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 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.

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

So, which is better, CPU or GPU?

It depends on what you are trying to do!

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.

A Mechanical Equivalent…

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)

OpenGL Compute Shader – the Basic Idea

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 (ouch…). 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 array statement.

More information on items 3 and 4 are coming up…
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 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!

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.

Setting up the Shader Storage Buffer Objects in Your C Program

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 Data Needs to be Divided into Large Quantities called 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 2D.

```
#GlobalInvocationSize = WorkGroups * WorkGroupSize
5x4 = 20x12 / 4x3
4 Work-Groups
5 Work-Groups
```

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

Invoking the Compute Shader in Your C Program

```
#include <GL/glew.h>
void glDispatchCompute( num_groups_x, num_groups_y, num_groups_z );
```

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;
```

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;
```

The Particle System Compute Shader -- Setup

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

```
#include <GL/glew.h>
const int NUM_PARTICLES = 1024;
```

```
#include <GL/glew.h>
```

```
#include <GL/glew.h>
```

```
const int NUM_PARTICLES = 1024;
```

```
#include <GL/glew.h>
```

```
#include <GL/glew.h>
```

```
const int NUM_PARTICLES = 1024;
```
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 );
}

VECTOR Velocities[  gid ];
POINT     Positions[  gid ];

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;

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

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

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