OpenGL Compute Shader – the Basic Idea

Paraphrased from the ARB_compute_shader spec:

Recent graphics hardware has become extremely powerful. A strong desire to harness this power for work that does not fit the traditional graphics pipeline has emerged. To address this, Compute Shaders are a new single-stage program. They are launched in a manner that is essentially stateless. This allows arbitrary workloads to be sent to the graphics hardware with minimal disturbance to the GL state machine.

In most respects, a Compute Shader is identical to all other OpenGL shaders, with graphics hardware with minimal disturbance to the GL state machine. This allows arbitrary workloads to be sent to the graphics hardware with minimal disturbance to the GL state machine.

Why Not Just Use OpenCL Instead?

OpenCL is great! It does a super job of using the GPU for general-purpose data-parallel computing. But OpenGL is more feature-rich than OpenCL, compute shaders. So, why use Compute Shaders even if you’ve got OpenCL? Here’s what I think:

1. OpenCL requires installing a separate driver and separate libraries. While this is not a huge deal, it does take time and effort. When everyone catches up to OpenGL 4.3, Compute Shaders will just “be there” as part of core OpenGL.
2. Compute Shaders use the GLSL language, something that all OpenGL programmers should already be familiar with (or will be soon).
3. Compute shaders use the same context as does the OpenGL rendering pipeline. There is no need to acquire and release the context as OpenGL+OpenCL must do.
4. I’m assuming that calls to OpenGL compute shaders are more lightweight than calls to OpenCL kernels are. (true?) This should result in better performance. (true? how much?)
5. Using OpenCL is somewhat cumbersome. It requires a lot of setup (queries, platforms, devices, queues, kernels, etc.). Compute Shaders look to be more convenient. They just kind of flow in with the graphics.

If I Know GLSL, What Do I Need to Do Differently to Write a Compute Shader?

Not much:
1. A Compute Shader is created just like any other GLSL shader, except that its type is GL_COMPUTE_SHADER. (duh…). You compile it and link it just like any other GLSL shader program.
2. A Compute Shader must be in a shader program all by itself. There cannot be vertex, fragment, etc. shaders in there with it. (why?)
3. A Compute Shader has access to uniform variables and buffer objects, but cannot access any pipeline variables such as attributes or variables from other stages. It stands alone.
4. A Compute Shader needs to declare the number of work-items in each of its work-groups in a special GLSL layout statement.

Passing Data to the Compute Shader Happens with a Cool New Buffer Type – the Shader Storage Buffer Object

The tricky part is getting data in and out of the Compute Shader. This trickiness comes from the specification phrase: “In most respects, a Compute Shader is identical to all other OpenGL shaders, with similar status, uniforms, and other such properties. It has access to many of the same data as all other shader types, such as textures, image textures, atomic counters, and so on. However, the Compute Shader has no predefined inputs, nor any fixed-function outputs. It cannot be part of a rendering pipeline and its visible side effects are through its actions on shader storage buffers, image textures, and atomic counters.”

Shader Storage Buffer Objects are created with arbitrary data (same as other buffer objects), but what is new is that the shaders can read and write them in the same C-like way as they were created, including accessing parts of the buffer as an array of structures – perfect for data-parallel computing!
Passing Data to the Compute Shader Happens with a Cool New Buffer Type – the Shader Storage Buffer Object

And, like other OpenGL buffer types, Shader Storage Buffer Objects can be bound to indexed binding points, making them easy to access from inside the Compute Shaders.

OpenGL Context

Shader Storage Buffer Object

The Example We Are Going to Use Here is a Particle System

The Compute Shader Moves the Particles by Recomputing the Position and Velocity Buffers

The OpenGL Rendering Draws the Particles by Reading the Position Buffer

Setting up the Shader Storage Buffer Objects in Your C Program

```
#define NUM_PARTICLES 1024*1024 // total number of particles to move
#define WORK_GROUP_SIZE 128 // # work-items per work-group
struct pos
{
    float x, y, z, w; // positions
};
struct vel
{
    float vx, vy, vz, vw; // velocities
};
struct color
{
    float r, g, b, a; // colors
};

// need to do the following for both position, velocity, and colors of the particles:
GLuint posSSbo;
GLuint velSSbo;
GLuint colSSbo;
```

Note that .w and .vw are not actually needed. But, by making these structure sizes a multiple of 4 floats, it doesn’t matter if they are declared with the std140 or the std430 qualifier. I think this is a good thing.

The Data Needs to be Divided into Large Quantities call Work-Groups, each of which is further Divided into Smaller Units Called Work-Items

20 total items to compute: 5 Work Groups

```
5x4 20

WorkGroups = GlobalInvocationSize / WorkGroupSize
```

The Invocation Space can be 1D, 2D, or 3D. This one is 1D.

20x12 (=240) total items to compute: 4 Work Groups

```
5x4 4x3

WorkGroups = GlobalInvocationSize / WorkGroupSize
```

The Invocation Space can be 1D, 2D, or 3D. This one is 2D.
Running the Compute Shader from the Application

```
void glDispatchCompute( num_groups_x, num_groups_y, num_groups_z );
```

If the problem is 2D, then
num_groups_z = 1

If the problem is 1D, then
num_groups_y = 1 and
num_groups_z = 1

A Mechanical Equivalent...

```
http://news.cision.com

"Streaming Multiprocessor"
"CUDA Cores"
"Data"
```

Invoking the Compute Shader in Your C Program

```
[glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 4, posSSbo );
glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 5, velSSbo );
glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 6, colSSbo );
]
[glUseProgram( MyComputeShaderProgram );
glDispatchCompute( NUM_PARTICLES / WORK_GROUP_SIZE, 1, 1 );
glMemoryBarrier( GL_SHADER_STORAGE_BARRIER_BIT );
]
[glUseProgram( MyRenderingShaderProgram );
glBindBuffer( GL_ARRAY_BUFFER, posSSbo );
]
```

Writing a C++ Class to Handle Everything is Fairly Straightforward

```
//GLSLProgram *Particles = new GLSLProgram( );
bool valid = Particles->Create( "particles.cs" );
if( ! valid ) { . . . }

Particles->Use( );
Particles->DispatchCompute( NUM_PARTICLES / WORK_GROUP_SIZE, 1, 1 );

Render->Use( );
```

Setup:

```

Using:

Particles->Use( );
Particles->DispatchCompute( NUM_PARTICLES / WORK_GROUP_SIZE, 1, 1 );
```

Special Pre-set Variables in the Compute Shader

```
in uvec3 gl_NumWorkGroups ;
const uvec3 gl_WorkGroupSize ;
in uvec3 gl_WorkGroupID ;
in uvec3 gl_GlobalInvocationID ;
in uvec3 gl_LocalInvocationID ;
in uint gl_LocalInvocationIndex ;
```

Two Pre-set Variables in the Compute Shader

```
in uvec3 gl_NumWorkGroups ;
const uvec3 gl_WorkGroupSize ;
in uvec3 gl_WorkGroupID ;
in uvec3 gl_GlobalInvocationID ;
in uvec3 gl_LocalInvocationID ;
in uint gl_LocalInvocationIndex ;
```

```
< gl_WorkGroupID < gl NumWorkGroups
< gl_LocalInvocationID < gl WorkGroupSize
```

```
< gl_GlobalInvocationID < ( gl_NumWorkGroups * gl_WorkGroupSize )
< gl_LocalInvocationIndex < ( gl_LocalInvocationID.y * gl_WorkGroupSize.x + gl_LocalInvocationID.x )
```

The Particle System Compute Shader -- Setup

```
// Extension 430 compatibility:
enable GL_ARB_compute_shader ;
// Extension GL_ARB_shader_storage_buffer_object :
enable;
```

```
layout( std140, binding=4 )  buffer  Pos
{
   vec4  Positions[   ]; // array of structures
};
layout( std140, binding=5 )  buffer Vel
{
   vec4  Velocities[   ]; // array of structures
};
layout( std140, binding=6 )  buffer  Col
{
   vec4  Colors[   ]; // array of structures
};
```

```
const vec3 G = vec3(0., -9.8, 0.);
const float DT = 0.1;

uint gid = gl_GlobalInvocationID.x; // the y and z are both 1 in this case
vec3 p = Positions[gid].xyz;
vec3 v = Velocities[gid].xyz;
vec3 pp = p + v*DT + 0.5*DT*DT*G;
vec3 vp = v + G*DT;

if( IsInsideSphere(pp, Sphere) )
{
    vp = BounceSphere(p, v, Sphere);
    pp = p + vp*DT + 0.5*DT*DT*G;
}

Positions[gid].xyz = pp;
Velocities[gid].xyz = vp;

Graphics Trick Alert: Making the bounce happen from the surface of the sphere is

There are some applications, such as image convolution, where threads within a work-

group need to operate on each other’s input or output data. In those cases, it is usual-

ty a good idea to create a local shared array that all of the threads in the work-group

can access. You do it like this:

layout( std140, binding = 6 ) buffer Col
{
    vec4 Colors[ ];
};
layout( shared ) vec4 rgba[ gl_WorkGroupSize.x ];
uint gid = gl_GlobalInvocationID.x;
uint lid = gl_LocalInvocationID.x;
rgba[lid] = Colors[gid];
memory_barrier_shared();
<< operate on the rgba array elements >>
Colors[gid] = rgba[lid];