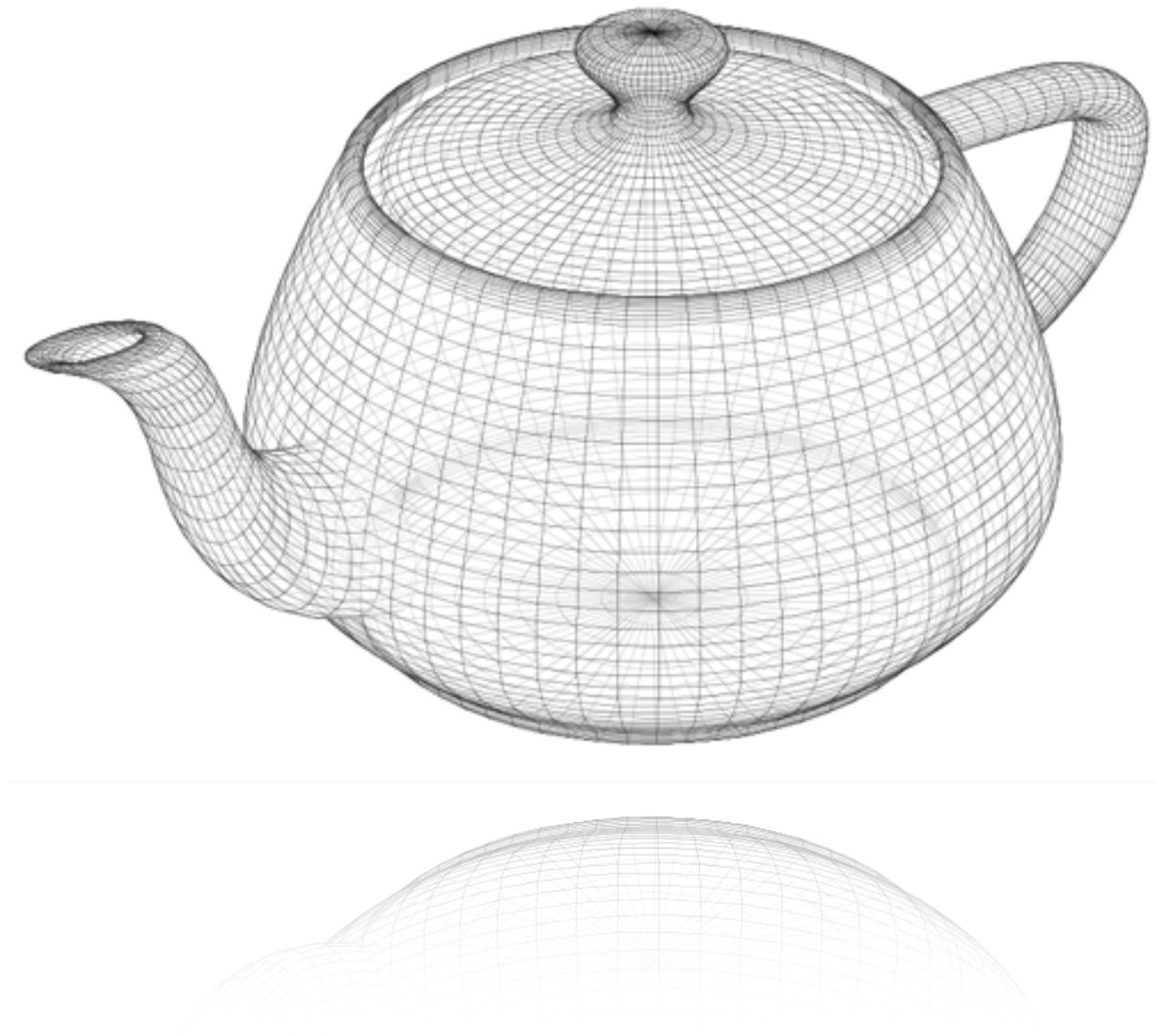
Introduction to GPU Computing

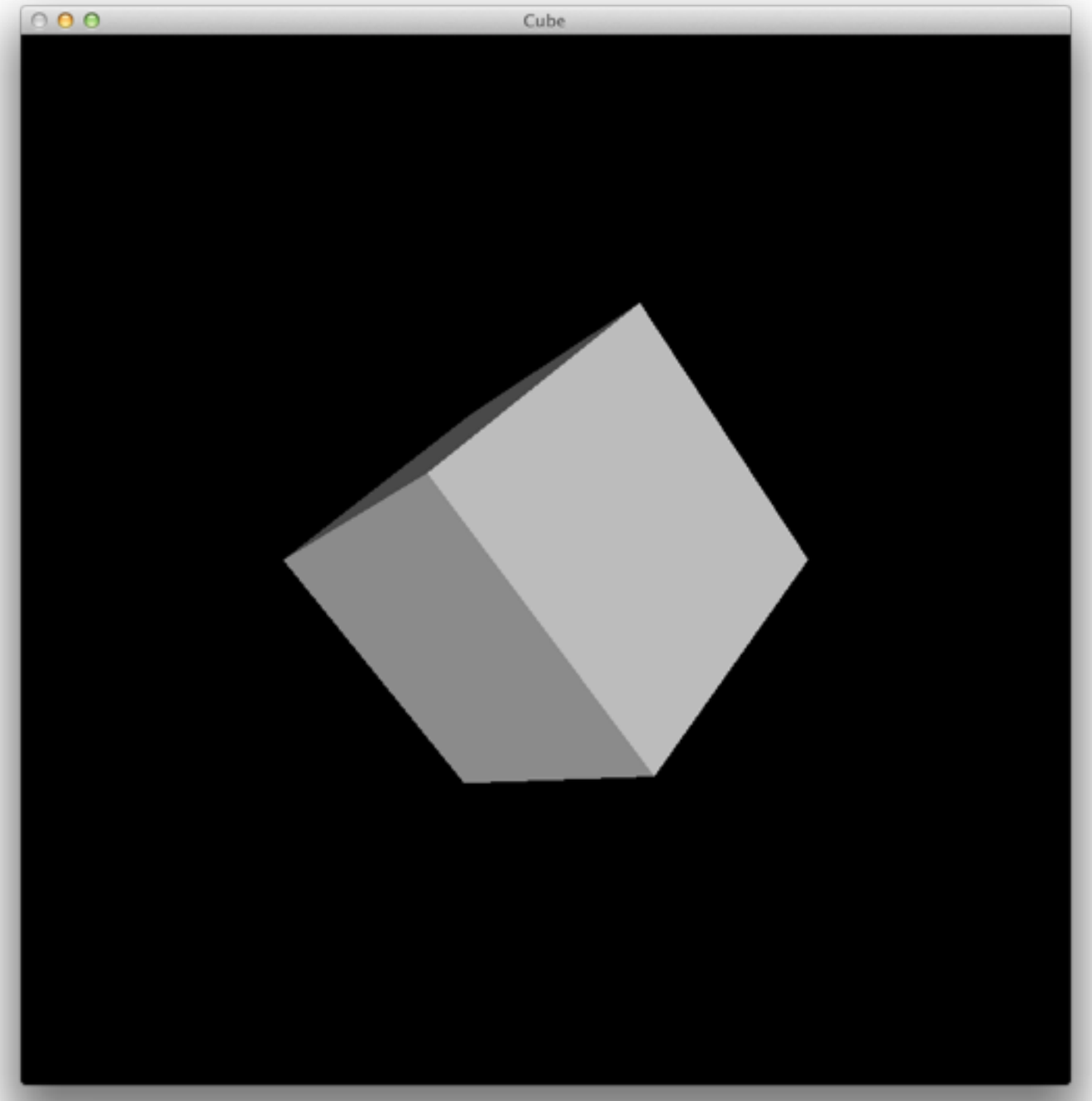
Ilya Kuzovkin

13 May 2014, Tartu



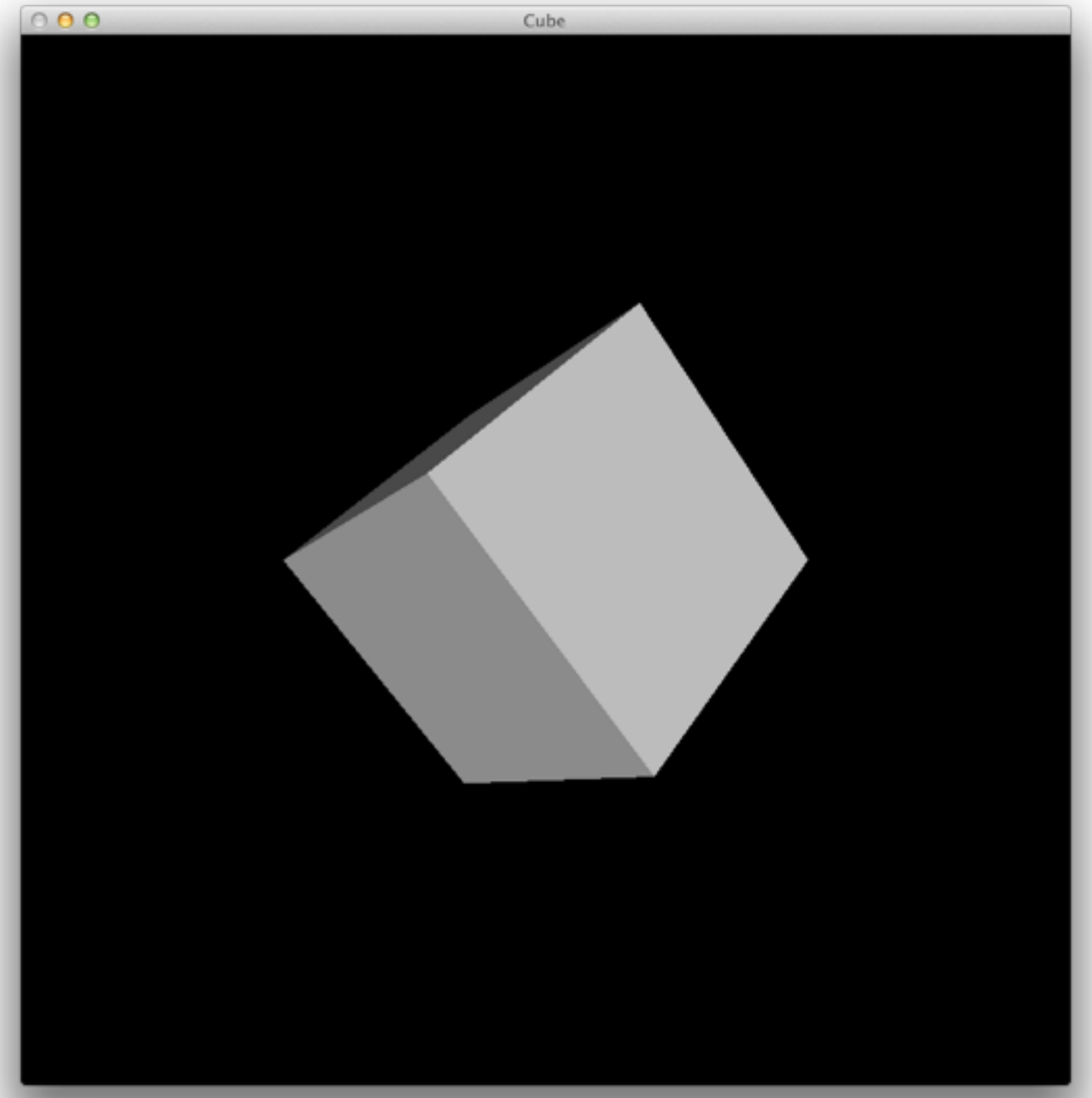
Part I
“Teapot”

Simple OpenGL Program



Idea of **computing on GPU** emerged because GPUs became very **good** at **parallel** computations.

Simple OpenGL Program



Idea of **computing on GPU** emerged because GPUs became very **good** at **parallel** computations.

Let us start from observing an **example** of **parallelism** in a simple **OpenGL** application.

Simple OpenGL Program

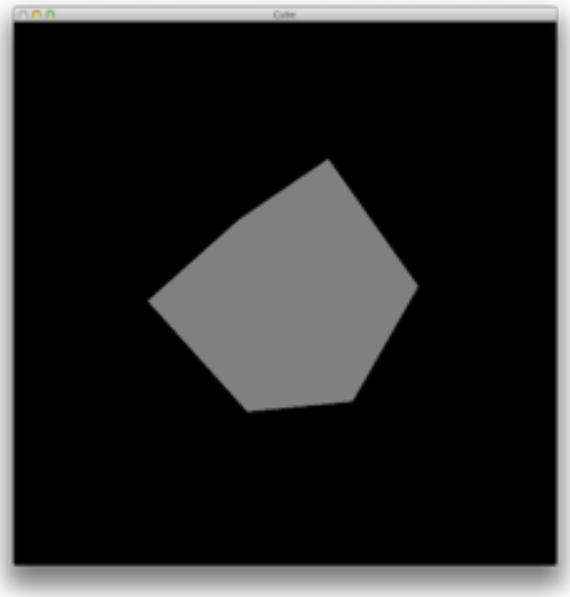
You will need CodeBlocks[Windows, Linux] or XCode[Mac] to run this example.

- Install CodeBlocks bundled with MinGW compiler from http://www.codeblocks.org/downloads/26

- Download codebase from https://github.com/kuz/Introduction-to-GPU-Computing

- Open the project from the **`code/Cube`**

- Compile & run it



Shader Program

Program which is **executed on GPU**.
Has to be written using **shading language**.
In OpenGL this language is **GLSL**, which is based on C.

http://www.opengl.org/wiki/Shader

Shader Program

Program which is **executed on GPU**.
Has to be written using **shading language**.
In OpenGL this language is **GLSL**, which is based on C.

OpenGL has 5 main shader stages:

- Vertex Shader
- Tessellation Control
- Geometry Shader
- Fragment Shader
- Compute Shader (since 4.3)

http://www.opengl.org/wiki/Shader

Shader Program

Program which is **executed on GPU**.
Has to be written using **shading language**.
In OpenGL this language is **GLSL**, which is based on C.

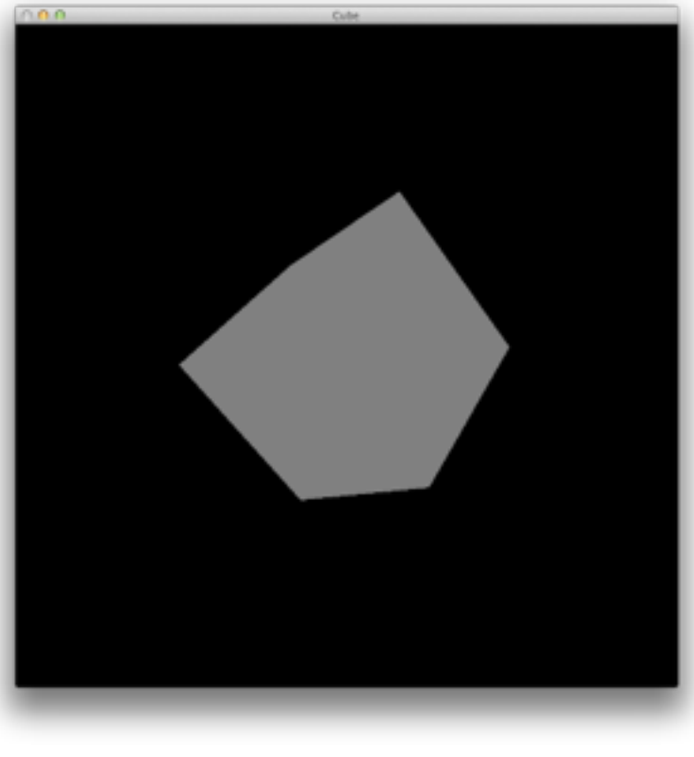
OpenGL has 5 main shader stages:

- Vertex Shader
- Tessellation Control
- Geometry Shader
- Fragment Shader
- Compute Shader (since 4.3)

http://www.opengl.org/wiki/Shader

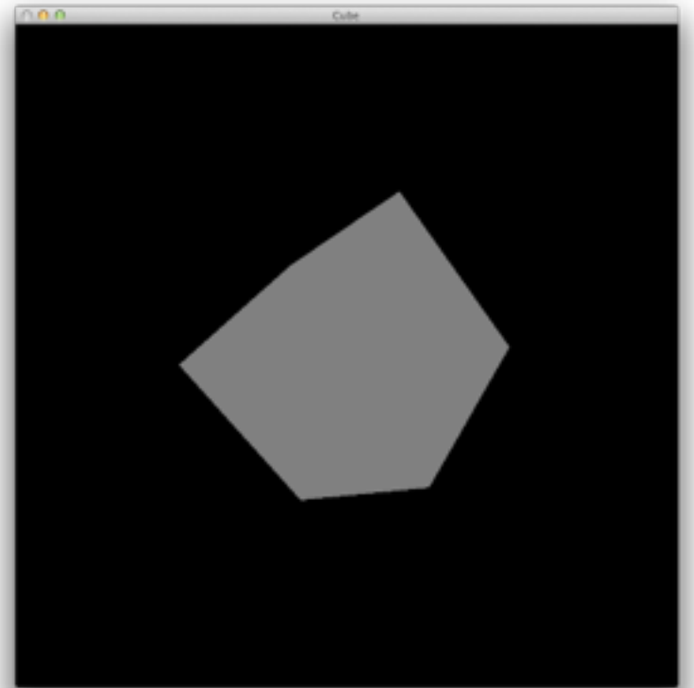
LIGHTING

Is it a cube or not?
We will find out as soon as
we add **lighting** to the scene.



LIGHTING

Is it a cube or not?
We will find out as soon as
we add **lighting** to the scene.

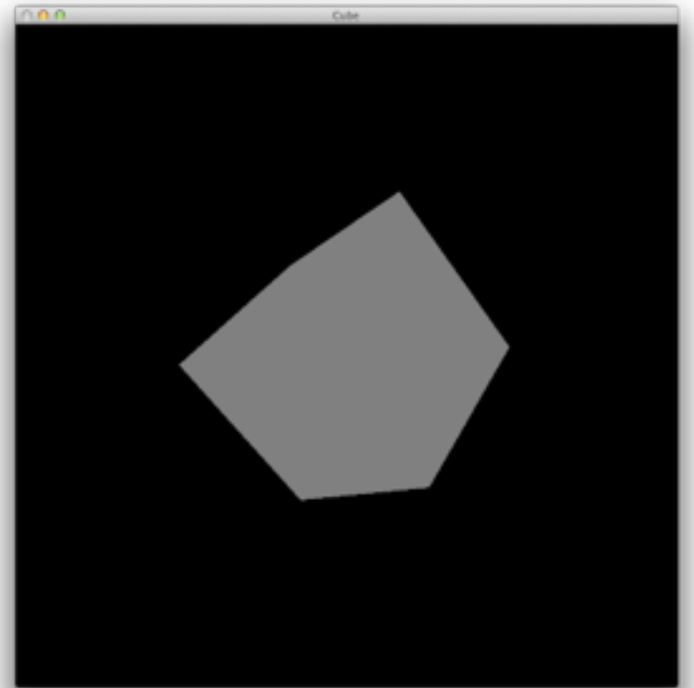


$$I_A \cdot A + I_D \cdot D \cdot \boldsymbol{l}^T \boldsymbol{n} + I_S \cdot S \cdot \left(\boldsymbol{r}^T \boldsymbol{v}\right)^S$$

https://github.com/konstantint/ComputerGraphics2013/blob/master/Lectures/07%20-%20Color%20and%20Lighting/slides07_colorandlighting.pdf

LIGHTING

Is it a cube or not?
We will find out as soon as
we add **lighting** to the scene.



$$I_A \cdot A + I_D \cdot D \cdot \boldsymbol{l}^T \boldsymbol{n} + I_S \cdot S \cdot \left(\boldsymbol{r}^T \boldsymbol{v}\right)^S$$

Exercise: code that equation into
fragment shader of the Cube program

https://github.com/konstantint/ComputerGraphics2013/blob/master/Lectures/07%20-%20Color%20and%20Lighting/slides07_colorandlighting.pdf

LIGHTING

```
// Configuration
vec3 color = vec3(0, 0, 0);
vec3 light = vec3(0, 0, 1);
vec3 viewer = vec3(0, 0, 1);
vec3 normal = vertexNormal;

// Ambient
vec3 l_ambient = vec3(0.2, 0.2, 0.2);
vec3 m_ambient = vec3(0.5, 0.5, 0.5);
color += l_ambient * m_ambient;

// Diffuse
vec3 l_diffuse = vec3(0.7, 0.7, 0.7);
vec3 m_diffuse = vec3(0.5, 0.5, 0.5);
color += l_diffuse * m_diffuse * dot(light, normal);

// Specular
vec3 l_specular = vec3(1.0, 1.0, 1.0);
vec3 m_specular = vec3(0.5, 0.5, 0.5);
float sh = 20.0;
vec3 reflection = reflect(-light, normal);
color += l_specular * m_specular * pow(clamp(dot(reflection, viewer), 0, 1), sh);

// Return final color
fragColor = vec4(color, 0);
```

COMPARE FPS

- Run the program with **lighting enabled** and look at **FPS** values

COMPARE FPS

- Run the program with **lighting enabled** and look at **FPS** values
- In **`cube.cpp idle()`** function uncomment **dummy code** which simulates approximately **same** amount of **computations** as Phong lighting model requires.

COMPARE FPS

- Run the program with **lighting enabled** and look at **FPS** values

- In **`cube.cpp idle()`** function uncomment **dummy code** which simulates approximately **same** amount of **computations** as Phong lighting model requires.

- Note that these computations are performed on **CPU**

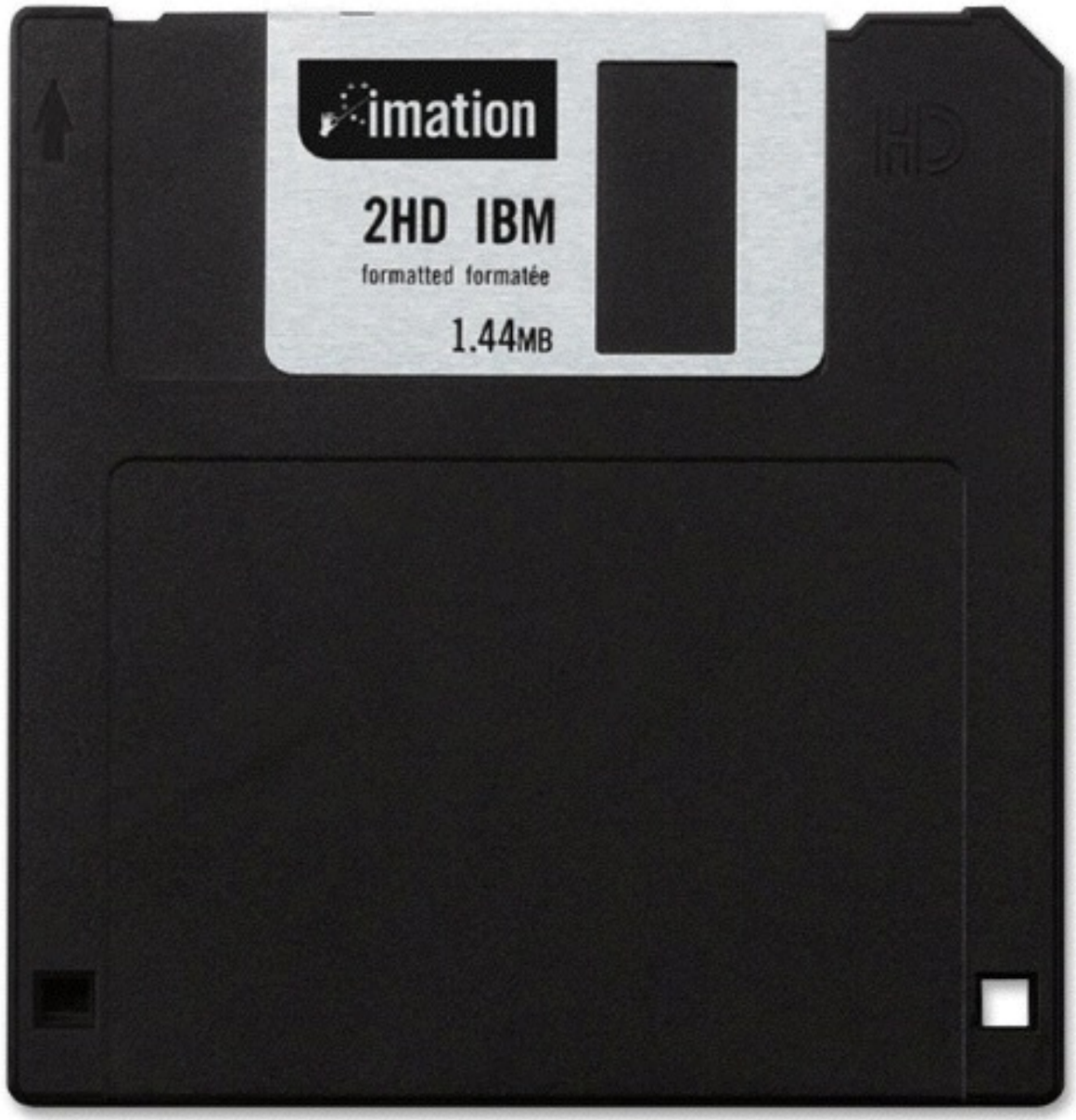
COMPARE FPS

- Run the program with **lighting enabled** and look at **FPS** values
- In **`cube.cpp idle()`** function uncomment **dummy code** which simulates approximately **same** amount of **computations** as Phong lighting model requires.
- Note that these computations are performed on **CPU**
- Observe how FPS has **changed**

COMPARE FPS

- Run the program with **lighting enabled** and look at **FPS** values
- Note that these computations are performed on **CPU**
- Observe how FPS has **changed**

Parallel computations are fast on GPU.
Lets use it to compute something useful.



Part II
“Old School”

OpenGL pipeline + GLSL

Take the **input data** from the CPU memory and put it as an **image** into the GPU memory

http://www.opengl.org/wiki/Framebuffer

OpenGL pipeline + GLSL

Take the **input data** from the CPU memory and put it as an **image** into the GPU memory

In the fragment shader perform a computation on each of the pixels of that **image**

http://www.opengl.org/wiki/Framebuffer

OpenGL pipeline + GLSL

Take the **input data** from the CPU memory and put it as an **image** into the GPU memory

In the fragment shader perform a computation on each of the pixels of that **image**

Store the resulting image to the **Render Buffer** inside the GPU memory

http://www.opengl.org/wiki/Framebuffer

OpenGL pipeline + GLSL

Take the **input data** from the CPU memory and put it as an **image** into the GPU memory

In the fragment shader perform a computation on each of the pixels of that **image**

Store the resulting image to the **Render Buffer** inside the GPU memory

Read **output** from the GPU memory back to the CPU memory

http://www.opengl.org/wiki/Framebuffer

OpenGL pipeline + GLSL

- Create **texture** where will store the **input data**

```
glGenTextures(1, &fbo->texid);
glBindTexture(GL_TEXTURE_2D, fbo->texid);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA32F_ARB,
             width, height, 0, GL_RGBA, GL_FLOAT, data);
```

http://www.opengl.org/wiki/Framebuffer

OpenGL pipeline + GLSL

- Create **texture** where will store the **input data**

```
glGenTextures(1, &fbo->texid);
glBindTexture(GL_TEXTURE_2D, fbo->texid);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA32F_ARB,
             width, height, 0, GL_RGBA, GL_FLOAT, data);
```

- Create **FrameBuffer Object** (FBO) to “render” to

```
glGenFramebuffers(1, &fbo->fb);          // frame buffer id
glGenRenderbuffers(1, &fbo->rb);         // render buffer id
glBindFramebuffer(GL_FRAMEBUFFER, fbo->fb);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
                       GL_TEXTURE_2D, fbo->texid, 0);
```

http://www.opengl.org/wiki/Framebuffer

OPENGL PIPELINE + GLSL

- Run OpenGL pipeline

http://www.opengl.org/wiki/Framebuffer

OPENGL PIPELINE + GLSL

- Run **OpenGL pipeline**
  - Render GL_QUADS of **same size** as the **texture** matrix

http://www.opengl.org/wiki/Framebuffer

OPENGL PIPELINE + GLSL

- Run **OpenGL pipeline**
  - Render GL_QUADS of **same size** as the **texture** matrix
  - Use **fragment shader** to perform per-fragment computations using data from the **texture**

http://www.opengl.org/wiki/Framebuffer

OPENGL PIPELINE + GLSL

- Run **OpenGL pipeline**
  - Render GL_QUADS of **same size** as the **texture** matrix
  - Use **fragment shader** to perform per-fragment computations using data from the **texture**
  - OpenGL will **store result** in the texture given to the **Render Buffer** (within Framebuffer Object)

http://www.opengl.org/wiki/Framebuffer

OpenGL pipeline + GLSL

- Run **OpenGL pipeline**
  - Render GL_QUADS of **same size** as the **texture** matrix
  - Use **fragment shader** to perform per-fragment computations using data from the **texture**
  - OpenGL will **store result** in the texture given to the **Render Buffer** (within Framebuffer Object)

- **Read** the data from the Render Buffer

```
glReadBuffer(GL_COLOR_ATTACHMENT0);
glReadPixels(0, 0, texSize, texSize,
             GL_RGBA, GL_FLOAT, result);
```

http://www.opengl.org/wiki/Framebuffer

OpenGL pipeline + GLSL

- Run **OpenGL pipeline**
  - Render GL_QUADS of **same size** as the **texture** matrix
  - Use **fragment shader** to perform per-fragment computations using data from the **texture**
  - OpenGL will **store result** in the texture given to the **Render Buffer** (within Framebuffer Object)

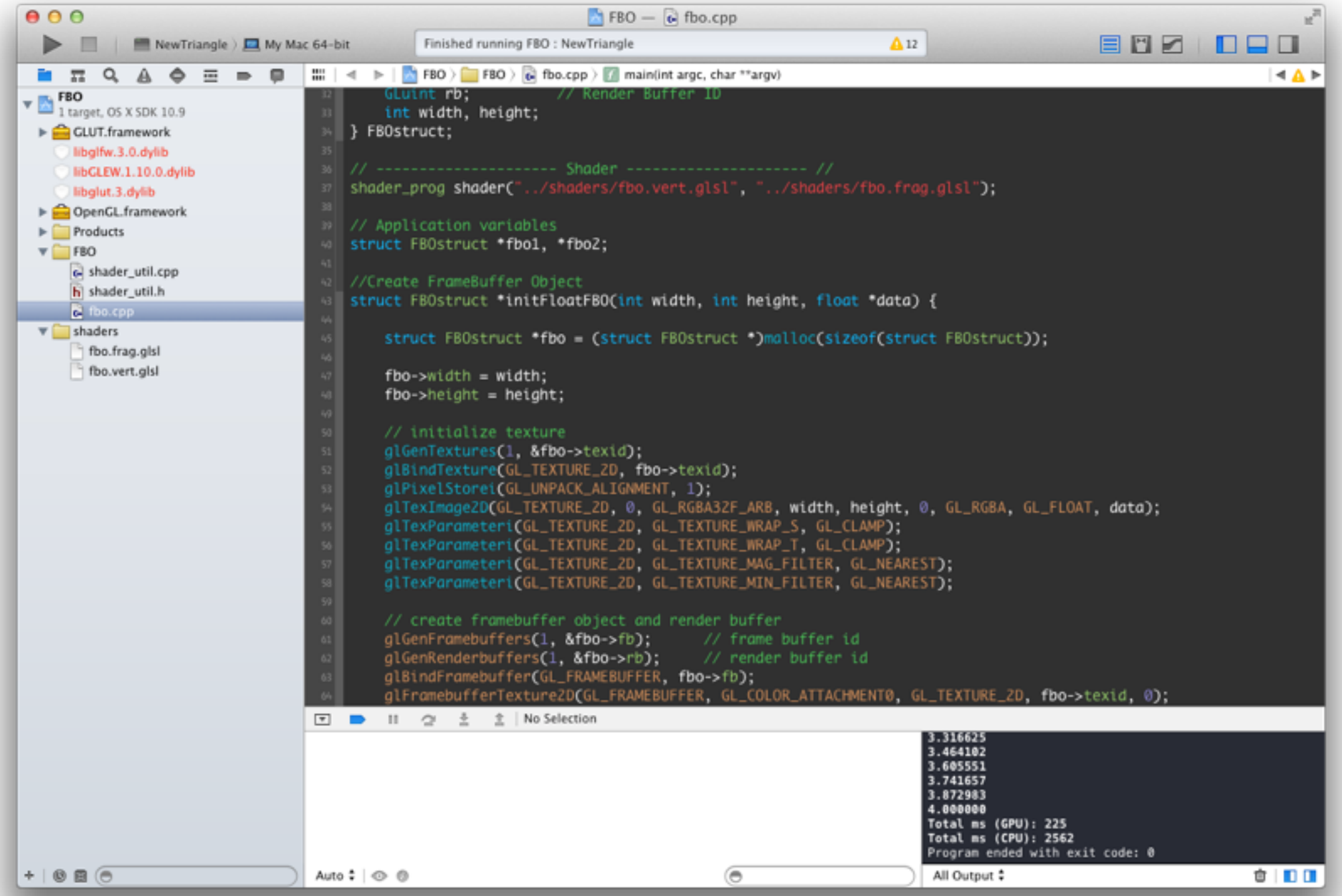
- **Read** the data from the Render Buffer

```
glReadBuffer(GL_COLOR_ATTACHMENT0);
glReadPixels(0, 0, texSize, texSize,
             GL_RGBA, GL_FLOAT, result);
```

- Can we use that to properly **debug** GLSL?

http://www.opengl.org/wiki/Framebuffer

DEMO



Run the project from the **`code/FBO`**

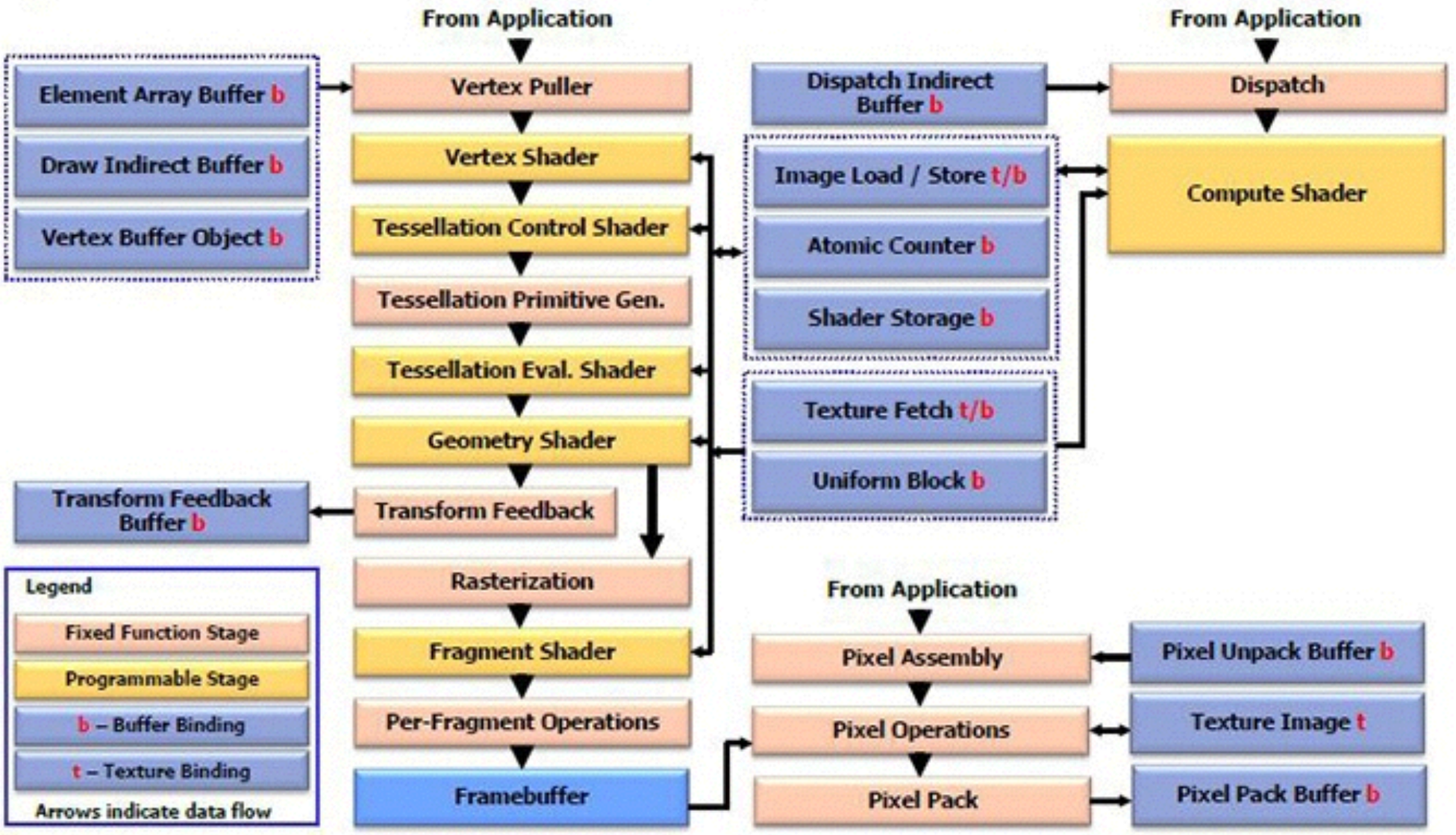
Part III
"Modern Times"

Compute Shader

- Since OpenGL 4.3
- Used to compute things not related to rendering directly

COMPUTE SHADER

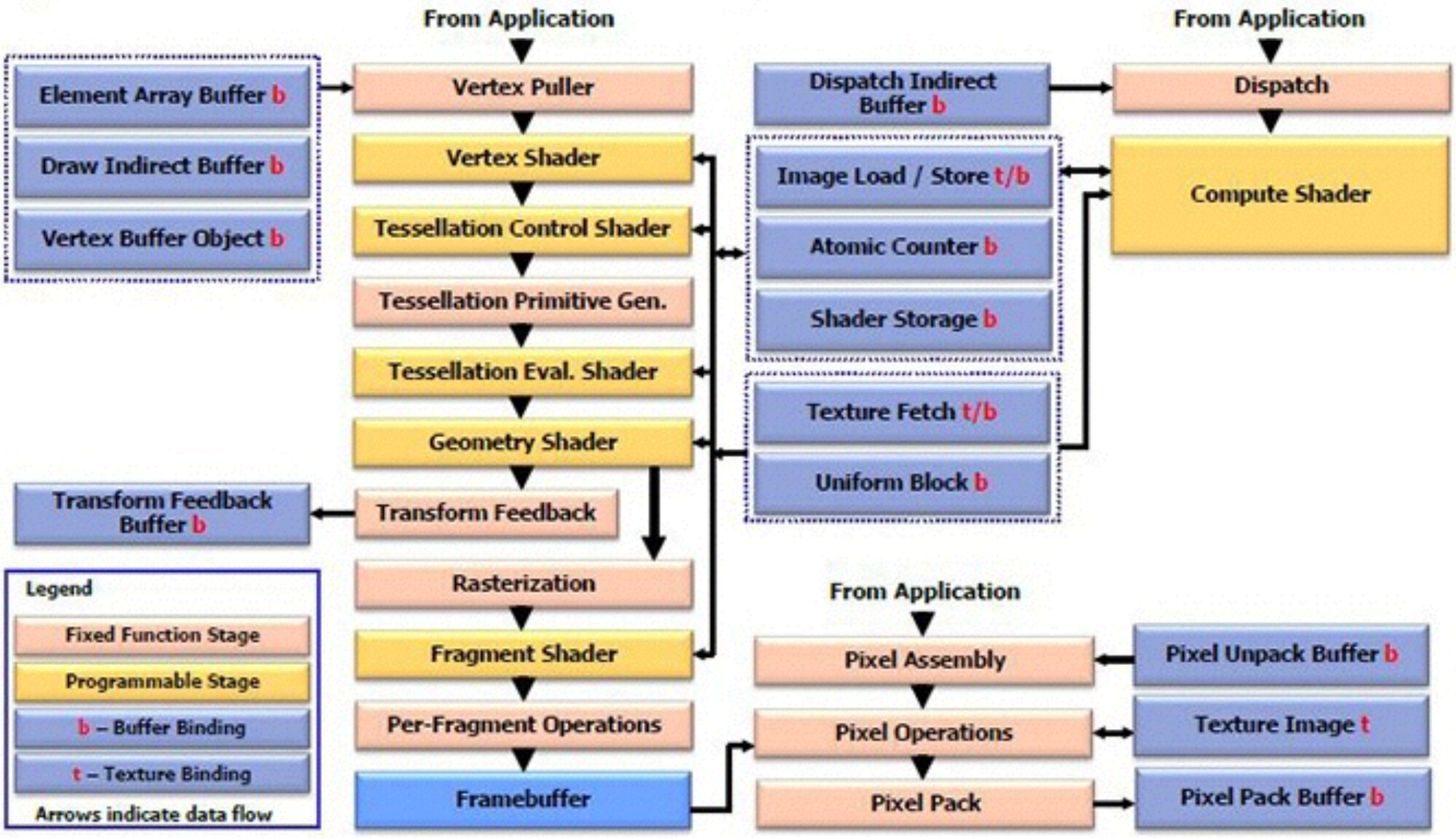
- Since OpenGL 4.3
- Used to compute things not related to rendering directly



COMPUTE SHADER

Will not talk about it

- Since OpenGL 4.3
- Used to compute things not related to rendering directly



http://web.engr.oregonstate.edu/~mjb/cs557/Handouts/compute.shader.1pp.pdf

OpenCL

http://wiki.tiker.net/CudaVsOpenCL

Supported only by nVidia hardware

https://developer.nvidia.com/cuda-gpus

Supported by nVidia, AMD, Intel, Qualcomm

http://www.khronos.org/conformance/adopters/conformant-products#opencl

http://wiki.tiker.net/CudaVsOpenCL

Supported only by nVidia hardware

https://developer.nvidia.com/cuda-gpus

Supported by nVidia, AMD, Intel, Qualcomm

http://www.khronos.org/conformance/adopters/conformant-products#opencl

Implementations only by nVidia

OpenCL

http://wiki.tiker.net/CudaVsOpenCL

Supported only by nVidia hardware

https://developer.nvidia.com/cuda-gpus

Supported by nVidia, AMD, Intel, Qualcomm

http://www.khronos.org/conformance/adopters/conformant-products#opencl

Implementations only by nVidia

OpenCL

~same performance levels

http://wiki.tiker.net/CudaVsOpenCL

Supported only by nVidia hardware

https://developer.nvidia.com/cuda-gpus

Supported by nVidia, AMD, Intel, Qualcomm

http://www.khronos.org/conformance/adopters/conformant-products#opencl

Implementations only by nVidia

OpenCL

~same performance levels

Developer-friendly

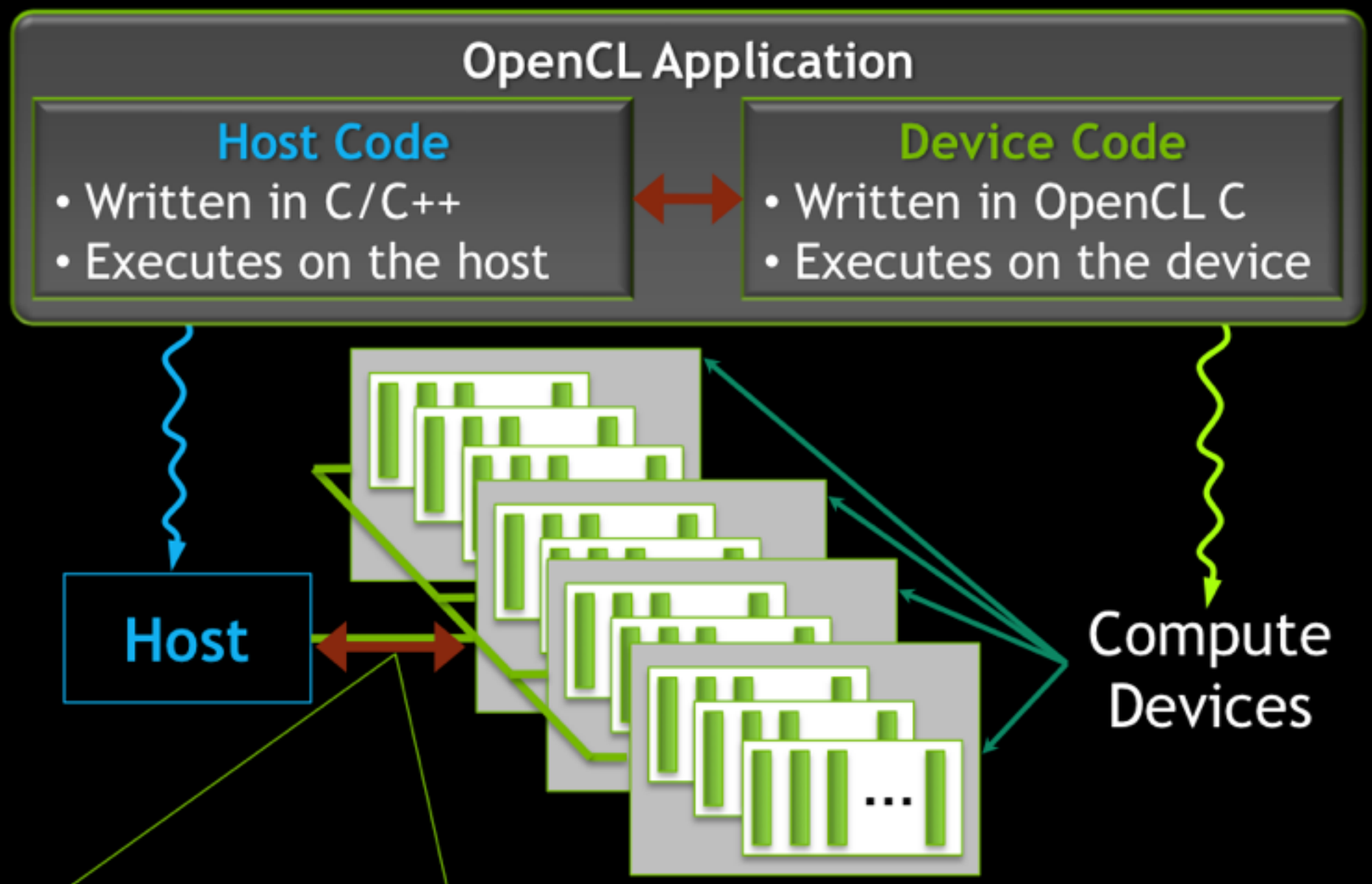
OpenCL

http://wiki.tiker.net/CudaVsOpenCL

OpenCL

PART III
CHAPTER 1

Anatomy of an OpenCL Application



Host code sends commands to the Devices:

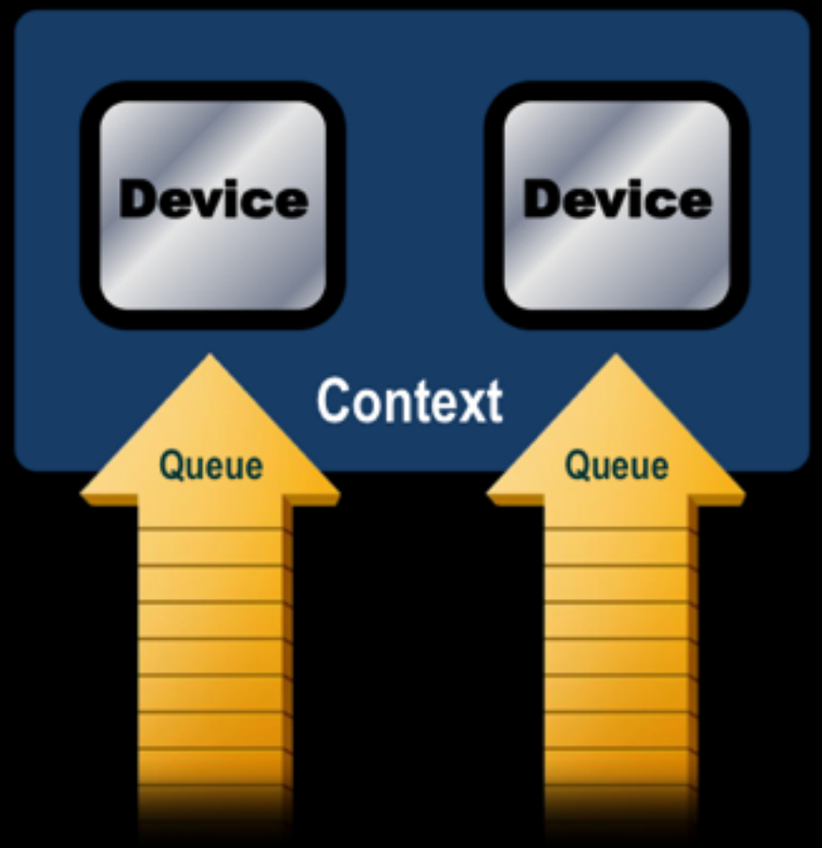
... to transfer data between host memory and device memories

... to execute device code

OpenCL Execution Model

The application runs on a Host which submits work to the Devices

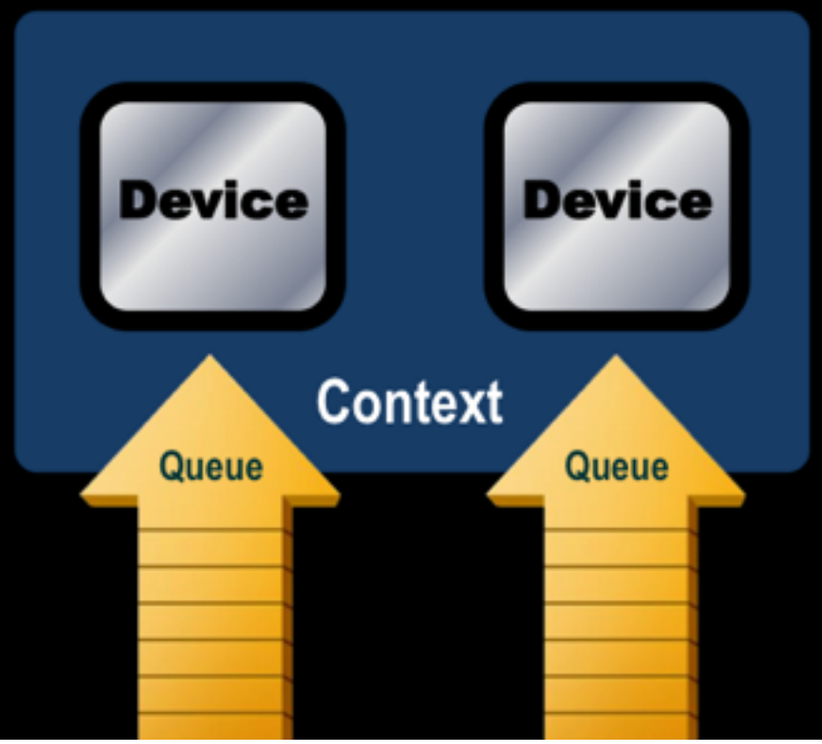
- **Context**: The environment within which work-items execute; includes devices and their memories and command queues
- **Command Queue**: A queue used by the Host application to submit work to a Device (e.g., kernel execution instances)
  - Work is queued in-order, one queue per device
  - Work can be executed in-order *or* out-of-order



OpenCL Execution Model

The application runs on a Host which submits work to the Devices

- **Context**: The environment within which work-items execute; includes devices and their memories and command queues
- **Command Queue**: A queue used by the Host application to submit work to a Device (e.g., kernel execution instances)
  - Work is queued in-order, one queue per device
  - Work can be executed in-order *or* out-of-order

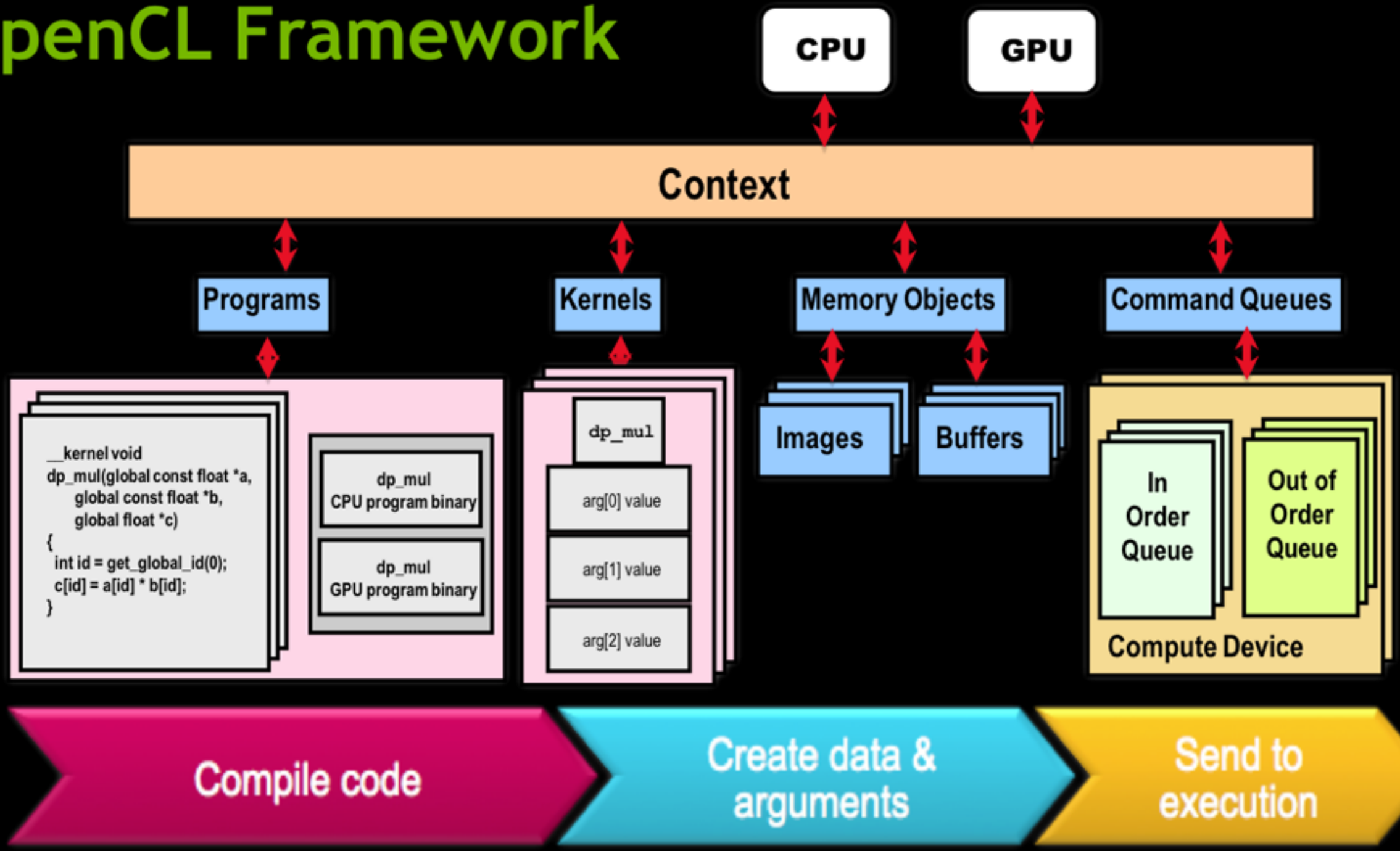



```
// Connect to a compute device
int gpu = 1;
err = clGetDeviceIDs(NULL, gpu ? CL_DEVICE_TYPE_GPU : CL_DEVICE_TYPE_CPU,
                     1, &device_id, NULL);

// Create a compute context
context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);

// Create a command queue
commands = clCreateCommandQueue(context, device_id, 0, &err);
```

OpenCL Framework



OpenCL Framework: Platform Layer

- Query platform information
  - clGetPlatformInfo(): profile, version, vendor, extensions
  - clGetDeviceIDs(): list of devices
  - clGetDeviceInfo(): type, capabilities

- Create an OpenCL context for one or more devices

Context = cl_context

- One or more devices
  - cl_device_id
- Memory and device code shared by these devices
  - cl_mem
  - cl_program
- Command queues to send commands to these devices
  - cl_command_queue

KERNEL

Decompose task into *work-items*

- Define N-dimensional computation domain
- Execute a *kernel* at each point in computation domain

Traditional loop as a function in C

```
void
trad_mul(int n,
         const float *a,
         const float *b,
         float *c)
{
  int i;
  for (i=0; i<n; i++)
    c[i] = a[i] * b[i];
}
```

OpenCL C kernel

```
__kernel void
dp_mul(__global const float *a,
       __global const float *b,
       __global float *c)

{
  int id = get_global_id(0);

  c[id] = a[id] * b[id];
} // execute over n "work items"
```

- A *kernel* is a function executed for each work-item

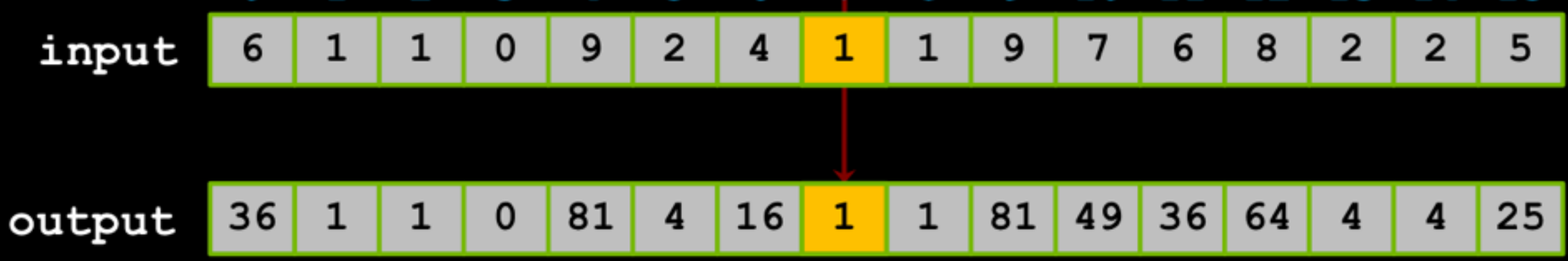
```
__kernel void square(__global float* input, __global float* output)
{
  int i = get_global_id(0);
  output[i] = input[i] * input[i];
}
```

Function qualifier

Built-in function

Address space qualifier

get_global_id(0) = 7



- A *kernel* is a function executed for each work-item

```
__kernel void square(__global float* input, __global float* output)
{
  int i = get_global_id(0);
  output[i] = input[i] * input[i];
}
```

Function qualifier

Built-in function

Address space qualifier

get_global_id(0) = 7

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
input	6	1	1	0	9	2	4	1	1	9	7	6	8	2	2	5
output	36	1	1	0	81	4	16	1	1	81	49	36	64	4	4	25

```
// Create the compute program from the source buffer
program = clCreateProgramWithSource(context, 1, (const char **) &
KernelSource, NULL, &err);

// Build the program executable
err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);

// Create the compute kernel in the program we wish to run
kernel = clCreateKernel(program, "sqrt", &err);
```

WRITE AND READ DATA ON GPU

```
// Create the input and output arrays in device memory
for our calculation
input = clCreateBuffer(context,  CL_MEM_READ_ONLY,
                       sizeof(float) * data_size,
                       NULL, NULL);
output = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
                        sizeof(float) * data_size,
                        NULL, NULL);
```

```
// Write our data set into the input array in device
memory
 err = clEnqueueWriteBuffer(commands, input, CL_TRUE, 0,
sizeof(float) * data_size, data, 0, NULL, NULL);
```

Write and Read Data on GPU

```
// Create the input and output arrays in device memory
for our calculation
input = clCreateBuffer(context,  CL_MEM_READ_ONLY,
                       sizeof(float) * data_size,
                       NULL, NULL);
output = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
                        sizeof(float) * data_size,
                        NULL, NULL);
```

```
// Write our data set into the input array in device
memory
 err = clEnqueueWriteBuffer(commands, input, CL_TRUE, 0,
sizeof(float) * data_size, data, 0, NULL, NULL);
```

… run computations here …

Write and Read Data on GPU

```
// Create the input and output arrays in device memory
for our calculation
input = clCreateBuffer(context,  CL_MEM_READ_ONLY,
                       sizeof(float) * data_size,
                       NULL, NULL);
output = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
                        sizeof(float) * data_size,
                        NULL, NULL);
```

```
// Write our data set into the input array in device
memory
err = clEnqueueWriteBuffer(commands, input, CL_TRUE, 0,
sizeof(float) * data_size, data, 0, NULL, NULL);
```

… run computations here …

```
// Read back the results from the device
err = clEnqueueReadBuffer( commands, output, CL_TRUE, 0,
sizeof(float) * data_size, gpu_results, 0, NULL, NULL );
```

The Computation

```
// Set the arguments to our compute kernel
err = 0;
err  = clSetKernelArg(kernel, 0, sizeof(cl_mem), &input);
err |= clSetKernelArg(kernel, 1, sizeof(cl_mem), &output);
err |= clSetKernelArg(kernel, 2, sizeof(unsigned long), &data_size);
```

The Computation

```
// Set the arguments to our compute kernel
err = 0;
err  = clSetKernelArg(kernel, 0, sizeof(cl_mem), &input);
err |= clSetKernelArg(kernel, 1, sizeof(cl_mem), &output);
err |= clSetKernelArg(kernel, 2, sizeof(unsigned long), &data_size);
```

```
__kernel void square( __global float* input,
                      __global float* output,
                      const unsigned long data_size)
{
    int i = get_global_id(0);
    if(i < data_size)
        output[i] = sqrt(input[i]);
}
```

The Computation

```
__kernel void square( __global float* input,
                      __global float* output,
                      const unsigned long data_size)
{
   int i = get_global_id(0);
   if(i < data_size)
       output[i] = sqrt(input[i]);
}
```

```
// Set the arguments to our compute kernel
err = 0;
err  = clSetKernelArg(kernel, 0, sizeof(cl_mem), &input);
err |= clSetKernelArg(kernel, 1, sizeof(cl_mem), &output);
err |= clSetKernelArg(kernel, 2, sizeof(unsigned long), &data_size);

// Get the maximum work group size for executing
// the kernel on the device
err = clGetKernelWorkGroupInfo(kernel, device_id,
CL_KERNEL_WORK_GROUP_SIZE,
sizeof(local), &local, NULL);
```

THE COMPUTATION

```
// Set the arguments to our compute kernel
err = 0;
err  = clSetKernelArg(kernel, 0, sizeof(cl_mem), &input);
err |= clSetKernelArg(kernel, 1, sizeof(cl_mem), &output);
err |= clSetKernelArg(kernel, 2, sizeof(unsigned long), &data_size);

// Get the maximum work group size for executing
// the kernel on the device
err = clGetKernelWorkGroupInfo(kernel, device_id,
                               CL_KERNEL_WORK_GROUP_SIZE,
                               sizeof(local), &local, NULL);

// Execute the kernel over the entire range of our 1D input data set
// using the maximum number of work group items for this device
global = data_size;
err = clEnqueueNDRangeKernel(commands, kernel, 1, NULL,
                                    &global, &local, 0, NULL, NULL);
```

The Computation

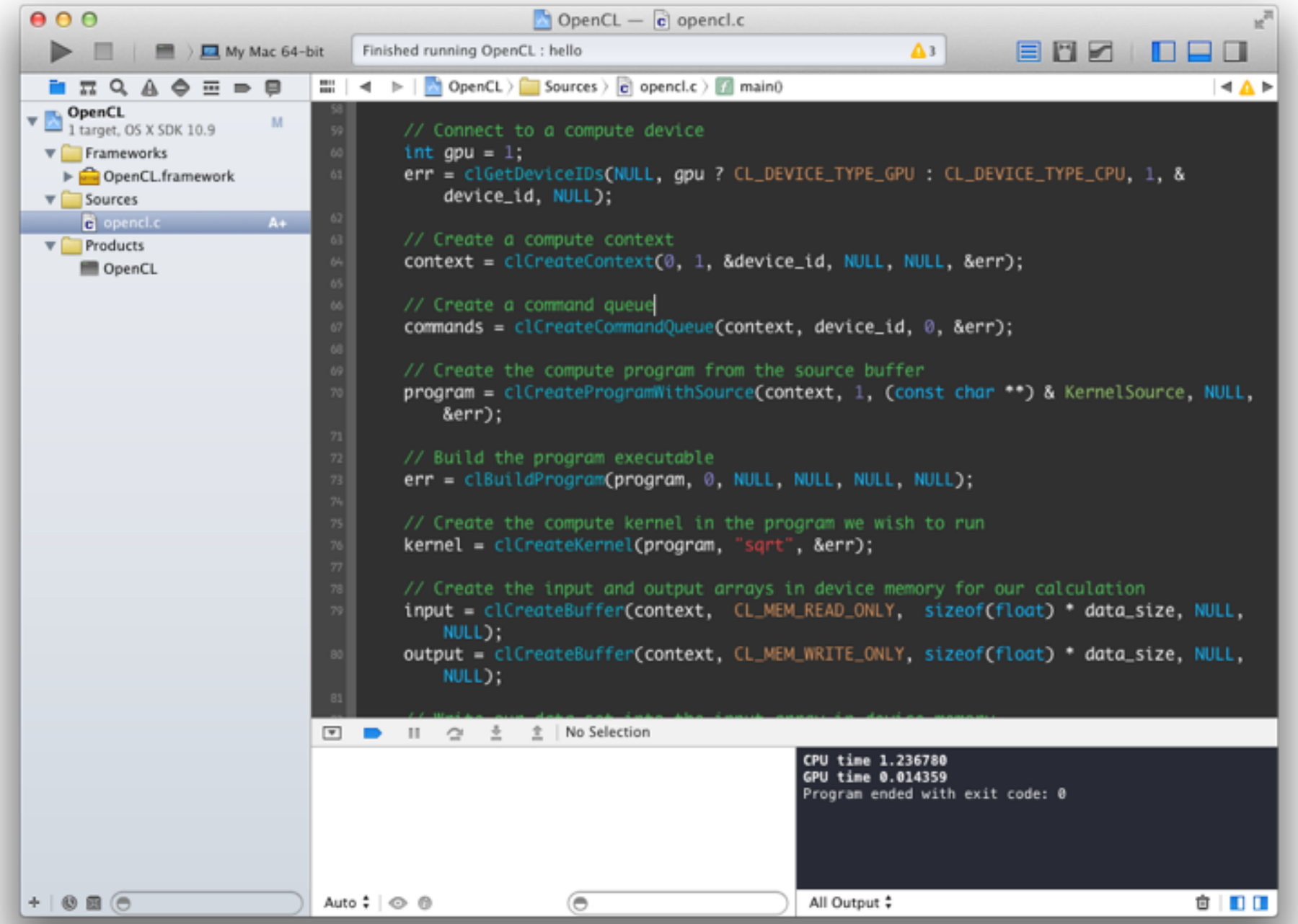
```
// Set the arguments to our compute kernel
err = 0;
err  = clSetKernelArg(kernel, 0, sizeof(cl_mem), &input);
err |= clSetKernelArg(kernel, 1, sizeof(cl_mem), &output);
err |= clSetKernelArg(kernel, 2, sizeof(unsigned long), &data_size);

// Get the maximum work group size for executing
// the kernel on the device
err = clGetKernelWorkGroupInfo(kernel, device_id,
                               CL_KERNEL_WORK_GROUP_SIZE,
                               sizeof(local), &local, NULL);

// Execute the kernel over the entire range of our 1D input data set
// using the maximum number of work group items for this device
global = data_size;
err = clEnqueueNDRangeKernel(commands, kernel, 1, NULL,
                                    &global, &local, 0, NULL, NULL);

// Wait for the command commands to get serviced before
//reading back results
clFinish(commands);
```

Demo



Open, study and run the project from the **`code/OpenCL`**

PART III
CHAPTER 2

CUDA Programming model

- CPU is called "*host*"
  - Move data CPU <-> GPU memory `cudaMemcopy`
  - Allocate memory `cudaMalloc`
  - Launch *kernels* on GPU
- GPU is called "*device*"

CUDA PROGRAMMING MODEL

- CPU is called "*host*"
  - Move data CPU <-> GPU memory `cudaMemcopy`
  - Allocate memory `cudaMalloc`
  - Launch *kernels* on GPU
- GPU is called "*device*"

Typical CUDA program

1. CPU allocates memory on GPU
2. CPU copies data to GPU memory
3. CPU launches kernels on GPU (process the data)
4. CPU copies results back to CPU memory

CUDA Programming model

- CPU is called "*host*"
  - Move data CPU <-> GPU memory `cudaMemcopy`
  - Allocate memory `cudaMalloc`
  - Launch *kernels* on CPU
- GPU is calle

Very similar to the logic of OpenCL

Typical CUDA p

1. CPU allocates memory on GPU
2. CPU copies data to GPU memory
3. CPU launches kernels on GPU (process the data)
4. CPU copies results back to CPU memory

EXAMPLE

```
#include <stdio.h>

// Kernel
__global__ void cube(float * d_out, float * d_in){
    int id = threadIdx.x;
    d_out[id] = d_in[id] * d_in[id] * d_in[id];
}

int main(int argc, char ** argv) {
    const int ARRAY_SIZE = 96;
    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);
```

EXAMPLE

```
#include <stdio.h>

// Kernel
__global__ void cube(float * d_out, float * d_in){
    int id = threadIdx.x;
    d_out[id] = d_in[id] * d_in[id] * d_in[id];
}

int main(int argc, char ** argv) {
    const int ARRAY_SIZE = 96;
    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);
```

EXAMPLE

```
#include <stdio.h>

// Kernel
__global__ void cube(float * d_out, float * d_in){
    int id = threadIdx.x;
    d_out[id] = d_in[id] * d_in[id] * d_in[id];
}

int main(int argc, char ** argv) {
    const int ARRAY_SIZE = 96;
    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);
```

EXAMPLE

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

// launch the kernel
cube<<<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]);
    }

    cudaFree(d_in);
    cudaFree(d_out);

    return 0;
}
```

EXAMPLE

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

// launch the kernel
cube<<<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]);
    }

    cudaFree(d_in);
    cudaFree(d_out);

    return 0;
}
```

EXAMPLE

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

// launch the kernel
cube<<<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]);
    }

    cudaFree(d_in);
    cudaFree(d_out);

    return 0;
}
```

EXAMPLE

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

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

// copy
cudaMem

// prin
for (in
    pri
}

cudaFre
cudaFre

return
}
```

```
$ 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
$
```

Part IV
"Discussion"

Links

- Code repository for this presentation
  - https://github.com/kuz/Introduction-to-GPU-Computing
  - Feel free to leave feature requests there, ask questions, etc.

- OpenCL tutorials & presentations
  - http://streamcomputing.eu/knowledge/for-developers/tutorials/
  - http://opencl.codeplex.com/wikipage?title=OpenCL%20Tutorials%20-%201&referringTitle=OpenCL%20Tutorials
  - http://www.cc.gatech.edu/~vetter/keeneland/tutorial-2011-04-14/06-intro_to_opencl.pdf

- Introduction to Parallel Computing @ www.udacity.com
  - https://www.udacity.com/course/cs344

- Slides about Compute Shader
  - http://web.engr.oregonstate.edu/~mjb/cs557/Handouts/compute.shader.1pp.pdf

- Introduction to Computer Graphics with codebases
  - https://github.com/konstantint/ComputerGraphics2013

- GLSL Computing (aka “Old school”)
  - http://www.computer-graphics.se/gpu-computing/lab1.html