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!*
Today’s GPU Devices are less task-specific

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?

The “Core-Score”. How can this be?

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.
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)

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.

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 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.”

OpenCL 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!

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

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.

The OpenGL Rendering Draws the Particles by Reading the Position Buffer

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

```c
#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
};

GLint posSSbo;
GLint velSSbo;
GLint colSSbo;
```
Setting up the Shader Storage Buffer Objects in Your C Program

```c
mjb – February 12, 2018
17

Computer Graphics

17

glGenBuffers( 1, &posSSbo);

17

GLuint bufMask = GL_MAP_WRITE_BIT | GL_MAP_INVALIDATE_BUFFER_BIT; // the invalidate makes a big difference when re-writing

17

struct pos *points = (struct pos *) glMapBufferRange( GL_SHADER_STORAGE_BUFFER, 0, NUM_PARTICLES * sizeof(struct pos), bufMask );

17

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.;
}

17

glUnmapBuffer( GL_SHADER_STORAGE_BUFFER );

glGenBuffers( 1, &velSSbo);

17

glBindBuffer( GL_SHADER_STORAGE_BUFFER, velSSbo );

17

glBufferData( GL_SHADER_STORAGE_BUFFER, NUM_PARTICLES * sizeof(struct vel), NULL, GL_STATIC_DRAW );

17

struct vel *vels = (struct vel *) glMapBufferRange( GL_SHADER_STORAGE_BUFFER, 0, NUM_PARTICLES * sizeof(struct vel), bufMask );

17

for( int i = 0; i < NUM_PARTICLES; i++ )
{
   vels[ i ].vx = Ranf( VXMIN, VXMAX );
   vels[ i ].vy = Ranf( VYMIN, VYMAX );
   vels[ i ].vz = Ranf( VZMIN, VZMAX );
   vels[ i ].vw = 0.;
}

17

glUnmapBuffer( 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

```
20 total items to compute:

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

\text{5x4} = \frac{20}{4}
```

Running the Compute Shader from the Application

```c
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
Invoking the Compute Shader in Your C Program

```c
#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 -- Setup

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

```c
uint gid = gl_GlobalInvocationID.x; // the y and z are both 1 in this case
POIINT     p  = Positions[  gid ];
VECTOR  v  = Velocities[  gid ];
POINT      pp = p + v*DT + .5*DT*DT*G;
VECTOR  vp = v + G*DT;
Positions[  gid ] = pp;
Velocities[  gid ] = vp;
```

Special Pre-set Variables in the Compute Shader

```c
in uvec3  gl_NumWorkGroups ;  // same numbers as in the glDispatchComputes call
const uvec3  gl_WorkGroupSize ;  // same numbers as in the layout local_size *
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 uint   gl_LocalInvocationIndex ;  // 1D representation of the gl_LocalInvocationId
          // (used for indexing into a shared array)
```

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

const SPHERE S = vec4(-100., -800., 0., 600.); // x, y, z, r
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?

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;

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?