

# CS344 Introduction to Parallel Programming

## Lesson 1: The GPU Programming Model

### L1-1.3- Why Are You Taking This Class? Quiz 1

OPTIONAL BEFORE WE START THE CLASS,  
TYPE A FEW SENTENCES ABOUT WHY YOU'RE HERE.



### L1-1.5-How To Make Computers Run Faster Quiz 2

QUIZ WHAT ARE 3 TRADITIONAL WAYS HW DESIGNERS MAKE COMPUTERS RUN FASTER?

- FASTER CLOCKS       LARGER HARD DISK
- LONGER CLOCK PERIOD       MORE PROCESSORS
- MORE WORK/CLOCK CYCLE       REDUCE AMOUNT OF MEMORY

### L1-1.6- Chickens or Oxen

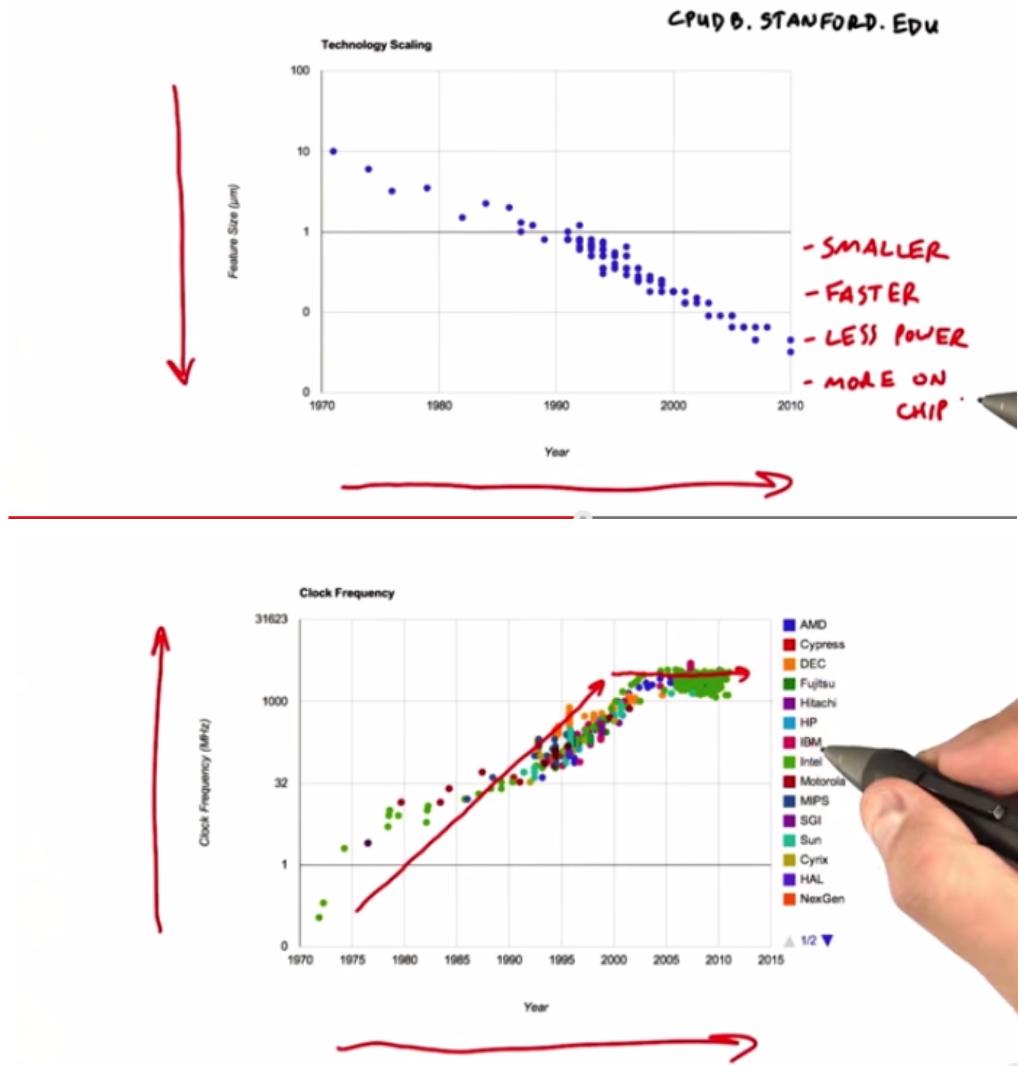
SEYMOUR CRAY: WOULD YOU RATHER PLOW A FIELD WITH TWO STRONG OXEN OR 1024 CHICKENS?

I ❤️ CHICKENS!

MODERN GPU: - THOUSANDS OF ALUs  
- HUNDREDS OF PROCESSORS  
- TENS OF THOUSANDS OF CONCURRENT THREADS

THIS CLASS: LEARN TO THINK IN PARALLEL  
(LIKE THE CHICKENS)

### L1-1.7- CPU Speed Remaining Flat



### L1-1.8-How Are CPUs Getting Faster? Quiz 3

QUIZ

ARE PROCESSORS TODAY GETTING FASTER BECAUSE

- WE'RE CLOCKING THEIR TRANSISTORS FASTER
- WE HAVE MORE TRANSISTORS AVAILABLE FOR COMPUTATION?

### L1-1.9-Why We Cannot Keep Increasing CPU Speed?

WHY DON'T WE KEEP INCREASING CLOCK SPEED?

HAVE TRANSISTORS STOPPED GETTING SMALLER + FASTER?

NO.

(INSTEAD) HEAT!



WHAT MATTERS TODAY: POWER!

CONSEQUENCE:

- SMALLER, MORE EFFICIENT PROCESSORS
- MORE OF THEM.

### L1-1.10-What Kind of Processors Are We Building?

WHAT KIND OF PROCESSORS WILL WE BUILD?

(MAJOR DESIGN CONSTRAINT: POWER.)

CPU: — COMPLEX CONTROL HARDWARE  
↑ FLEXIBILITY + PERFORMANCE!  
↓ EXPENSIVE IN TERMS OF POWER

GPU: — SIMPLER CONTROL HARDWARE  
↑ MORE HW FOR COMPUTATION  
↑ POTENTIALLY MORE POWER EFFICIENT (OPS/WATT)  
↓ MORE RESTRICTIVE PROGRAMMING MODEL

### L1-1.11-Techniques To Building Power-efficient Chips Quiz4

QUIZ

WHAT TECHNIQUES ARE COMPUTER DESIGNERS USING TODAY TO BUILD MORE POWER-EFFICIENT CHIPS?

- FEWER, MORE COMPLEX PROCESSORS
- MORE, SIMPLER PROCESSORS
- MAXIMIZING THE SPEED OF THE PROCESSOR CLOCK
- INCREASING THE COMPLEXITY OF THE CONTROL HW

### L1-1.12-Building A Power Efficient Processor

LET'S BUILD A (POWER-EFFICIENT) HIGH PERFORMANCE PROCESSOR!

LATENCY  
 (TIME)  
 (SECONDS)

THROUGHPUT  
 (STUFF/TIME)  
 (JOBS/HOUR)



### L1-1.13-Latency vs Bandwidth Quiz 4

YouTube video player

CAR:

LATENCY  Hours  
 THROUGHPUT  People/Hour

Bus:

LATENCY:  Hours  
 THROUGHPUT  People/Hour



CAR: 2 PEOPLE,  
 200 KM/H  
 Bus: 40 PEOPLE,  
 50 KM/H



### L1-1.14- Core GPU Design Tenets

CORE GPU DESIGN TENETS

- ① LOTS OF SIMPLE COMPUTE UNITS  
TRADE SIMPLE CONTROL FOR MORE COMPUTE
- ② EXPLICITLY PARALLEL PROGRAMMING MODEL
- ③ OPTIMIZE FOR THROUGHPUT NOT LATENCY

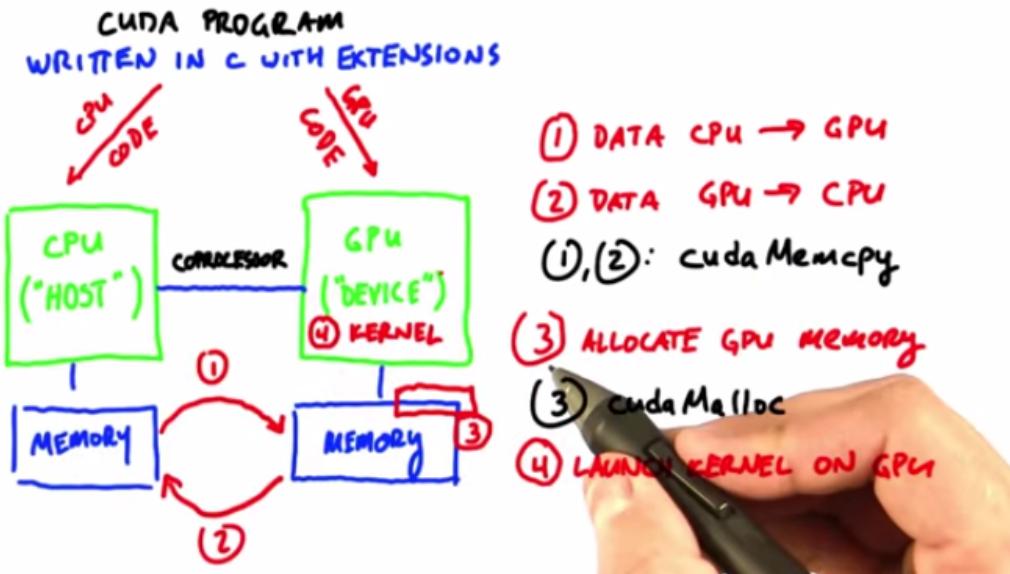


### L1-1.15-GPU from the Point of View of the Developer

GPUs FROM THE POINT OF VIEW OF THE SOFTWARE DEVELOPER  
 — IMPORTANCE OF PROGRAMMING IN PARALLEL

8 CORE NY BRIDGE (INTEL)  
 x 8-WIDE AVX VECTOR OPERATIONS / CORE  
 x 2 THREADS / CORE (HYPERTHREADING)  
128-WAY PARALLELISM

### L1-1.16-CUDA Program Diagram



### L1-1.17-What Can GPU Do in CUDA Quiz 6

QUIZ THE GPU CAN DO THE FOLLOWING (T / F)

- INITIATE DATA SEND GPU → CPU
- RESPOND TO CPU REQUEST TO SEND DATA GPU → CPU
- INITIATE DATA REQUEST CPU → GPU
- RESPOND TO CPU REQUEST TO RECV DATA CPU → GPU
- COMPUTE A KERNEL LAUNCHED BY CPU.
- COMPUTE A KERNEL LAUNCHED BY GPU

## L1-1.18-A CUDA Program

### 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*

## L1-1.19-Defining the GPU Computation Quiz 7

### 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

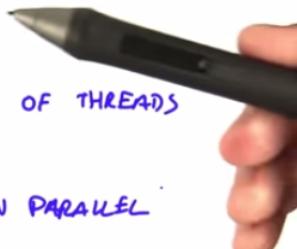
MAKE SURE YOU UNDERSTAND THIS

### WHAT IS THE GPU GOOD AT ?

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

## L1-1.20-What the GPU is Good at

WHAT IS THE GPU GOOD AT?



① EFFICIENTLY LAUNCHING LOTS OF THREADS

② RUNNING LOTS OF THREADS IN PARALLEL

SIMPLE EXAMPLE:

IN: FLOAT ARRAY  $[0 \ 1 \ 2 \ \dots \ 63]$

OUT: FLOAT ARRAY  $[0 \ 1^2 \ 2^2 \ \dots \ 63^2]$   
 $[0 \ 1 \ 4 \ 9 \ \dots \ ]$

KERNEL: SQUARE

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

## L1-1.21-Squaring A Number on the CPU

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

## L1-1.22-Calculation Time on the CPU Quiz 8

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

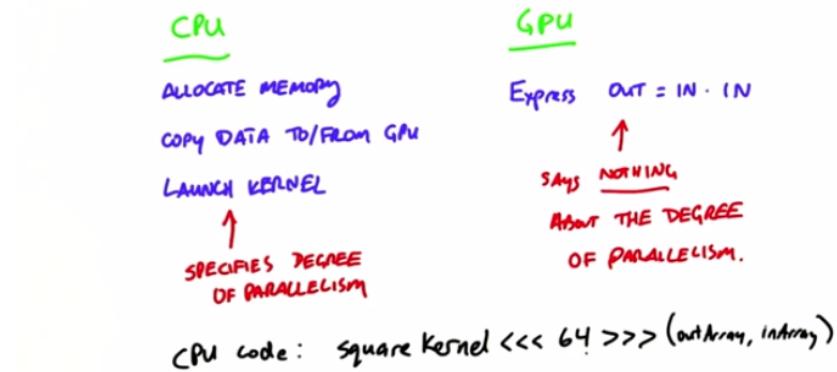
QUIZ:

HOW MANY MULTIPLICATIONS?

I \* TAKES 2 ns.  
HOW LONG TO EXECUTE?

### L1-1.23-GPU Code A High Level View

CPU CODE: A HIGH-LEVEL VIEW



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



### L1-1.24-Calculation Time on The GPU

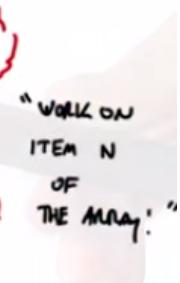
BUT HOW DOES IT WORK IF I LAUNCH 64 INSTANCES OF THE SAME PROGRAM?

QUIZ:

HOW MANY MULTIPLICATIONS?

IF EACH MULT. TAKES

10 ns, HOW LONG FOR THE ENTIRE COMPUTATION?



### L1-1.25-Squaring Numbers Using CUDA Part 1

### L1-1.26-Squaring Numbers Using CUDA Part 2

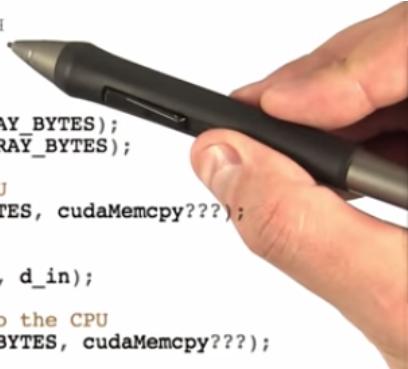
### L1-1.27-Copy to Host or Copy to Device Quiz10

### L1-1.28-Squaring Numbers Using CUDA Part 3

### L1-1.29-Squaring Numbers Using CUDA Part 4

```

1 #include <stdio.h>
2
3 __global__ void square(float * d_out, float * d_in) {
4     int idx = threadIdx.x;
5     float f = d_in[idx];
6     d_out[idx] = f * f;
7 }
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
32    square<<<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, cudaMemcpyDeviceToHost);
36
37    // print out the resulting array
38    for (int i = 0; i < ARRAY_SIZE; i++) {
39        printf("%f", h_out[i]);
40        printf((i % 4) != 3) ? "\t" : "\n";
41    }
42
43    // free GPU memory allocation
44    cudaFree(d_in);
45    cudaFree(d_out);
46
47    return 0;
48 }
```



### L1-1.30-Cubing Numbers Using CUDA Quiz11

### L1-1.31-Configuring the Kernel Launch Parameters Part 1

#### 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

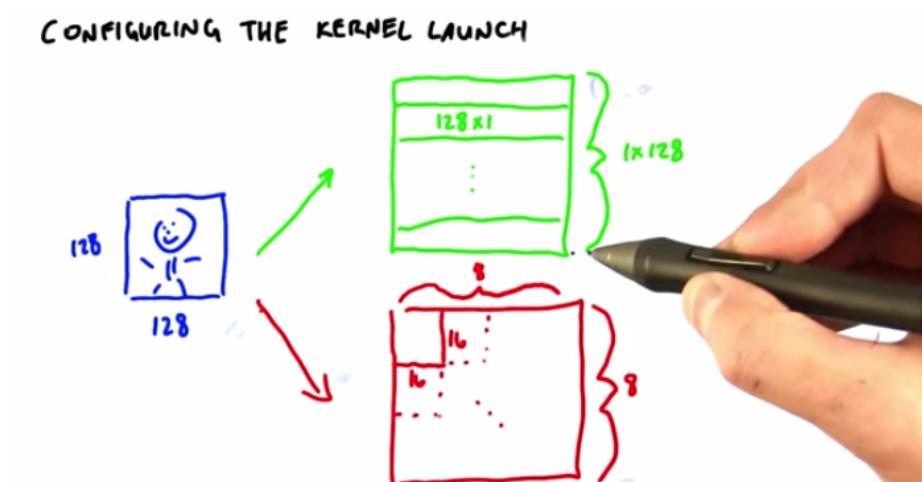
SQUARE <<< 1, 64 >>> (d\_out, d\_in)

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 >>> ( ... )

↓                    ↓

1, 2, or 3D        1, 2, or 3D

dim3(x, y, z)

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

Square <<< 1, 64 >>> == Square <<< dim3(1, 1, 1),  
dim3(64, 1, 1) >>>

## L1-1.32-Configuring the Kernel Launch Parameters Part 2

### CONFIGURING THE KERNEL LAUNCH

KERNEL <<< GRID OF BLOCKS, BLOCK OF THREADS >>> ( ... )  
square <<< dim3(bx,by,bz), dim3(tx,ty,tz), shmem >>> ( ... )

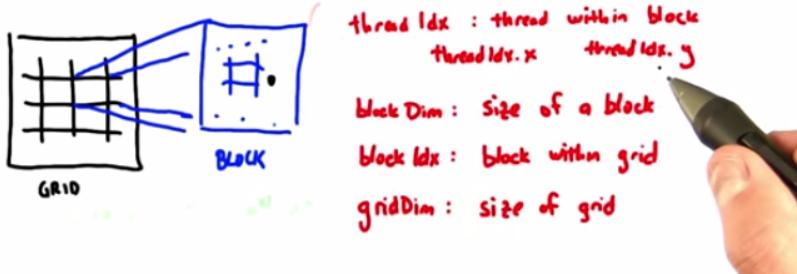
grid of blocks  
bx·by·bz

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 >>> ( ... )



### CONFIGURING THE KERNEL LAUNCH

KERNEL <<< GRID OF BLOCKS, BLOCK OF THREADS >>> ( ... )  
square <<< dim3(bx,by,bz), dim3(tx,ty,tz), shmem >>> ( ... )

Q12 kernel << dim3(8,4,2), dim3(16,16) >> ( ... )

How many blocks?

How many threads/block?

How many total threads?

### L1-1.33-What We Know So Far

#### 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

### L1-1.34-Map

#### 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
- ADD ONE TO EACH ELEMENT IN AN INPUT ARRAY
- SUM UP ALL ELEMENTS IN AN INPUT ARRAY
- COMPUTE THE AVERAGE OF AN INPUT ARRAY

### L1-1.35-Summary of Lesson 1

#### WHAT WE LEARNED)

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



### L1-1.37-Problem Set #1

Problem Set #1



#### How Pixels Are Represented



```
struct uchar4 {
    // Red
    unsigned char r;
    // Green
    unsigned char g;
    // Blue
    unsigned char b;
    // Alpha
    unsigned char a;
}
```

3

#### Converting Color to Black and White

$$I = (R + G + B) / 3$$

$$I = .299f * R + .587f * G + .114f * B$$

