



**Vulkan.**

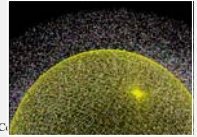
**A Review of OpenGL Compute Shaders**





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

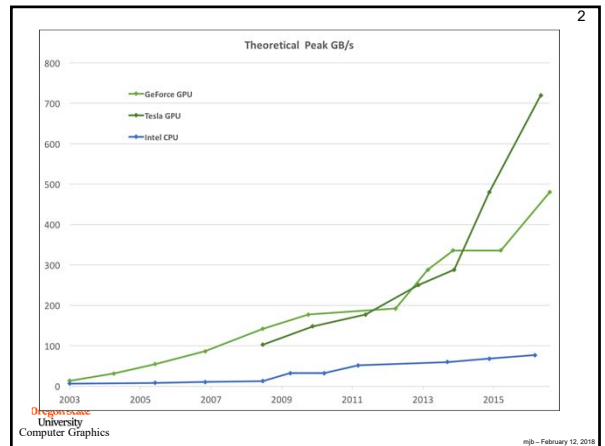
This work is licensed under a [Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License](https://creativecommons.org/licenses/by-nc-nd/4.0/)



Application Invokes the Compute Shader to Modify the Object's Buffer Data

Application Invokes OpenGL Rendering which Reads the Buffer Data

CopingComputeShaders.ppt © 2018



**Why have GPUs Been Outpacing CPUs in Performance?**

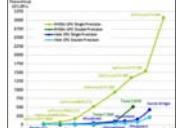
Due to the nature of graphics computations, GPU chips are customized to handle **streaming data**.

Another reason is that GPU chips do not need the significant amount of **cache** space that occupies much of the real estate on general-purpose CPU chips. The GPU die real estate can then be re-targeted to hold more cores and thus to produce more processing power.

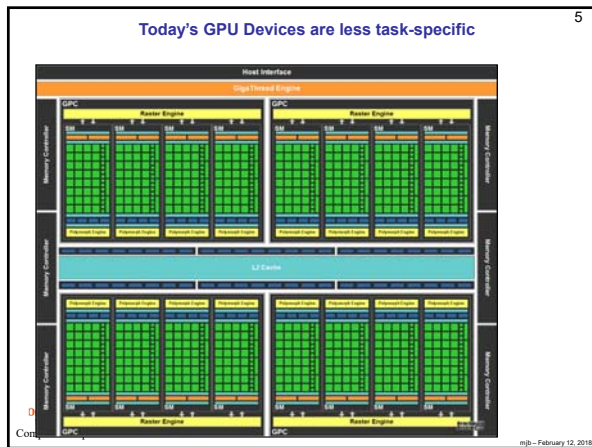
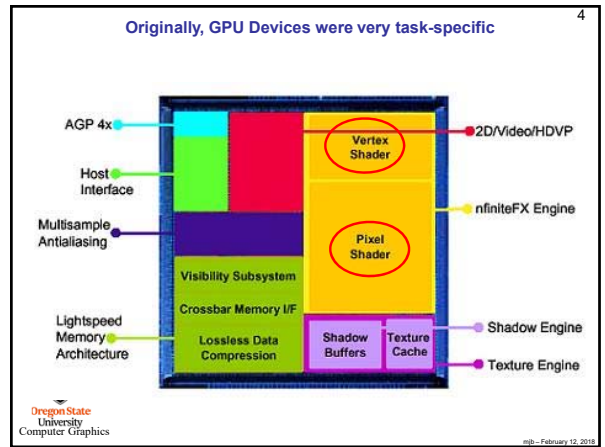
Another reason is that general CPU chips contain on-chip logic to do **branch prediction**. This, too, takes up chip die space.

Another reason is that general CPU chips contain on-chip logic to process instructions **out-of-order** if the CPU is blocked and is waiting on something (e.g., a memory fetch). This, too, takes up chip die space.

So, which is better, CPU or GPU?  
**It depends on what you are trying to do!**



Oregon State University Computer Graphics mjb - February 12, 2018



Consider the architecture of the NVIDIA Titan Black we have in our rabbit System

15 Streaming Multiprocessors (SMs) / chip  
192 cores / SM  
Wow! 2880 cores / chip? Really?

Oregon State University Computer Graphics mjb - February 12, 2018

### The "Core-Score". How can this be?

1024 4 8  
DOWN YDS. TO GO BALL ON

Oregon State University Computer Graphics  
#10 - February 12, 2018

### What is a "Core" in the GPU Sense?

Look closely, and you'll see that NVIDIA really calls these "CUDA Cores"

Look even more closely and you'll see that these CUDA Cores have no control logic – they are **pure compute units**. (The surrounding SM has the control logic.)

Other vendors refer to these as "Lanes". You might also think of them as 192-way SIMD.

Oregon State University Computer Graphics  
#10 - February 12, 2018

### A Mechanical Equivalent...

"Streaming Multiprocessor"

"CUDA Cores"

"Data"

University Computer Graphics  
<http://news.cision.com>  
#10 - February 12, 2018

### How Can You Gain Access to that GPU Power?

There are three ways:

1. Write a graphics display program (≥ 1985)
2. Write an application that looks like a graphics display program, but uses the fragment shader to do some computation (≥ 2002)
3. Write in OpenCL (or CUDA), which looks like C++ (≥ 2006)

Oregon State University Computer Graphics  
#10 - February 12, 2018

### OpenGL Compute Shader – the Basic Idea

A Shader Program, with only a Compute Shader in it

Application Invokes the Compute Shader to Modify the OpenGL Buffer Data

Application Invokes OpenGL Rendering which Reads the Buffer Data

Another Shader Program, with pipeline rendering in it

Oregon State University Computer Graphics  
#10 - February 12, 2018

### 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.

More information on items 3 and 4 are coming up . . .

Oregon State University Computer Graphics  
#10 - February 12, 2018

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

The tricky part is getting data into and out of the Compute Shader. The 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."

OpenGL programs have access to general arrays of data, and also access to OpenGL arrays of data in the form of buffer objects. Compute Shaders, looking like other shaders, haven't had *direct* access to general arrays of data (hacked access, yes; direct access, no). But, because Compute Shaders represent opportunities for massive data-parallel computations, that is exactly what you want them to use.

Thus, OpenGL 4.3 introduced the **Shader Storage Buffer Object**. This is very cool, and has been needed for a long time!

**Shader Storage Buffer Object**

Read/Write for Arbitrary data, including Arrays of Structures

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 treating parts of the buffer as an array of structures – perfect for data-parallel computing!

#13 – February 12, 2018

### 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.

#14 – February 12, 2018

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

#15 – February 12, 2018

### 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;
    
```

#16 – February 12, 2018

### Setting up the Shader Storage Buffer Objects in Your C Program

```

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 = Randf( XMIN, XMAX );
    points[ i ].y = Randf( YMIN, YMAX );
    points[ i ].z = Randf( ZMIN, ZMAX );
    points[ i ].w = 1.;
}
glUnmapBuffer( GL_SHADER_STORAGE_BUFFER );

glGenBuffers( 1, &velSSbo);
glBindBuffer( GL_SHADER_STORAGE_BUFFER, velSSbo );
glBufferData( GL_SHADER_STORAGE_BUFFER, NUM_PARTICLES * sizeof(struct vel), NULL, GL_STATIC_DRAW );

struct vel *vels = (struct vel *)glMapBufferRange( GL_SHADER_STORAGE_BUFFER, 0, NUM_PARTICLES * sizeof(struct vel), bufMask );
for( int i = 0; i < NUM_PARTICLES; i++)
{
    vels[ i ].vw = Randf( VXMIN, VXMAX );
    vels[ i ].vy = Randf( VYMIN, VYMAX );
    vels[ i ].vz = Randf( VZMIN, VZMAX );
    vels[ i ].w = 0.;
}
glUnmapBuffer( GL_SHADER_STORAGE_BUFFER );
    
```

The same would need to be done for the color shader storage buffer object

#17 – February 12, 2018

### The Data Needs to be Divided into Large Quantities call Work-Groups, each of 18 which is further Divided into Smaller Units Called Work-Items

20 total items to compute:

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

$$\#WorkGroups = \frac{GlobalInvocationSize}{WorkGroupSize}$$

$$5 \times 4 = \frac{20}{4}$$

#18 – February 12, 2018

### The Data Needs to be Divided into Large Quantities call Work-Groups, each of 19 which is further Divided into Smaller Units Called Work-Items

20x12 (=240) total items to compute:

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

$$\#WorkGroups = \frac{GlobalInvocationSize}{WorkGroupSize}$$

$$5 \times 4 = \frac{20 \times 12}{4 \times 3}$$

Oregon State University Computer Graphics

### 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

Oregon State University Computer Graphics

### 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 );
glVertexAttribPointer( 4, GL_FLOAT, 0, (void *)0 );
glEnableVertexAttribArray( GL_VERTEX_ARRAY );
glDrawArrays( GL_POINTS, 0, NUM_PARTICLES );
glDisableVertexAttribArray( GL_VERTEX_ARRAY );
glBindBuffer( GL_ARRAY_BUFFER, 0 );
```

The Compute Shader Moves the Particles by (Re)computing the Position and Velocity Buffers

The OpenGL Rendering Draws the Particles by Reading the Position Buffer

Oregon State University Computer Graphics

### Special Pre-set Variables in the Compute Shader

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

Same numbers as in the glDispatchCompute call

Same numbers as in the layout local\_size\_\*

Which workgroup this thread is in

Where this thread is in the current workgroup

Where this thread is in all the work items

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
gl_GlobalInvocationID = gl_WorkGroupID * gl_WorkGroupSize + gl_LocalInvocationID
gl_LocalInvocationIndex = gl_LocalInvocationID.x * gl_WorkGroupSize.y * gl_WorkGroupSize.z + gl_LocalInvocationID.y * gl_WorkGroupSize.x + gl_LocalInvocationID.z
```

Oregon State University Computer Graphics

### 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;
```

You can use the empty brackets, but only on the last element of the buffer. The actual dimension will be determined for you when OpenGL examines the size of this buffer's data store.

Oregon State University Computer Graphics

### The Particle System Compute Shader -- The Physics

```
#define POINT vec3
#define VECTOR vec3
#define SPHERE vec4

const VECTOR G = VECTOR( 0., -9.8, 0. );
const float DT = 0.1;
...
uint gid = gl_GlobalInvocationID.x;
// the .y and .z are both 1 in this case

POINT p = Positions[ gid ].xyz;
VECTOR v = Velocities[ gid ].xyz;

POINT pp = p + v*DT + .5*DT*DT*G;
VECTOR vp = v + G*DT;

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

$$p^1 = p + v \cdot t + \frac{1}{2} G \cdot t^2$$

$$v^1 = v + G \cdot t$$

Oregon State University Computer Graphics

### The Particle System Compute Shader – How About Introducing a Bounce?

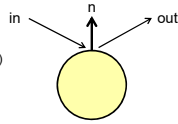
25

```
const SPHERE S = vec4(-100., -800., 0., 600.); // x, y, z, r
// (could also have passed this in)

VECTOR
Bounce(VECTOR vin, VECTOR n)
{
    VECTOR vout = reflect(vin, n);
    return vout;
}

VECTOR
BounceSphere(POINT p, VECTOR v, SPHERE s)
{
    VECTOR n = normalize(p - s.xyz);
    return Bounce(v, n);
}

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



### The Particle System Compute Shader – How About Introducing a Bounce?

26

```
uint gid = gl_GlobalInvocationID.x; // the .y and .z are both 1 in this case

POINT p = Positions[ gid ].xyz;
VECTOR v = Velocities[ gid ].xyz;

POINT pp = p + v*DT + .5*DT*DT*G;
VECTOR vp = v + G*DT;

if( IsInsideSphere( pp, S ) )
{
    vp = BounceSphere( p, v, S );
    pp = p + vp*DT + .5*DT*DT*G;
}

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

$$p' = p + v \cdot t + \frac{1}{2} G \cdot t^2$$

$$v' = v + G \cdot t$$

**Graphics Trick Alert:** Making the bounce happen from the surface of the sphere is time-consuming. Instead, bounce from the previous position in space. If DT is small enough, nobody will ever know...

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

27

