



Compute shaders

Framtiden för GPU computing eller sen efterrapning
av Direct Compute?

Tidigare rent Microsoft-koncept, Direct Compute

Senare även i OpenGL, ny shadertyp från OpenGL 4.3



Varför är det viktigt?

Varför använda det i stället för CUDA eller OpenCL?

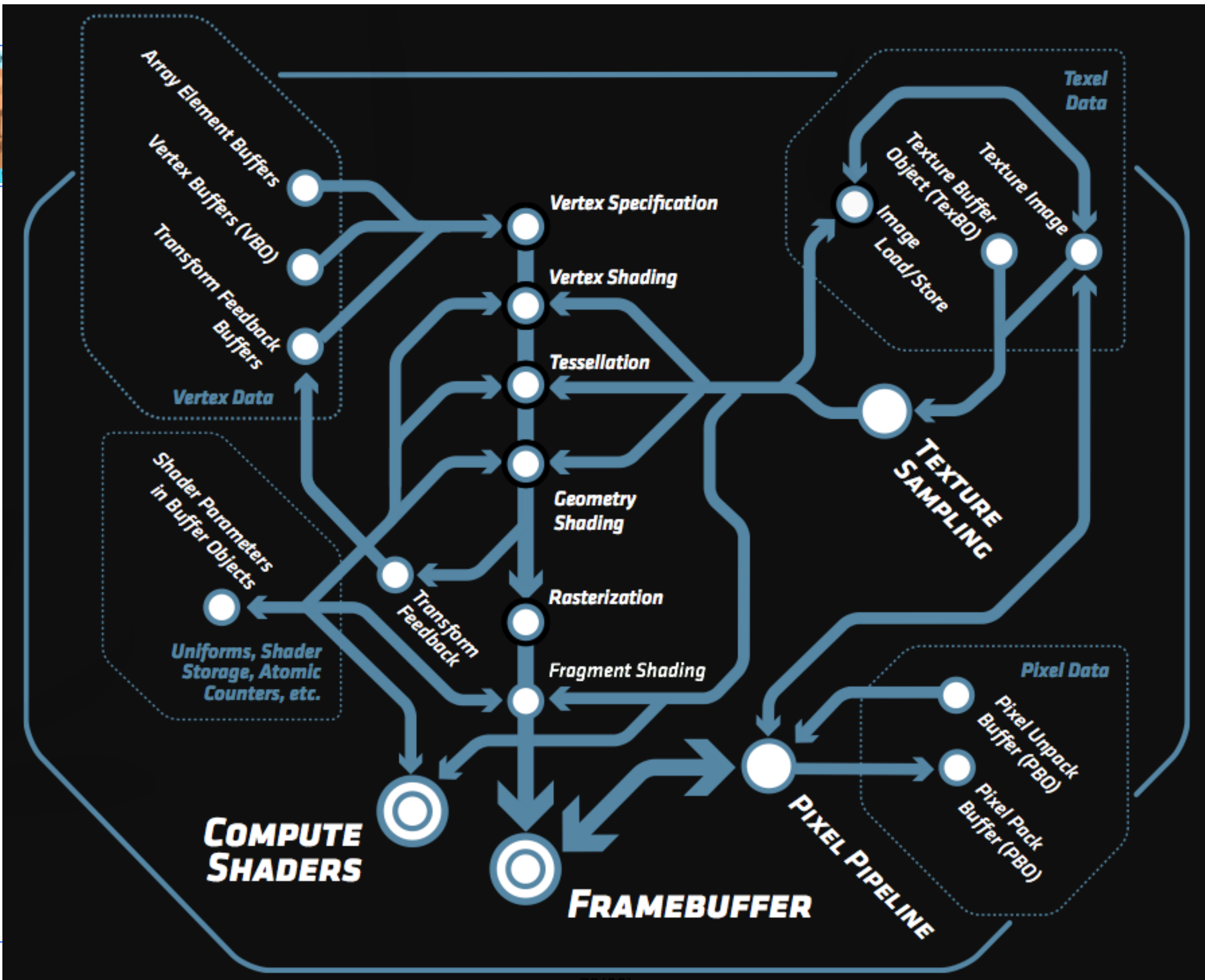
- + Bättre integration med OpenGL
- + Ingen extra installation behövs!
- + Enklare att konfigurera än OpenCL
- + Inte NVidia-specifikt som CUDA
- + Om du kan GLSL så är Compute Shaders (ganska) lätt!



Men det är ju inte bara plus...

- En del nya koncept
- Inte del av grafikpipelinen som fragment shaders
 - Apple har egen lösning

Compute shaders är ensamma, kompileras inte med några andra.





OK, hur gör jag?

Kompileras som alla andra shaders!

Modifiera labbkoden från GL_utilities, kompilera (ensam) som GL_COMPUTE_SHADER.

Lätta saker:

- Uniforms fungerar som vanligt
- Texturer fungerar som vanligt

(OBS att man kan skriva till texturer på Fermi och upp!)



Vadå skriva till texturer?

Japp. (Enbart nyaste GPUerna.)

Anrop i shader: `imageStore()`

```
imageStore(texUnit, texCoord, color);
```

Farligt! Risk för racing! Därför finns nytt anrop för synkronisering:

`glMemoryBarrier()` samt `memoryBarrier()` i shaders.

GLSL går mot allt generellare arkitektur - men frihet gör det inte lättare.

Tillbaka till Compute Shaders...



Lite annorlunda

Attribut finns inte

Inte en tråd per fragment

Shader Storage Buffer Objects:

Generell buffertyp för godtyckliga data

Kan deklareras så shadern ser det som en array av strukturer

Kan läsas och skrivas fritt av Compute Shaders!



Hur får jag in indata?

Ladda upp till SSBO:

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

Hur får shadern veta om den?

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

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




Hur accessar jag data i shadern?

Bestäm antal trådar per block:

```
layout(local_size_x = width, local_size_y = height)
```

Trådnummer:

```
gl_GlobalInvocation  
gl_LocalInvocation
```

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



Hur kör jag kärnan?

```
glUseProgram(program);
```

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

Argumenten till `glDispatchProgram` anger antalet block / workgroups. Antal trådar (work items) per block anges av shadern.



Hur får jag ut utdata?

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

Läs sedan från ptr[i]

```
glUnmapBuffer(GL_SHADER_STORAGE);
```



Komplett huvudprogram:

```
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 "layout" in the shader.

    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]);
    }
}
```



Enkel Compute Shader:

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

OBS: Egentligen alldeles för mycket trådar för data (16*16*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];
}
```



Texturer som in- och utdata

Integration med OpenGL's pipeline

Viktig styrka hos Compute Shaders: Integrationen med OpenGL bättre än övriga.

Två nya demos:

"imagehack"

Genererar en enkel textur.

"juliahack"

Genererar en fraktal. Lite fler styrningsparametrar som uniforms.



Kod för texturaccess

TO DO

Demos finns men kan inte laddas upp än



Detta kan jag inte täcka nu:

Shared memory och synkronisering

Skicka buffrar mellan Compute Shader och grafikpipeline

Prestanda jämfört med CUDA/OpenCL

TO DO: DEMO MED SHARED MEMORY, DELVIS KLART



Kan du köra Compute Shaders?

Krav: OpenGL 4.3 + Kepler!

Inget svårt problem. 600-serien och uppåt.

I princip var som helst!



Tror vi på Compute Shaders?

- Portabelt mellan olika grafikkort och OS
- I princip samma funktionalitet som CUDA och OpenCL
 - Ingen separat installation



Projekt med Compute Shaders?

Varför inte - om du har ett problem stort nog.