OpenGL Compute Shaders

Mike Bailey
mjb@cs.oregonstate.edu
Oregon State University

OpenGL Compute Shader – the Basic Idea

Application Invokes OpenGL Rendering which Reads the Buffer Data

Why Not Just Use OpenCL Instead?

OpenCL is great! It does a super job of using the GPU for general-purpose data-parallel computing. And, OpenCL is more feature-rich than OpenGL 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.

The bottom line is that I will continue to use OpenCL for the big, bad stuff. But, for lighter-weight data-parallel computing that interacts with graphics, I will use the Compute Shaders.

I suspect that a good example of a lighter-weight data-parallel graphics-related application is a particle system. This will be shown here in the rest of these notes. I hope I’m right.

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.

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

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

```c
#include <GL/glew.h>
#include <GLFW/glfw3.h>

const char* vertexShaderSource = 
"#version 430
";

const char* fragmentShaderSource = 
"#version 430
";

void setupShaderProgram() {
    // ... setup code...
}
```

Note that `a` and `b` are not actually needed. But, by making these structure sizes a multiple of 4 bytes, it doesn’t matter if they are declared with the `std430` or the `std140` qualifier. I think this is a good thing. (is it?)
glGenBuffers( 1, &posSSbo);
glBindBuffer( GL_SHADER_STORAGE_BUFFER, posSSbo );
glBufferData( GL_SHADER_STORAGE_BUFFER, NUM_PARTICLES * sizeof(struct pos), NULL, GL_STATIC_DRAW );
GLint bufMask = GL_MAP_WRITE_BIT | GL_MAP_INVALIDATE_BUFFER_BIT ; // the invalidate makes a big difference when re-writing
struct pos *points = (struct pos *) glMapBufferRange( GL_SHADER_STORAGE_BUFFER, 0, NUM_PARTICLES * sizeof(struct pos), bufMask );
for( int i = 0; i < NUM_PARTICLES; i++ )
{
    points[i].x = Ranf( XMIN, XMAX );
    points[i].y = Ranf( YMIN, YMAX );
    points[i].z = Ranf( ZMIN, ZMAX );
    points[i].w = 1.;
}

ließlich Buffer(GL_SHADER_STORAGE_BUFFER)

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

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

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

Special Pre-set Variables in the Compute Shader

in uvec3 gl_NumWorkGroups ; Some numbers in the glDispatchCompute call
const uvec3 gl_WorkGroupSize ; Some numbers in the layout _local_Insta_*
in uvec3 gl_WorkGroupId ; Which workgroup this thread is in
in uvec3 gl_LocalInvocationID ; Where this thread is in the current workgroup
in uvec3 gl_GlobalInvocationID ; Where this thread is in all the work items
in unit gl_LocalInvocationIndex ; 1D representation of the gl LocalInvocationID (used for indexing into a shared array)

0 ≤ gl_WorkGroupId < gl_NumWorkGroups – 1
0 ≤ gl_LocalInvocationID < gl_WorkGroupSize - 1
0 ≤ gl_GlobalInvocationID < gl_WorkGroupId + gl_WorkGroupSize + gl_LocalInvocationIndex
0 ≤ gl_LocalInvocationIndex < gl_WorkgroupId + 1
The Particle System Compute Shader -- Setup

#version 430 compatibility
#extension GL_ARB_compute_shader : enable
#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
}
layout( local_size_x = 128, local_size_y = 1, local_size_z = 1 ) in;

The Particle System Compute Shader -- The Physics

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

The Particle System Compute Shader -- How About Introducing a Bounce?

const vec4 SPHERE = vec4(-100., -800., 0., 600.); // x, y, z, r
// (could also have passed this in)
vec3 Bounce( vec3 vin, vec3 n )
{
vec3 vout = reflect( vin, n );
return vout;
}
vec3 BounceSphere( vec3 p, vec3 v, vec4 s )
{
vec3 n = normalize( p - s.xyz );
return Bounce( v, n );
}

bool IsInsideSphere( vec3 p, vec4 s )
{
float r = length( p - s.xyz );
return ( r < s.w );
}

The Bouncing Particle System Compute Shader -- What Does It Look Like?