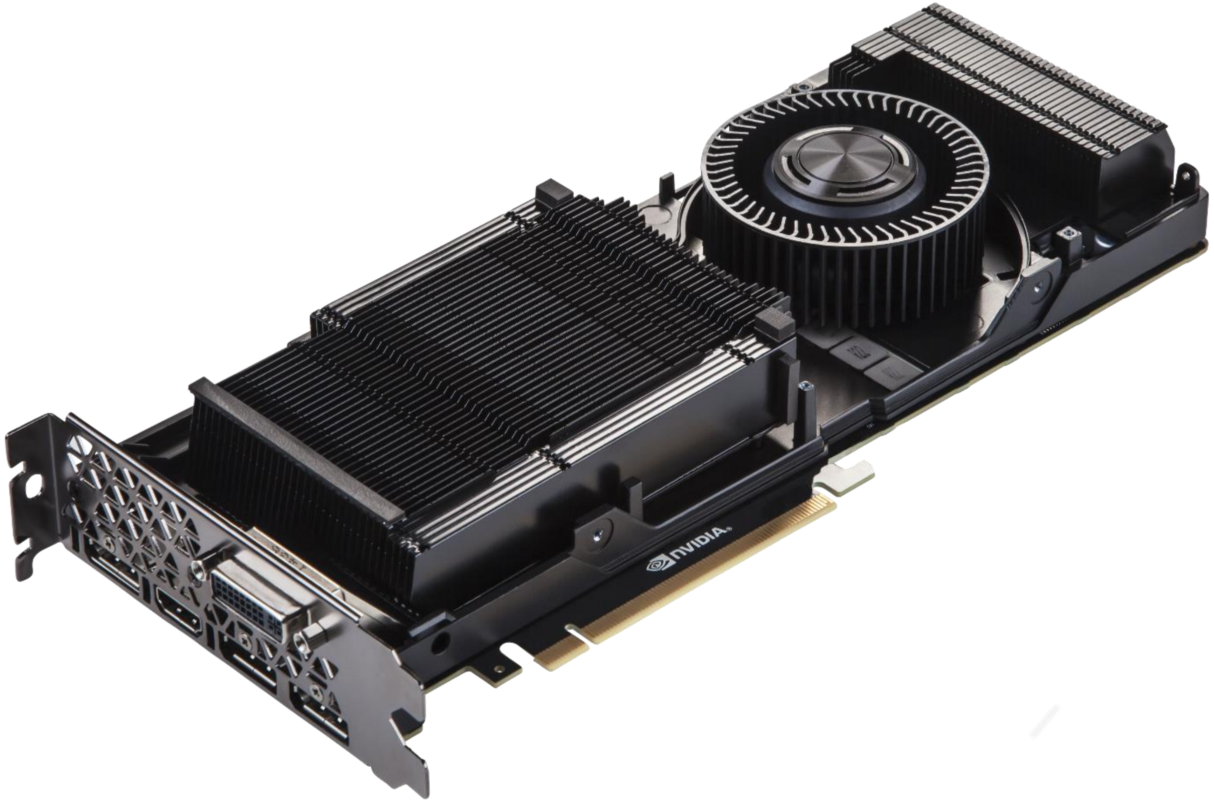
0368-2161

The Power of Parallelism

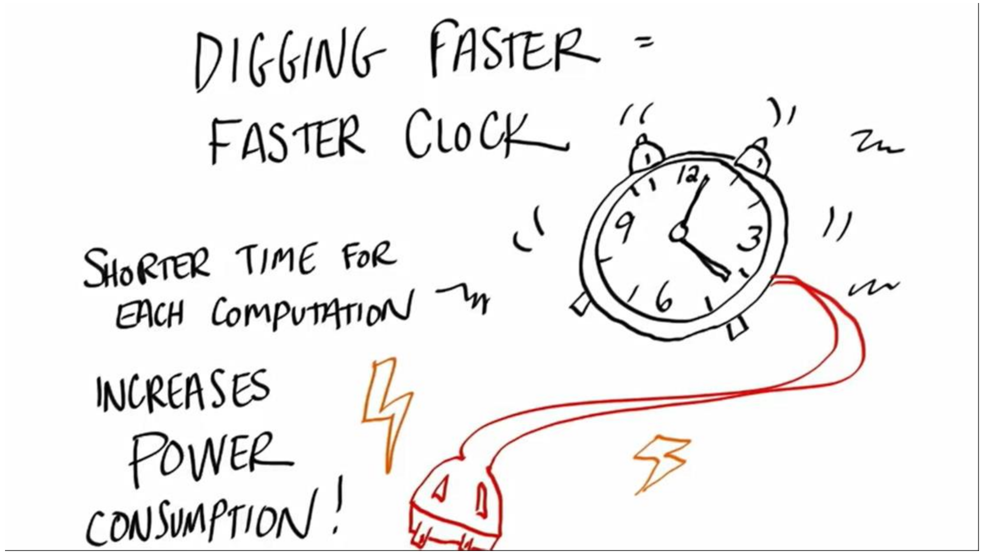
- Given a nXm gray level image, generate a binary image so that the value if any pixel is 1 if and only if the gray level of the original pixel is great than a given threshold.

- Algorithm?
- Time?
- Can we improve?

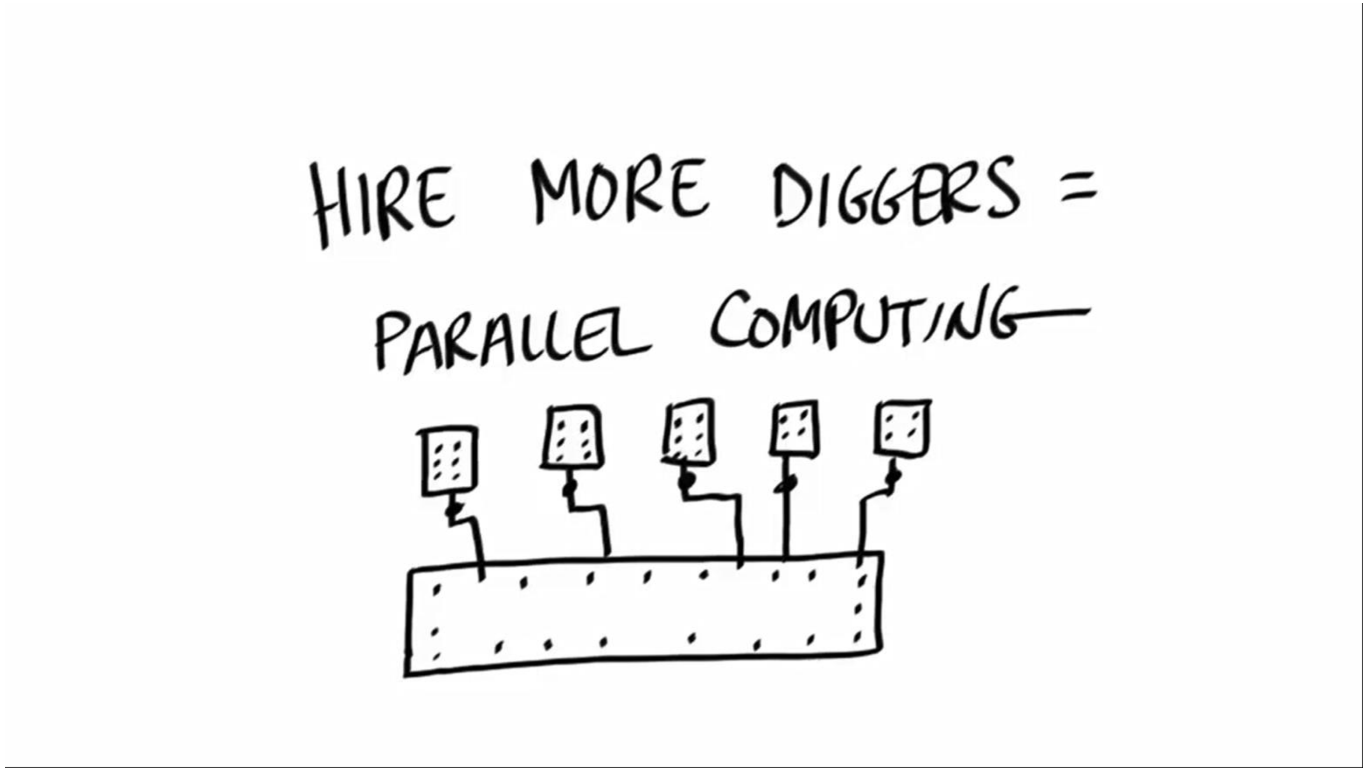
GPU – Graphics Processing Unit



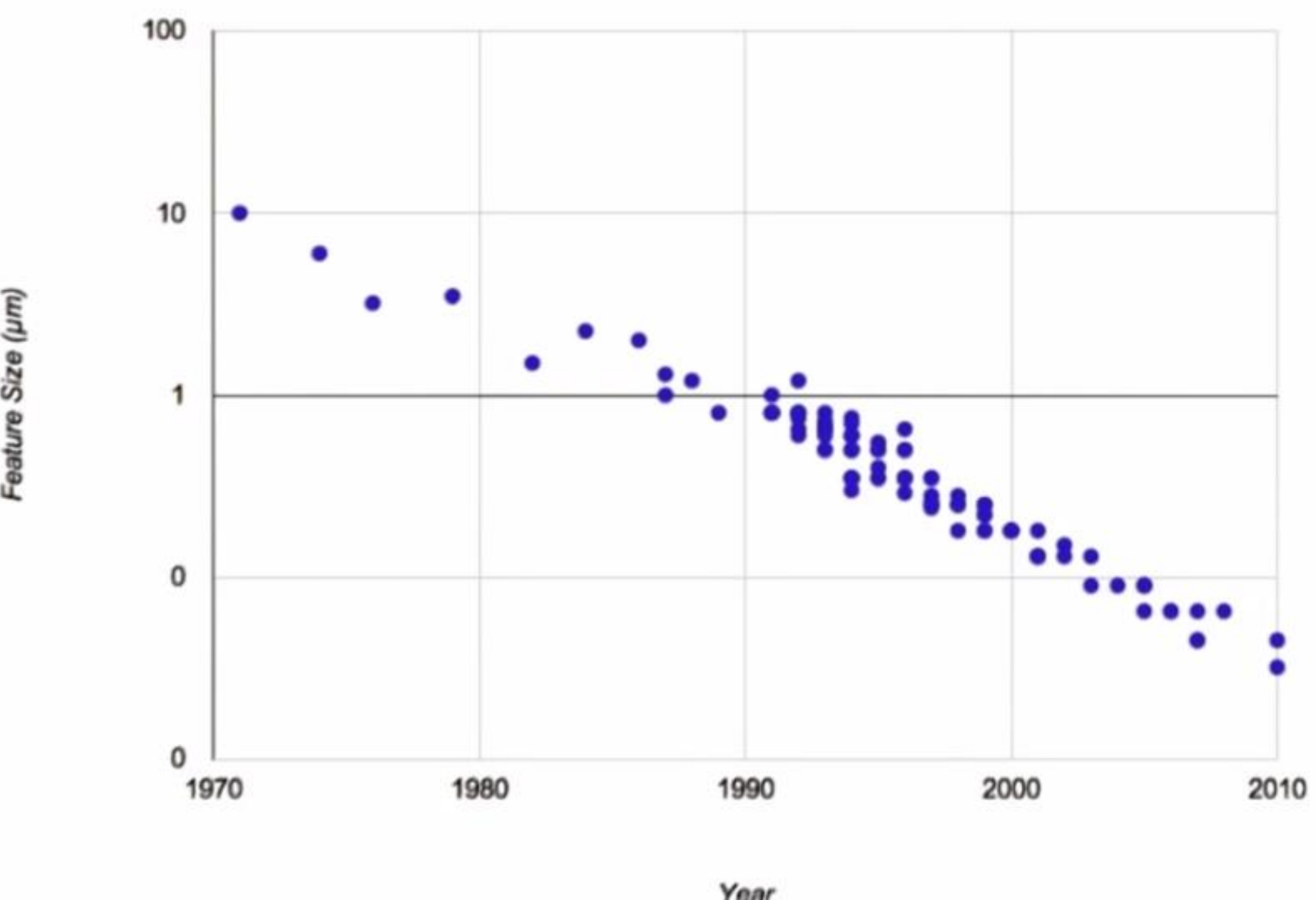

WORLD'S
FASTEST
HOLE
TO
CHINA


DIGGING FASTER =
FASTER CLOCK
SHORTER TIME FOR EACH COMPUTATION
INCREASES POWER CONSUMPTION!


BUYING A MORE PRODUCTIVE SHOVEL = MORE WORK PER STEP
$

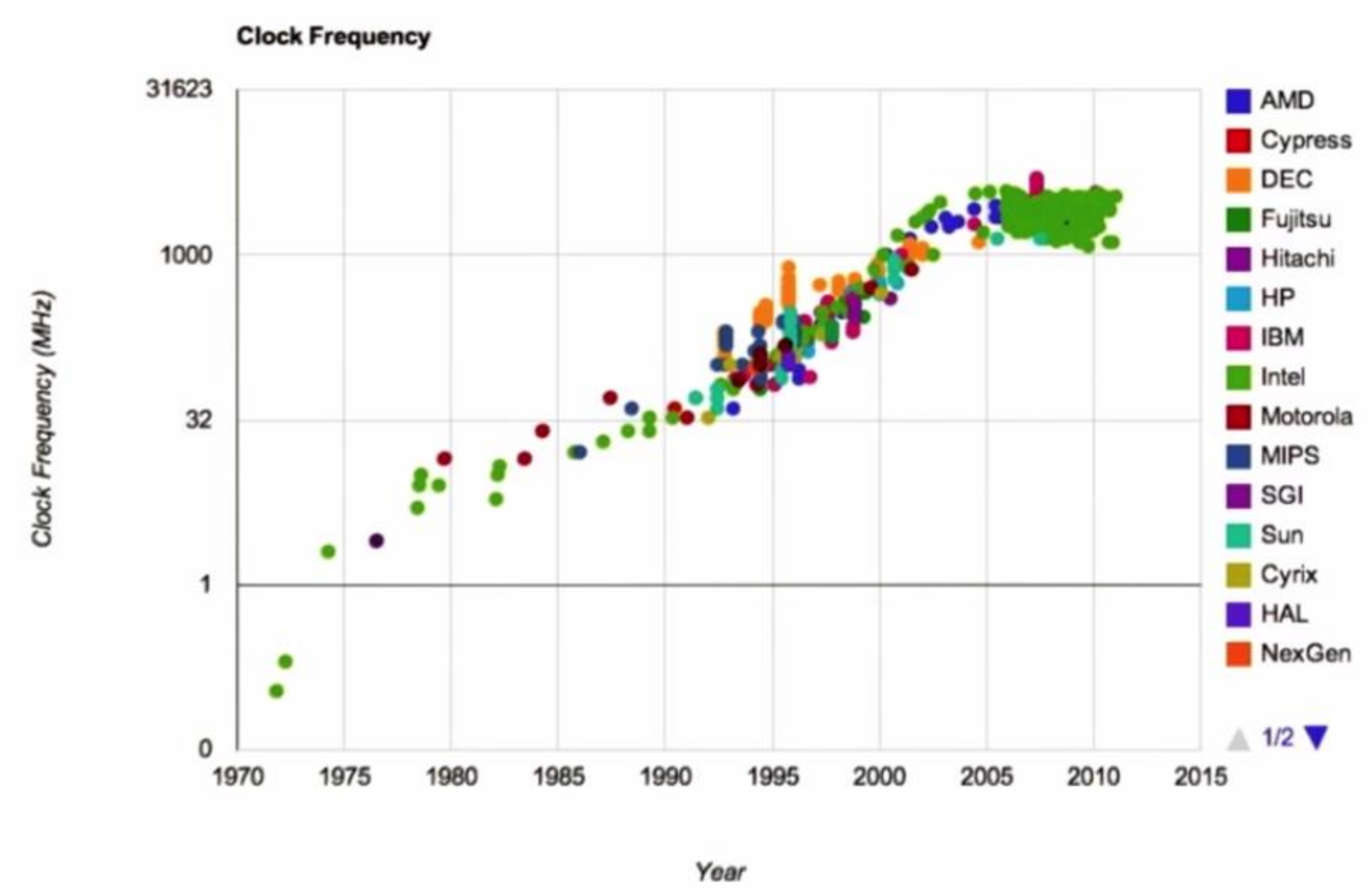

HIRE MORE DIGGERS =
PARALLEL COMPUTING

CPU Over the Years



- Smaller
- Faster
- Less power
- More on a chip

Clock Frequency Over the Years



Improvement stopped because:

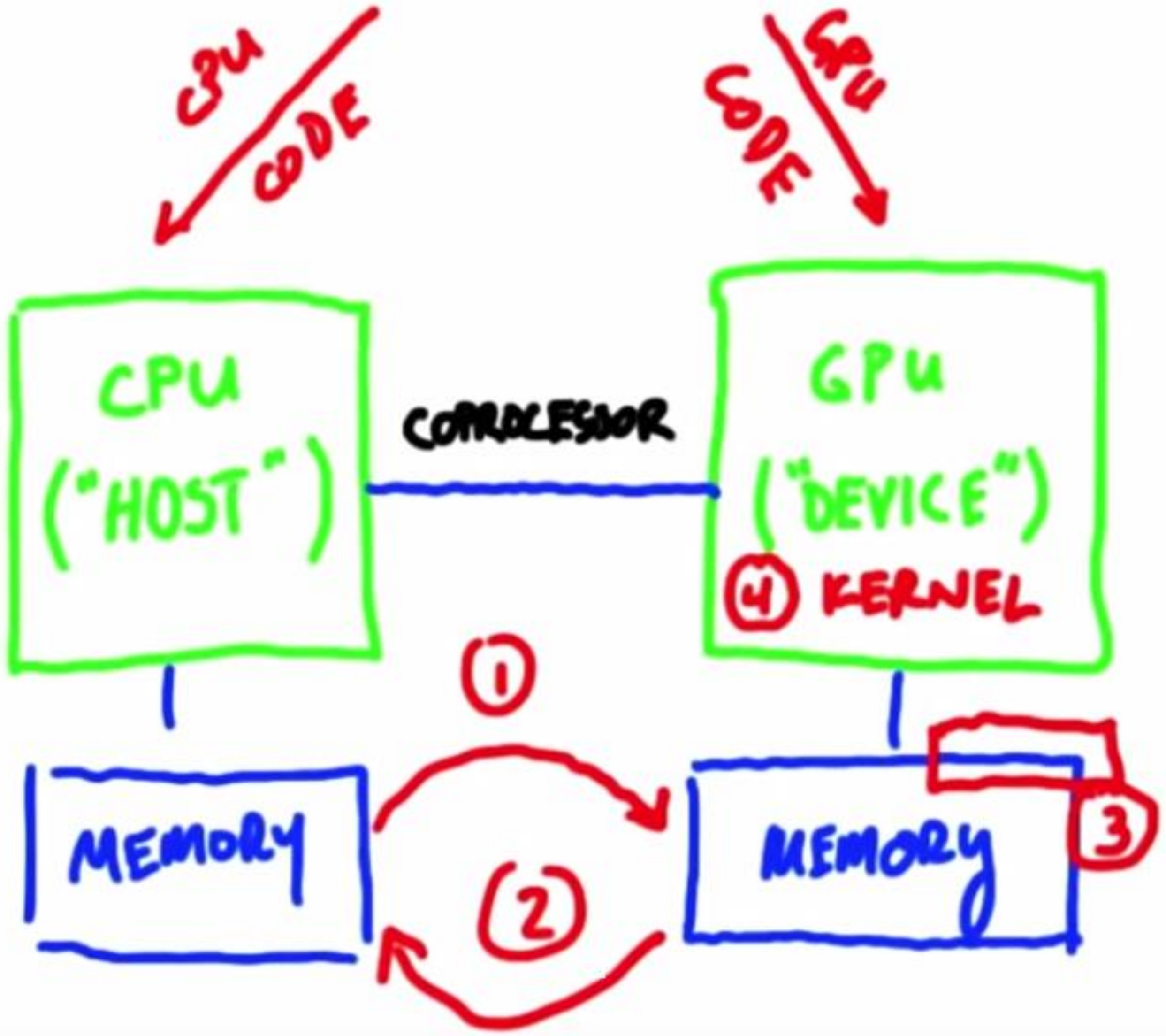
- Heat
- Need a lot of power

Solution:

- SMALLER, MORE EFFICIENT PROCESSORS
- MORE OF THEM.

CUDA PROGRAM

WRITTEN IN C WITH EXTENSIONS



1 DATA CPU → GPU

2 DATA GPU → CPU

1, 2: cudaMemcpy

3 ALLOCATE GPU MEMORY

3 cudaMalloc

4 LAUNCH KERNEL ON GPU

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

```
#include <stdio.h>

__global__ void square(float * d_out, float * d_in) {
  int idx = threadIdx.x;
  float f = d_in[idx];
  d_out[idx] = f * f;
}

int main(int argc, char ** argv) {
  const int ARRAY_SIZE = 64;
  const int ARRAY_BYTES = ARRAY_SIZE * sizeof(float);

  // generate the input array on the host
  float h_in[ARRAY_SIZE];
  for (int i = 0; i < ARRAY_SIZE; i++) {
    h_in[i] = float(i);
  }
  float h_out[ARRAY_SIZE];

  // declare GPU memory pointers
  float * d_in;
  float * d_out;

  // allocate GPU memory
  cudaMalloc((void **) &d_in, ARRAY_BYTES);
  cudaMalloc((void **) &d_out, ARRAY_BYTES);

  // transfer the array to the GPU
  cudaMemcpy(d_in, h_in, ARRAY_BYTES, cudaMemcpyHostToDevice);

  // launch the kernel
  square<<<1, ARRAY_SIZE>>>(d_out, d_in);

  // copy back the result array to the CPU
  cudaMemcpy(h_out, d_out, ARRAY_BYTES, cudaMemcpyDeviceToHost);

  // print out the resulting array
  for (int i = 0; i < ARRAY_SIZE; i++) {
    printf("%f", h_out[i]);
    printf(((i % 4) != 3) ? "\t" : "\n");
  }

  // free GPU memory allocation
  cudaFree(d_in);
  cudaFree(d_out);

  return 0;
}
```

```
$ nvcc -o square square.cu
$ ./square
0.000000        1.000000        4.000000        9.000000
16.000000       25.000000       36.000000       49.000000
64.000000       81.000000       100.000000      121.000000
144.000000      169.000000      196.000000      225.000000
256.000000      289.000000      324.000000      361.000000
400.000000      441.000000      484.000000      529.000000
576.000000      625.000000      676.000000      729.000000
784.000000      841.000000      900.000000      961.000000
1024.000000     1089.000000     1156.000000     1225.000000
1296.000000     1369.000000     1444.000000     1521.000000
1600.000000     1681.000000     1764.000000     1849.000000
1936.000000     2025.000000     2116.000000     2209.000000
2304.000000     2401.000000     2500.000000     2601.000000
2704.000000     2809.000000     2916.000000     3025.000000
3136.000000     3249.000000     3364.000000     3481.000000
3600.000000     3721.000000     3844.000000     3969.000000
$
```

A TYPICAL GPU PROGRAM

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

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

CPU is good at:

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

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?

CPU LAUNCHES 64 THREADS:

"WORK ON ITEM N OF THE ARRAY!"

CONFIGURING THE KERNEL LAUNCH

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

NUMBER OF BLOCKS

THREADS PER BLOCK

1. CAN RUN MANY BLOCKS AT ONCE
2. MAXIMUM NUMBER OF THREADS/BLOCK
   - 512 (OLDER GPUS)
   - 1024 (NEWER GPUS)

CONFIGURING THE KERNEL LAUNCH

SQUARE<<<1, 64>>>(d_out, d_in)

NUMBER OF BLOCKS

THREADS PER BLOCK

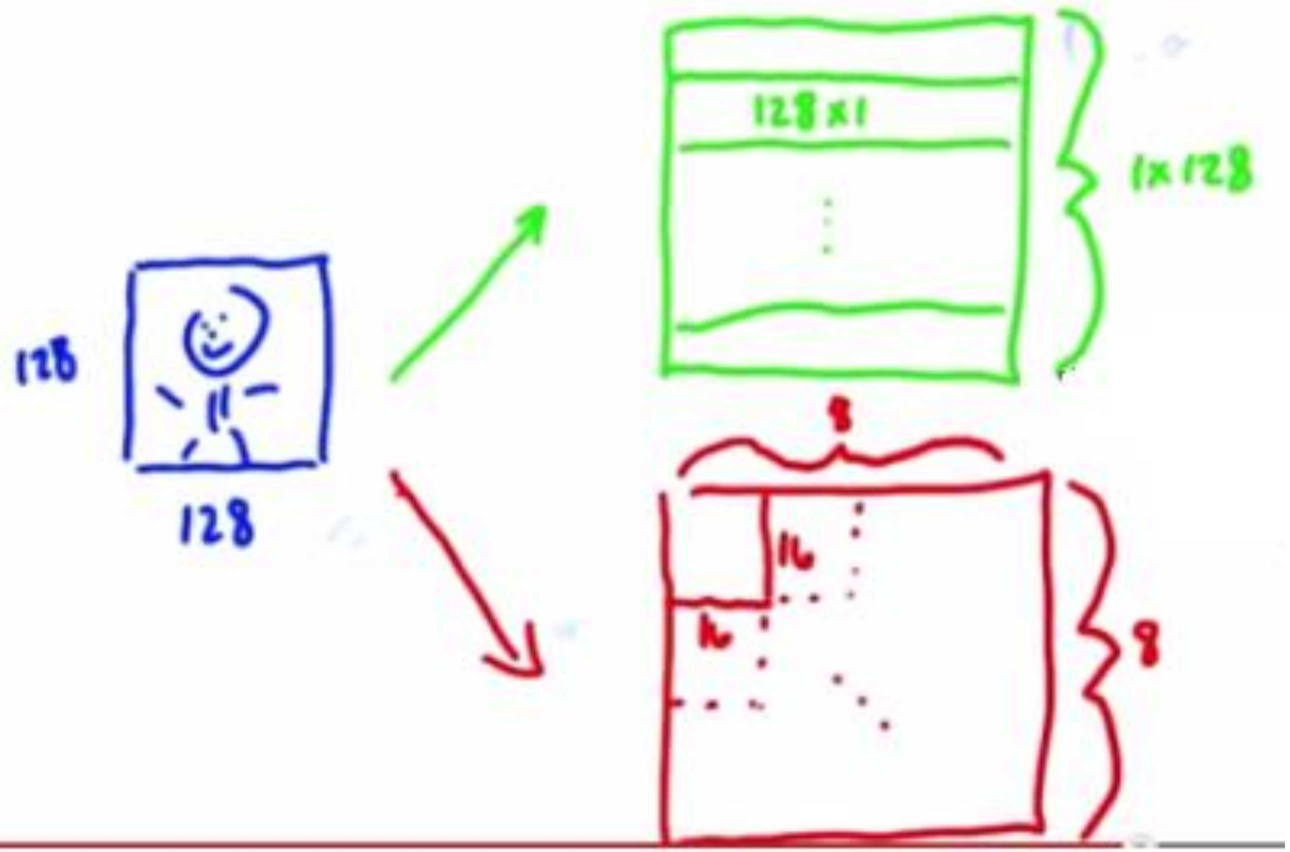
128 THREADS? SQUARE<<<1, 128>>>(...)

1280 THREADS? SQUARE<<<10, 128>>>(...)

SQUARE<<<5, 256>>>(...)

~~SQUARE<<<1, 1280>>>(...)~~

CONFIGURING THE KERNEL LAUNCH



CONFIGURING THE KERNEL LAUNCH

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

$\downarrow$ 1, 2, or 3D $\downarrow$ 1, 2, or 3D

dim3(x, y, z)

$\text{dim3}(w,1,1) == \underbrace{\text{dim3}(w)} == \underline{w}$

```
square<<<1, 64>>> == square<<<dim3(1,1,1),
                              dim3(64,1,1)>>>
```

CONFIGURING THE KERNEL LAUNCH

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

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

grid of blocks
$bx \cdot by \cdot bz$

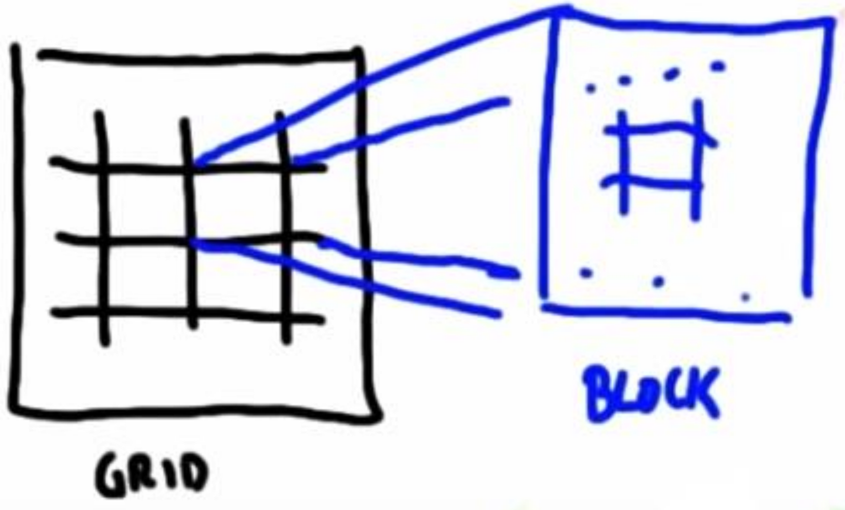
block of threads
$tx \cdot ty \cdot tz$

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 >>> (...)



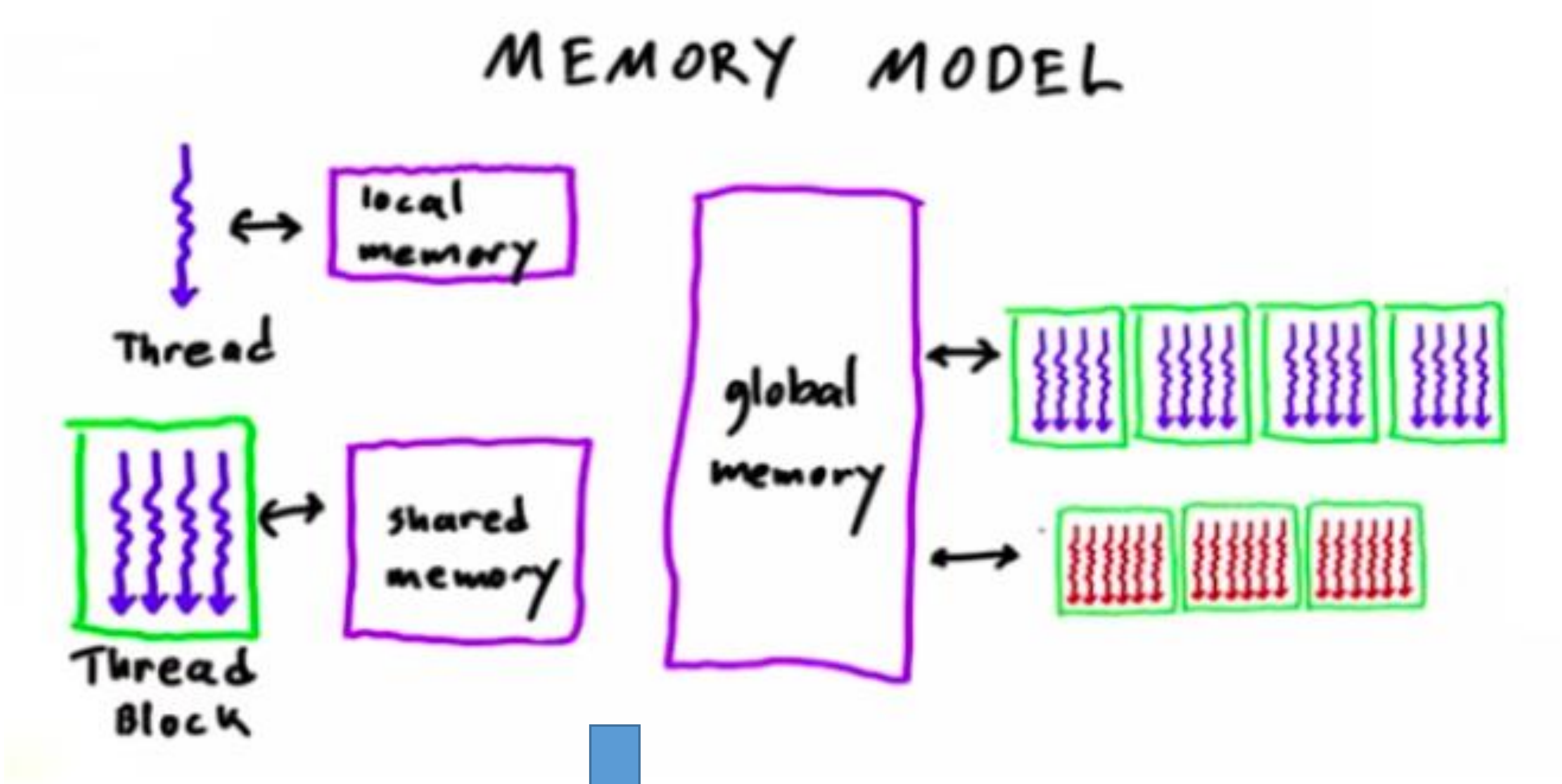
threadIdx : thread within block

threadIdx.x   threadIdx.y

blockDim : size of a block

blockIdx : block within grid

gridDim : size of grid




```
__shared__ int array[128];
```

Synchronization

threads can access each other's results through shared and global memory

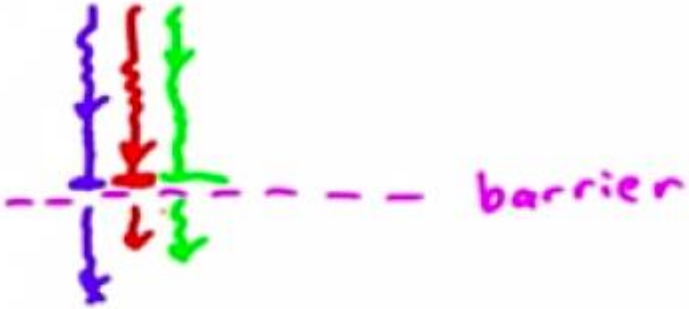
$\Rightarrow$ they can work together!

Danger: what if a thread reads a result before another thread writes it?

Threads need to synchronize.

Barrier - point in the program where threads stop and wait.

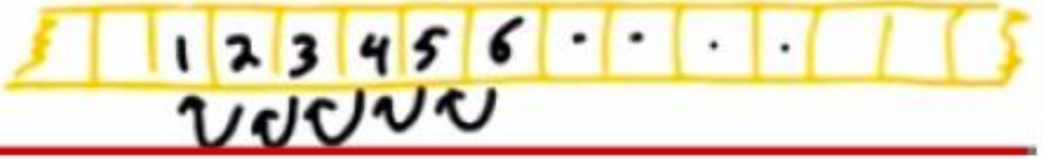
when all threads have reached the barrier, they can proceed.



The need for barriers

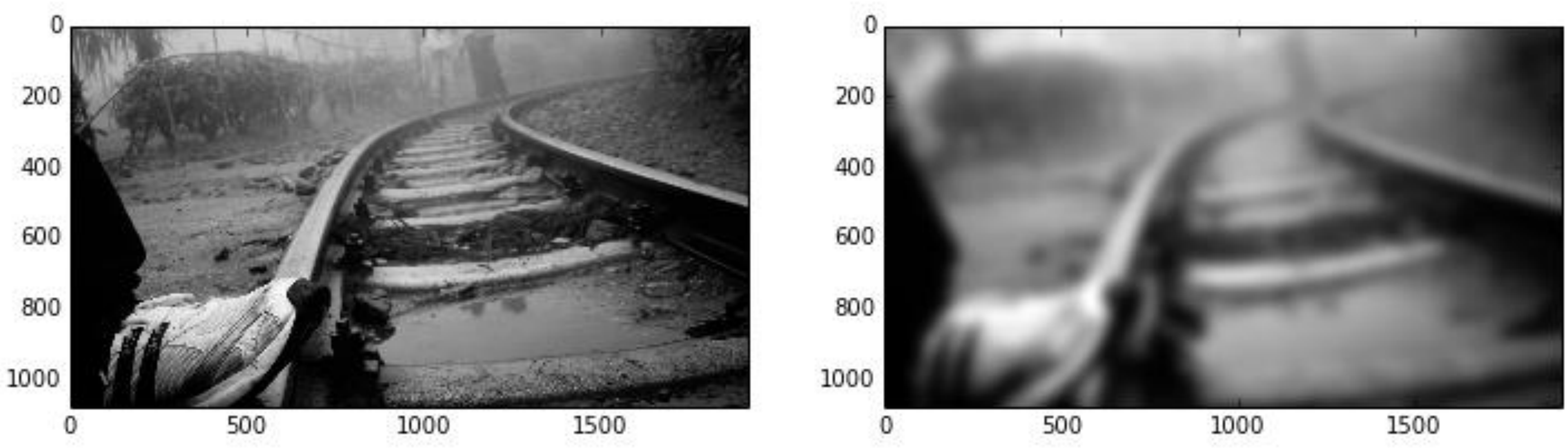
```
⋮
int idx = threadIdx.x
__shared__ int array[128];
array[idx] = threadIdx.x;
if (idx < 127)
    array[idx] = array[idx+1];
⋮
```

Quiz: how many barriers does this code need?



```
int idx = threadIdx.x;
__shared__ int array[128];
array[idx] = threadIdx.x;
__syncthreads();
if (idx < 127) {
    int temp = array[idx+1];
    __syncthreads();
    array[idx] = temp;
    __syncthreads();
}
```

Blurring Images in Parallel



- Use an average filter to blur a gray level image
- Note the synchronization

```
int main()
    ...
    char* imageHost = readImage();
    int height = getImageHeight();
    int width = getImageWidth();
    int size = height * width;

    char* imageDevice;
    cudaMalloc((void**)&imageDevice, size);

    cudaMemcpy(imageDevice, imageHost, size, cudaMemcpyHostToDevice);

    const int y1 = (int)(ceil(height / 32.0));
    const int x1 = (int)(ceil(width / 32.0));
    const int y2 = 32;
    const int x2 = 32;

    blurImage<<<dim3(x1, y1), dim3(x2, y2) >> >(imageDevice, height, width);

    char* imageHostOut = (char*) malloc(size * sizeof(char));
    cudaMemcpy(imageHostOut, imageDevice, size, cudaMemcpyDeviceToHost);
    ...
}
```

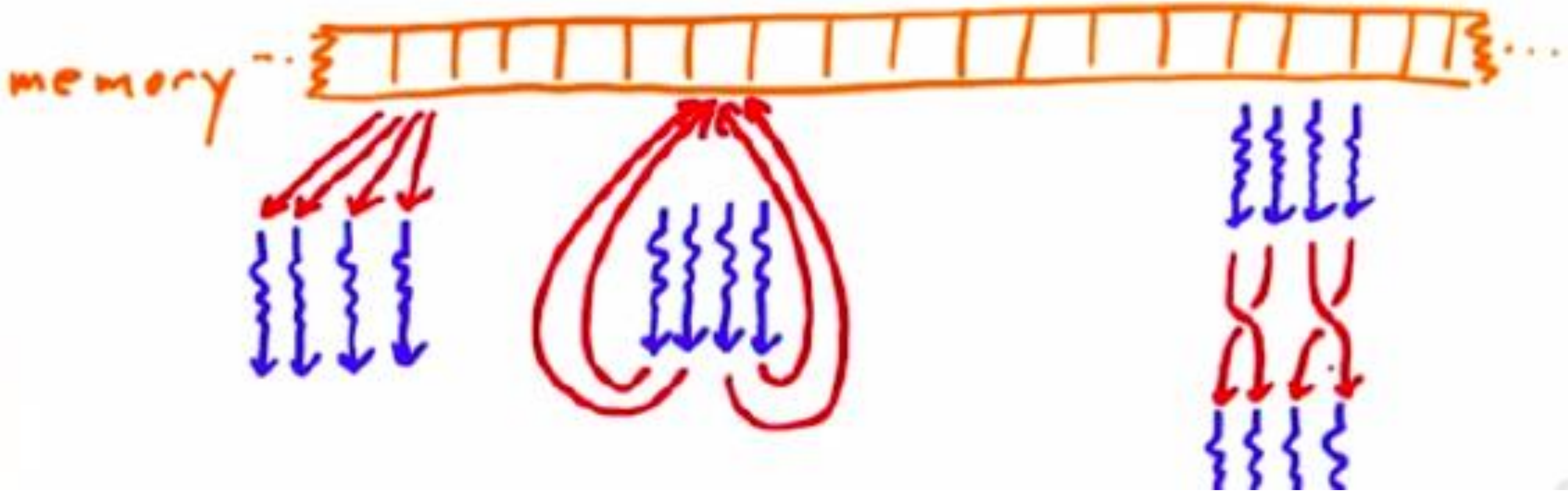
```
__global__
void blurImage(char* image, int height, int width)
{
    int y = blockIdx.y * blockDim.y + threadIdx.y;
    int x = blockIdx.x * blockDim.x + threadIdx.x;

    if (height <= x + 1 || width <= x + 1 || height == 0 || width == 0) {
        return;
    }

    char average = getAverage(x, y, width, image);
    __syncthreads();
    image[y * width + x] = average;
}
```

Tells the compiler that this is a function that runs on GPU

```
__device__
char getAverage(int x, int y, int width, char* image)
{
	int sum = 0;

	for(int i = -1; i <= 1; ++i) {
		for(int j = -1; j <= 1; ++j) {
			sum += image[i * width + j];
		}
	}

	sum /= 9;

	return sum;
}
```

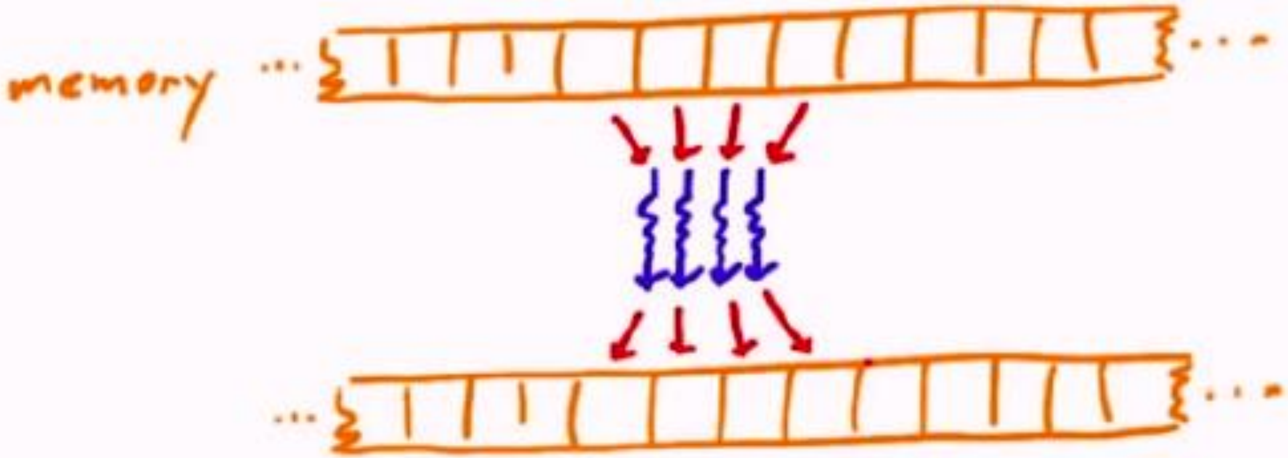
Parallel computing: many threads solving a problem by working together

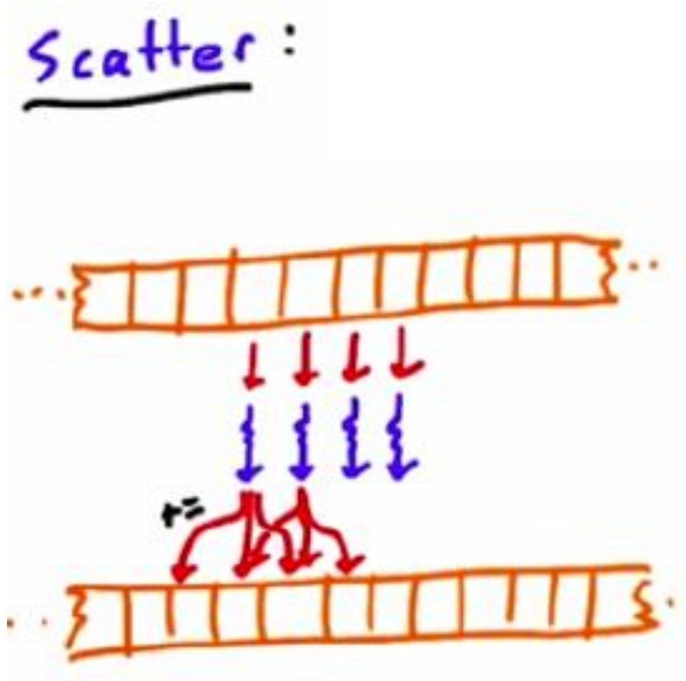
communication!



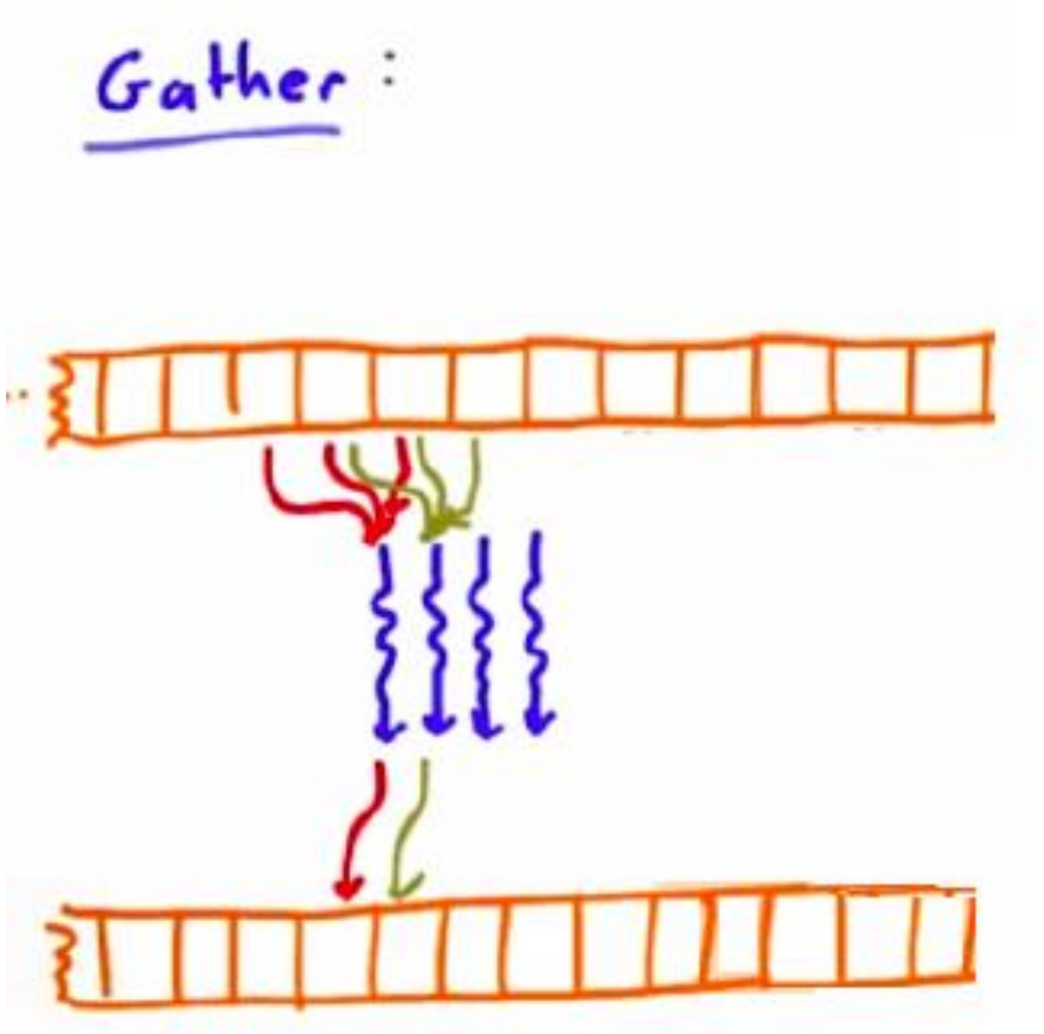
Parallel Communication Patterns

Map: Tasks read from and write to specific data elements





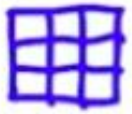
Be careful of race conditions!



Stencil: tasks read input from a fixed neighborhood in an array.

stencil

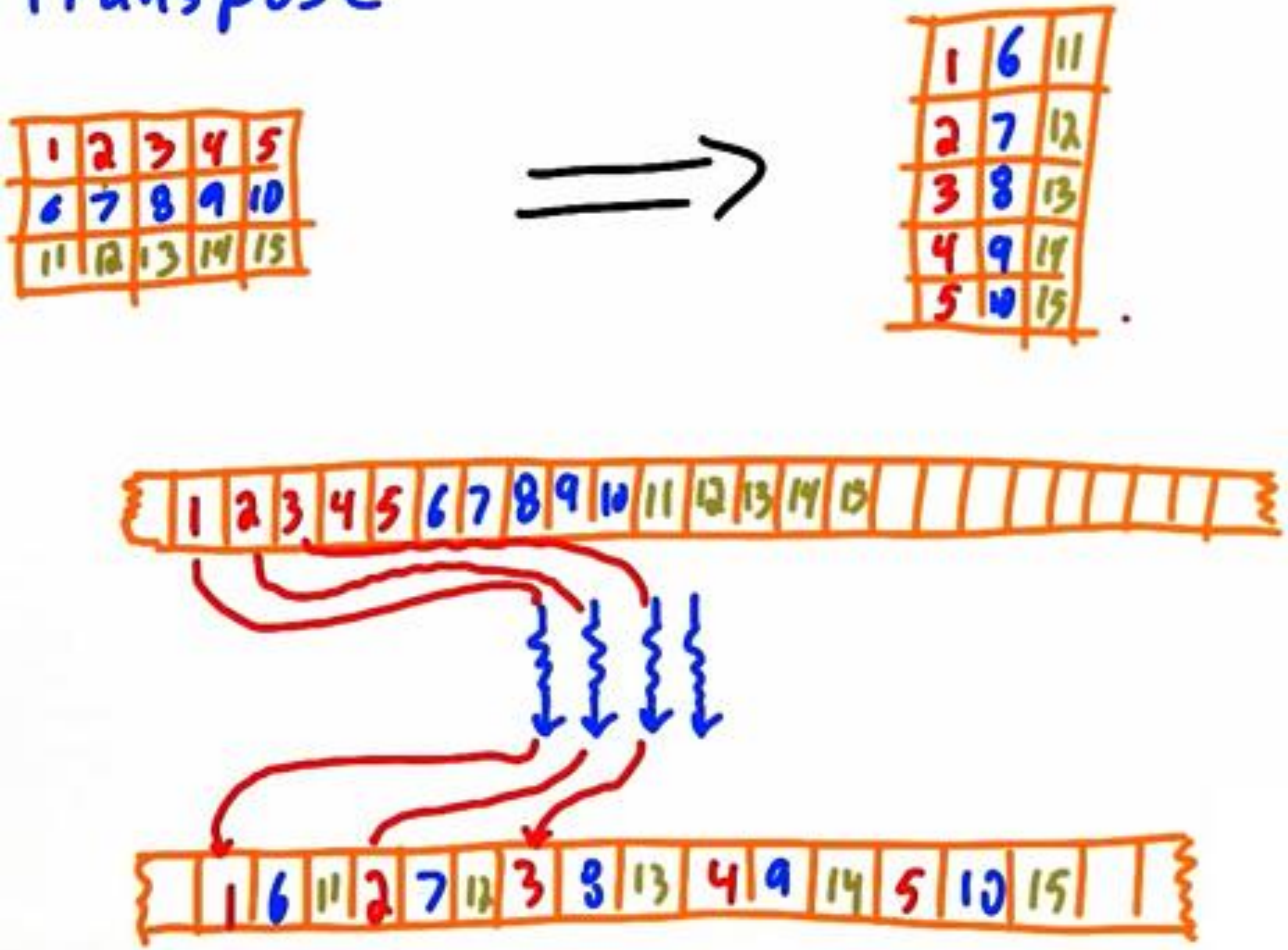
2D von Neumann

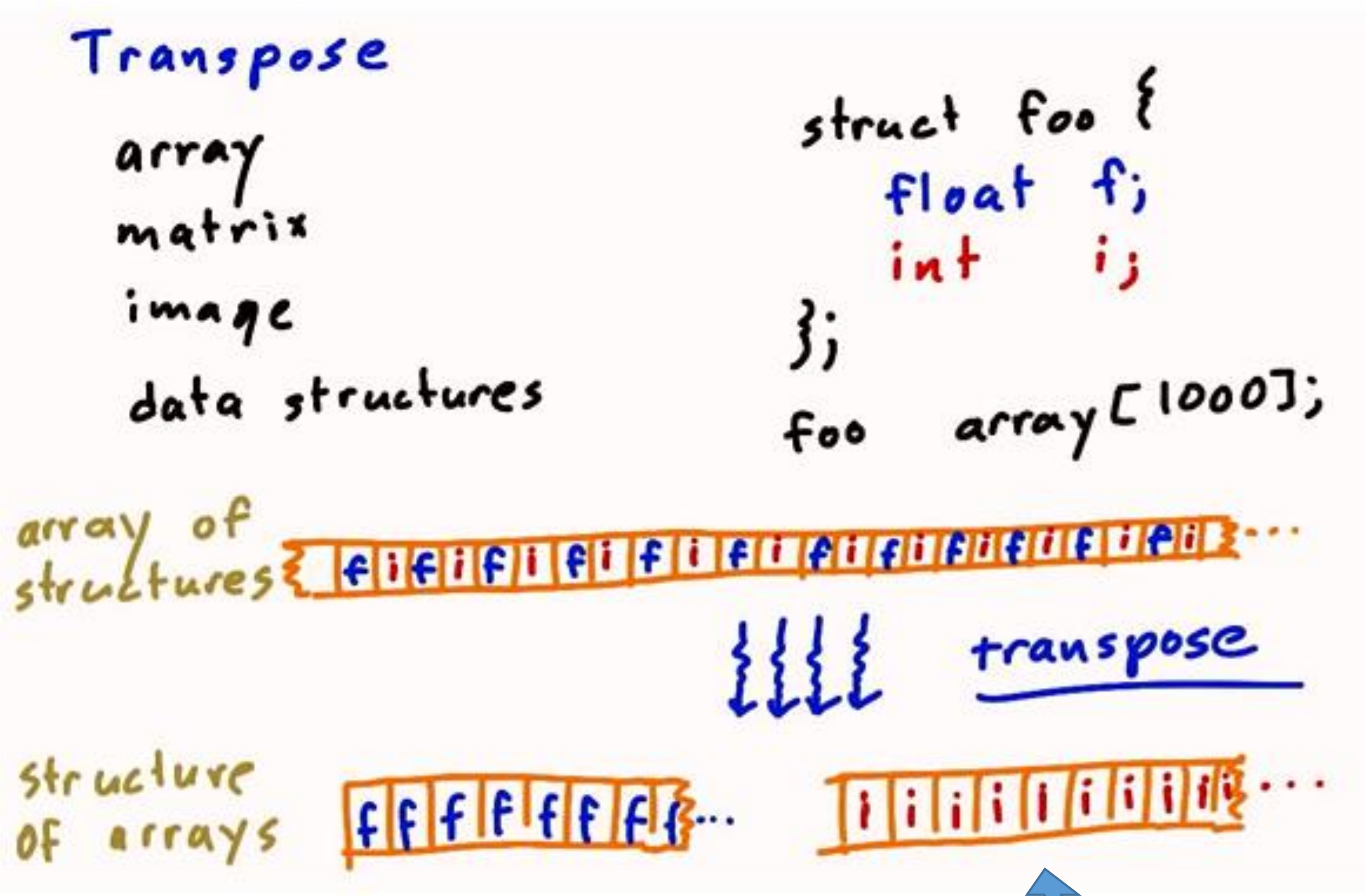


2D Moore

3D von Neumann

Transpose


1 2 3 4 5
6 7 8 9 10
11 12 13 14 15
1 6 11
2 7 12
3 8 13
4 9 14
5 10 15
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
1 6 11 2 7 12 3 8 13 4 9 14 5 10 15



Localization!

```
float out[], in[];
int i = threadIdx.x;
int j = threadIdx.y;

const float pi = 3.1415;

___ out[i] = pi * in[i];

___ out[i + j*128] = in[j + i*128];

if (i % 2) {
___     out[i-1] += pi * in[i]; out[i+1] += pi * in[i];

___     out[i] = (in[i] + in[i-1] + in[i+1]) * pi / 3.0f;
```

A. Map

B. Gather

C. Scatter

D. Stencil

E. Transpose

```
float out[], in[];
int i = threadIdx.x;
int j = threadIdx.y;

const float pi = 3.1415;
```

A

```
out[i] = pi * in[i];
```

E

```
out[i + j*128] = in[j + i*128];
```

C

```
if (i % 2) {
    out[i-1] += pi * in[i]; out[i+1] += pi * in[i];
```

B

```
    out[i] = (in[i] + in[i-1] + in[i+1]) * pi / 3.0f;
```

A. Map

B. Gather

C. Scatter

D. Stencil

E. Transpose

Not a stencil because
not working on each cell

Parallel Communication Patterns

Map	one-to-one	
Transpose	one-to-one	
Gather	many-to-one	
Scatter	one-to-many	
Stencil	several-to-one