

Administration

Covered topics:

- (GP)GPU computing/parallelization
- C++ CUDA (parallel computing platform)

TAs:

- cs179.ta@gmail.com
- Parker Won (jwon@caltech.edu)
- Nailen Matchstick (nailen@caltech.edu)
- Jordan Bonilla (jbonilla@caltech.edu)

Website:

- <http://courses.cms.caltech.edu/cs179/>

Overseeing Instructor:

- Al Barr (barr@cs.caltech.edu)

Class time:

- ANB 105, MWF 3:00 PM
 - Recitations on Fridays

Course Requirements

Fill out this survey: <https://goo.gl/forms/4GDYz4PtpcBr0qe03>

Homework:

- 6 weekly assignments
- Each worth 10% of grade

Final project:

- 4-week project
- 40% of grade total

P/F Students must receive at least 60% on every assignment AND the final project

Homework

Due on Wednesdays before class (3PM)

Collaboration policy:

- Discuss ideas and strategies freely, but all code must be your own

Office Hours: Located in ANB 104

- Times: TBA (will be announced before first set is out)

Extensions

- Ask a TA for one if you have a valid reason

Projects

Topic of your choice

- We will also provide many options

Teams of up to 2 people

- 2-person teams will be held to higher expectations

Requirements

- Project Proposal
- Progress report(s) and Final Presentation
- More info later...

Machines

Primary machine (multi-GPU):

- Currently being setup. You will gain access shortly after emailing cs179.ta@gmail.com

Secondary machines

- mx.cms.caltech.edu
- minuteman.cms.caltech.edu
- Use your CMS login
- NOTE: Not all assignments work on these machines

Change your password

- Use *passwd* command

Machines

Alternative: Use your own machine:

- Must have an NVIDIA CUDA-capable GPU
- Virtual machines won't work
 - Exception: Machines with I/O MMU virtualization and certain GPUs
- Special requirements for:
 - Hybrid/optimus systems
 - Mac/OS X

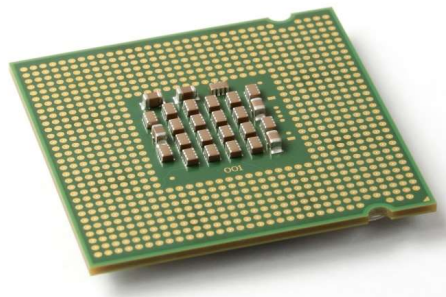
Setup guide on the website is outdated. Do not follow 2016 instructions

The CPU

The “Central Processing Unit”

Traditionally, applications use CPU for primary calculations

- General-purpose capabilities
- Established technology
- Usually equipped with 8 or less powerful cores
- Optimal for concurrent processes but not large scale parallel computations



Wikimedia commons: Intel_CPU_Pentium_4_640_Prescott_bottom.jpg

The GPU

The "Graphics Processing Unit"

Relatively new technology designed for parallelizable problems

- Initially created specifically for graphics
- Became more capable of general computations



GPUs – The Motivation

Raytracing:

for all pixels (i,j) :

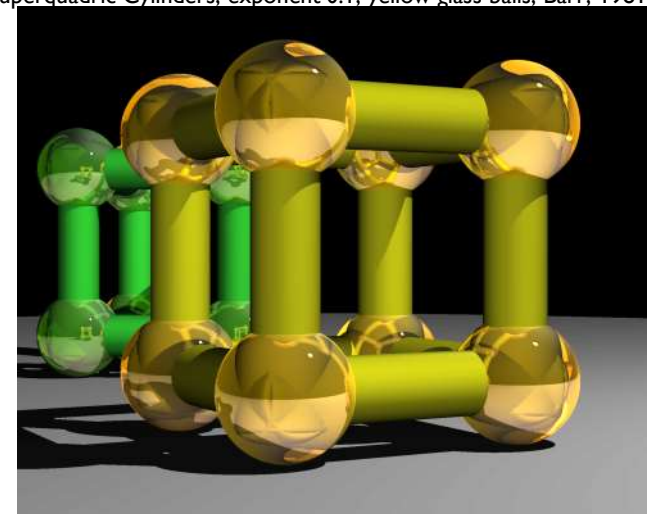
 calculate ray point and direction in 3d space

 if ray intersects object:

 calculate lighting at closest object

 store color of (i,j)

Superquadric Cylinders, exponent 0.1, yellow glass balls, Barr, 1981



EXAMPLE

Add two arrays

- $A[] + B[] \rightarrow C[]$

On the CPU:

```
float *C = malloc(N * sizeof(float));  
for (int i = 0; i < N; i++)  
C[i] = A[i] + B[i];  
return C;
```

This operates sequentially... can we do better?

A simple problem...

- On the CPU (multi-threaded, pseudocode):

```
(allocate memory for C)
Create # of threads equal to number of cores on processor
(around 2, 4, perhaps 8)
(Indicate portions of A, B, C to each thread...)
```

...

```
In each thread,
For (i from beginning region of thread)
C[i] <- A[i] + B[i]
//lots of waiting involved for memory reads, writes, ...
wait for threads to synchronize...
```

This is slightly faster – 2-8x (slightly more with other tricks)

A simple problem...

- How many threads? How does performance scale?
- Context switching:
 - High penalty on the CPU
 - Not an issue on the GPU

A simple problem...

- On the GPU:

(allocate memory for A, B, C on GPU)

Create the “kernel” – each thread will perform one (or a few) additions

Specify the following kernel operation:

For all i 's (indices) assigned to this thread:

$C[i] \leftarrow A[i] + B[i]$

start ~20000 (!) threads

wait for threads to synchronize...

GPU: Strengths Revealed

- Emphasis on parallelism means we have lots of cores
- This allows us to run many threads simultaneously with no context switches



GPU Computing: Step by Step

- Setup inputs on the host (CPU-accessible memory)
- Allocate memory for outputs on the host
- Allocate memory for inputs on the GPU
- Allocate memory for outputs on the GPU
- Copy inputs from host to GPU
- Start GPU kernel
- Copy output from GPU to host

NOTE: Copying can be asynchronous

The Kernel

- Our “parallel” function
- Given to each thread
- Simple implementation:

```
__global__ void  
cudaAddVectorsKernel(float * a, float * b, float * c) {  
    //Decide an index somehow  
    c[index] = a[index] + b[index];  
}
```

Indexing

```
__global__ void  
cudaAddVectorsKernel(float * a, float * b, float * c) {  
    unsigned int index = blockIdx.x * blockDim.x + threadIdx.x;  
    c[index] = a[index] + b[index];  
}
```

Calling the Kernel

```
void cudaAddVectors(const float* a, const float* b, float* c, size){
    //For now, suppose a and b were created before calling this function

    // dev_a, dev_b (for inputs) and dev_c (for outputs) will be
    // arrays on the GPU.

    float * dev_a;
    float * dev_b;

    float * dev_c;

    // Allocate memory on the GPU for our inputs:
    cudaMalloc((void **) &dev_a, size*sizeof(float));
    cudaMemcpy(dev_a, a, size*sizeof(float), cudaMemcpyHostToDevice);

    cudaMalloc((void **) &dev_b, size*sizeof(float)); // and dev_b
    cudaMemcpy(dev_b, b, size*sizeof(float), cudaMemcpyHostToDevice);

    // Allocate memory on the GPU for our outputs:
    cudaMalloc((void **) &dev_c, size*sizeof(float));
```

Calling the Kernel (2)

```
//At lowest, should be 32
//Limit of 512 (Tesla), 1024 (newer)
const unsigned int threadsPerBlock = 512;

//How many blocks we'll end up needing
const unsigned int blocks = ceil(size/float(threadsPerBlock));

//Call the kernel!
cudaAddVectorsKernel<<<blocks, threadsPerBlock>>>
    (dev_a, dev_b, dev_c);

//Copy output from device to host (assume here that host memory
//for the output has been calculated)

cudaMemcpy(c, dev_c, size*sizeof(float), cudaMemcpyDeviceToHost);

//Free GPU memory
cudaFree(dev_a);
cudaFree(dev_b);
cudaFree(dev_c);
}
```

Questions?

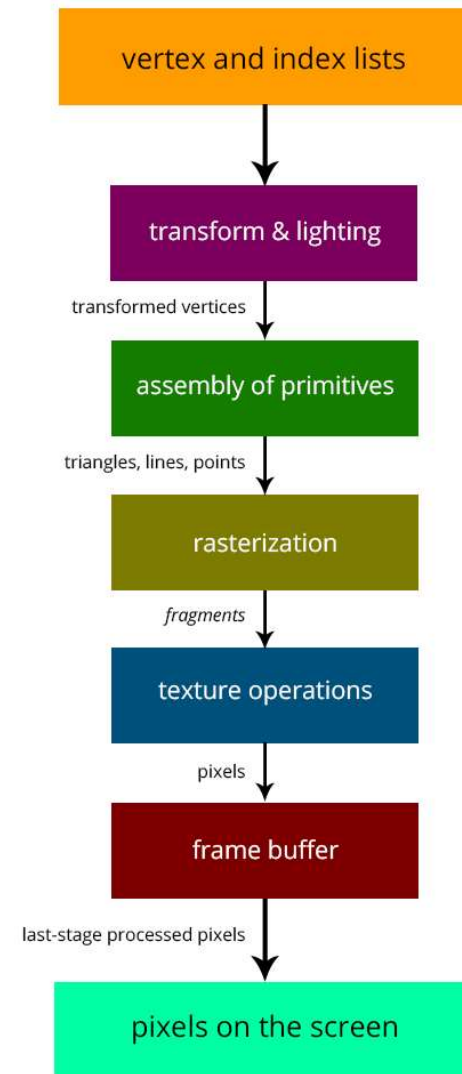
GPUs – Brief History

- Initially based on graphics focused fixed-function pipelines
 - Pre-set functions, limited options



<http://gamedevelopment.tutsplus.com/articles/the-end-of-fixed-function-rendering-pipelines-and-how-to-move-on--cms-21469>

Source: Super Mario 64, by Nintendo



GPUs – Brief History

- Shaders
 - Could implement one's own functions!
 - GLSL (C-like language)
 - Could “sneak in” general-purpose programming!



<http://minecraftsix.com/gsl-shaders-mod/>

GPUs – Brief History

- “General-purpose computing on GPUs” (GPGPU)
 - Hardware has gotten good enough to a point where it’s basically having a mini-supercomputer
- CUDA (Compute Unified Device Architecture)
 - General-purpose parallel computing platform for NVIDIA GPUs
- OpenCL (Open Computing Language)
 - General heterogenous computing framework
- Both are accessible as extensions to various languages
 - If you’re into python, checkout theano