Compute shaders

The future of GPU computing or a late rip-off of Direct Compute?

Compute shaders

Previously a Microsoft concept, Direct Compute

Now also in OpenGL, new kind of shader since the recet OpenGL 4.3

"Bleeding edge"

Why is this important?

Why use that instead of CUDA or OpenCL?

+ Better integration with OpenGL

+ No extra installation!

+ Easier to configure than OpenCL

+ Not NVidia specific like CUDA

+ If you know GLSL, Compute Shaders are (fairly) easy!

Not only plus...

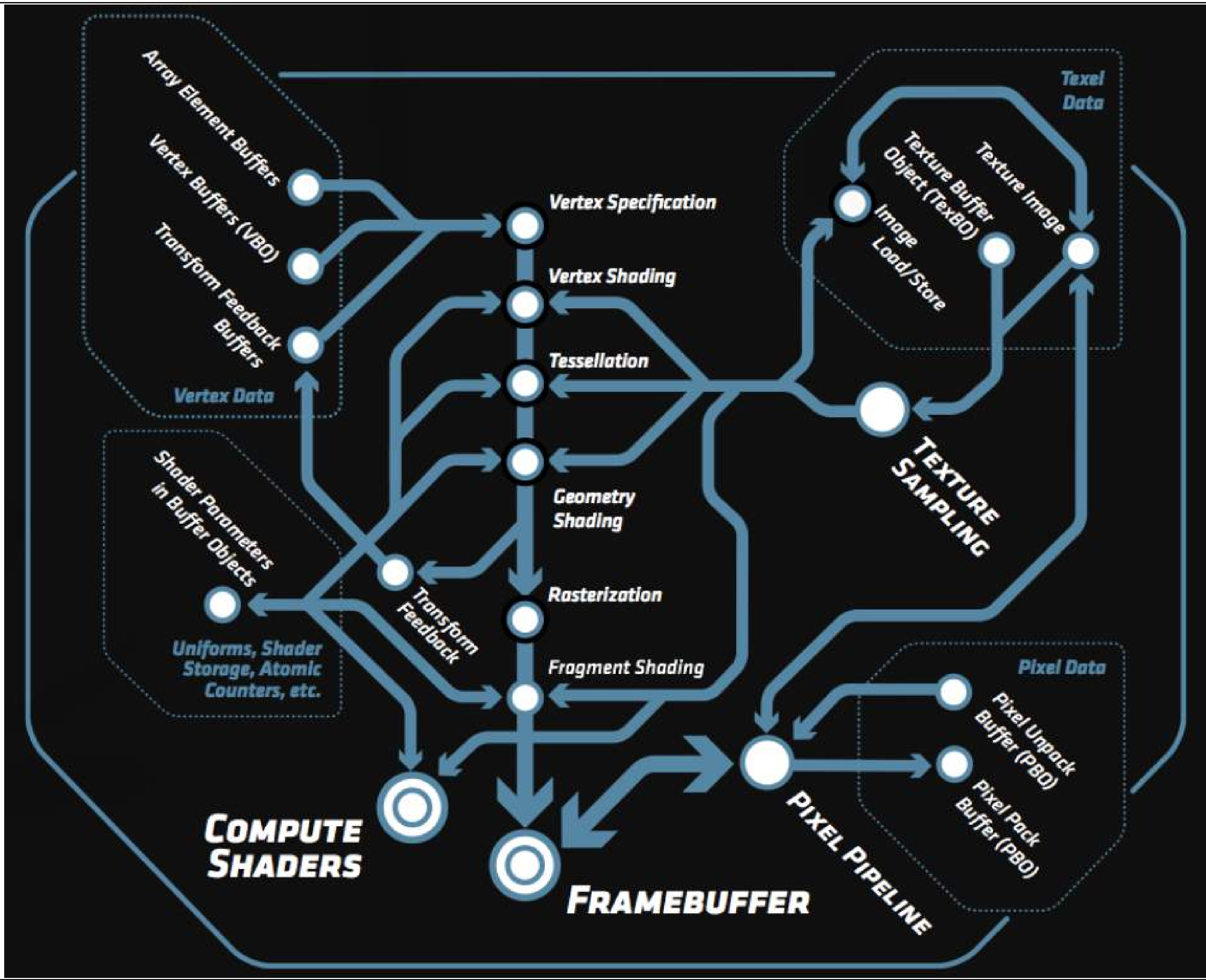
- Steep hardware demands! Kepler + 4.3

- Some new concepts

- Not part of the main graphics pipeline like fragment shaders

Compute shaders run alone, not compiled together with others.


OpenGL 4.3
CORE SPECIFICATION
Array Element Buffers
Vertex Buffers (VBO)
Transform Feedback Buffers
Vertex Data
Vertex Specification
Vertex Shading
Tessellation
Geometry Shading
Rasterization
Fragment Shading
Texel Data
Texture Buffer Object (TexBO)
Texture Image
Image Load/Store
TEXTURE SAMPLING
Shader Parameters in Buffer Objects
Uniforms, Shader Storage, Atomic Counters, etc.
Transform Feedback
Pixel Data
Pixel Unpack Buffer (PBO)
Pixel Pack Buffer (PBO)
COMPUTE SHADERS
FRAMEBUFFER
PIXEL PIPELINE


Array Element Buffers
Vertex Buffers (VBO)
Transform Feedback Buffers
Vertex Data
Vertex Specification
Vertex Shading
Tessellation
Geometry Shading
Rasterization
Fragment Shading
Transform Feedback
Shader Parameters in Buffer Objects
Uniforms, Shader Storage, Atomic Counters, etc.
Texel Data
Image Load/Store
Texture Buffer Object (TexBO)
Texture Image
Texture Sampling
Pixel Data
Pixel Unpack Buffer (PBO)
Pixel Pack Buffer (PBO)
Pixel Pipeline
Compute Shaders
Framebuffer

So how do I use it?

Compiled like other shaders!

Trivial change from the usual shader loader/compilers from graphics programs, just compile as GL_COMPUTE_SHADER.

Easy:

- **Uniforms work as usual**
- **Textures work as usual**

(Note that you can write to textures in Fermi and up!)

Write to textures?

Only newest GPUs.

Call in shader: imageStore()

imageStore(texUnit, texCoord, color);

Needs synchronisation! New call for that:
glMemoryBarrier() and memoryBarrier() in shaders.

GLSL is getting more and more general - but freedom does not always make life easier.

Back to Compute Shaders...

A bit different

No longer not one thread per fragment (output pixel)

Thereby: No thread specific output

<u>Shader Storage Buffer Objects:</u>

General buffer type fpr arbitrary data

Can be declared as an array of structures

Read and written freely by Compute Shaders!

How do I upload input data?

Upload to SSBO:

```
glGenBuffers(1, &ssbo);
glBindBuffer(GL_SHADER_STORAGE_BUFFER, ssbo);
glBufferData(GL_SHADER_STORAGE_BUFFER, size, ptr,
GL_STATIC_DRAW);
```

How does the shader know?

```
glBindBufferBase(GL_SHADER_STORAGE_BUFFER, id,
ssbo);
```

```
layout(std430, binding = id, buffer x {type y[];};
```

Access data in the shader

Set number of threads per block:

layout(local_size_x = width, local_size_y = height)

Thread number:

gl_GlobalInvocation
gl_localInvocation

```
void main()
{
  buffer[gl_GlobalInvocation.x] =
    - buffer[gl_GlobalInvocation.x];
}
```

Execute kernel

```
glUseProgram(program);

glDispatchCompute(sizex, sizey, sizez);
```

The arguments to glDispatchProgram set the number of blocks / workgroups. The number of threads (work items) per block are set by the shader.

Getting output data

```
glBindBuffer(GL_SHADER_STORAGE, ssbo);
ptr = (int *) glMapBuffer(GL_SHADER_STORAGE,
GL_READ_ONLY);
```

Then read from ptr[i]

```
glUnmapBuffer(GL_SHADER_STORAGE);
```

Complete main program:

```
int main(int argc, char **argv)
{
  glutInit (&argc, argv);
  glutCreateWindow("TEST1");

// Load and compile the compute shader
  GLuint p =loadShader("cs.csh");

  GLuint ssbo; //Shader Storage Buffer Object

  // Some data
  int buf[16] = {1, 2, -3, 4, 5, -6, 7, 8, 9,
10, 11, 12, 13, 14, 15, 16};
  int *ptr;

// Create buffer, upload data
  glGenBuffers(1, &ssbo);
  glBindBuffer(GL_SHADER_STORAGE_BUFFER, ssbo);
  glBufferData(GL_SHADER_STORAGE_BUFFER,
      16 * sizeof(int), &buf, GL_STATIC_DRAW);

// Tell it where the input goes!
// "5" matches "layuot" in the shader.
// (Can we ask the shader about the number?
I must try that.)
  glBindBufferBase(GL_SHADER_STORAGE_BUFFER,
         5, ssbo);

// Get rolling!
    glDispatchCompute(16, 1, 1);

// Get data back!
  glBindBuffer(GL_SHADER_STORAGE_BUFFER,
ssbo);
  ptr = (int *)glMapBuffer(
         GL_SHADER_STORAGE_BUFFER,
         GL_READ_ONLY);
  for (int i=0; i < 16; i++)
  {
    printf("%d\n", ptr[i]);
  }
}
```

Simple Compute Shader:

Note: Too many threads for data (16*16*16)

```
#version 430
#define width 16
#define height 16

// Compute shader invocations in each work group

layout(std430, binding = 5) buffer bbs {int bs[];};

layout(local_size_x=width, local_size_y=height) in;

//Kernel Program
void main()
{
  int i = int(gl_LocalInvocationID.x * 2);
  bs[gl_LocalInvocationID.x] = -bs[gl_LocalInvocationID.x];
}
```

Performance:

Preliminary results based on our FFT project

Similar to CUDA, but more time for setup

Can you use Compute Shaders?

My system: CentOS 6.4, GTX 650Ti, OpenGL 4.3 - WORKS

Southfork: GTX 660Ti (great) OpenGL 4.2 - not good enough (yet)

Other test machine: GT630, OpenGL 4.3 - not good enough

Are Compute Shaders an alternative?

- Portable between GPUs and OSes
- Steep hardware demands - for now
- All advantages in the future?

	Portable	Features	Install	Code
CUDA	Weak	Great	Weak	Great
OpenCL	Great	Good	Weak	OK
GLSL Fragment shaders	Great	Weak	Great	Messy
GLSL Compute shaders	Good	Good	Good	OK

GPU computing conclusions

The desktop supercomputer

Fast changing area

Great performance for big problems that fit the architecture

Good performance for many other problems