Compute Shaders

Christian Hafner

Institute of Computer Graphics and Algorithms
Vienna University of Technology
Motivation I

- Use parallel processing power of GPU for **General Purpose** (GP) computations
- Great for
  - Image processing
  - Simulations
  - Equation Solvers
  - etc.
- Implement any SIMD algorithm!
  - **Single Instruction, Multiple Data**
Motivation II

- Why not OpenCL or CUDA?

- One API for graphics and GP processing
- Avoid interop
- Avoid context switches
- You already know GLSL
Availability

- Core since
  - OpenGL 4.3 (Aug 2012)
  - OpenGL ES 3.1 (Mar 2014)

- Supported on
  - Nvidia GeForce 400+
  - Nvidia Quadro x000, Kxxx
  - AMD Radeon HD 5000+
  - Intel HD Graphics 4600+
  - Adreno 400+
OpenGL 4.3 with Compute Shaders

Input
- Vertex Shader
- Tessellation Control Shader
- Tessellation Primitive Gen.
- Tessellation Eval. Shader
- Geometry Shader
- Transform Feedback
- Rasterization
- Fragment Shader
- Per-Fragment Operations

Output
- Dispatch Indirect Buffer b
- Image Load / Store t/b
- Atomic Counter b
- Shader Storage b
- Texture Fetch t/b
- Uniform Block b
- Pixel Assembly
- Pixel Operations
- Pixel Pack
- Pixel Unpack Buffer b
- Texture Image t
- Pixel Pack Buffer b

Legend
- Fixed Function Stage
- Programmable Stage
- b – Buffer Binding
- t – Texture Binding

Arrows indicate data flow

No Automatic Input
- Compute Shader

No Automatic Output
- Compute

Graphics Pipeline
Overview

- Thread Hierarchy
- Memory Resources
- Shared Memory & Synchronization
Smallest execution unit
Thread = 1 Invocation

```cpp
#version 430 core

layout(local_size_x=16,
      local_size_y=16,
      local_size_z=1) in;

layout(r32ui, binding=0) uniform uimage2D valueImage;

uniform uvec2 globalSize;

void main(void)
{
    uvec2 gid = gl_GlobalInvocationID.xy;
    if (gid.x < globalSize.x
        && gid.y < globalSize.y)
    {
        uint value = imageLoad(valueImage,
                                ivec2(gid)).x;
        uint newValue = value + 1;
        imageStore(valueImage,
                    ivec2(gid),
                    uvec4(newValue, 0, 0, 1));
    }
}
```
Thread Hierarchy II

- Grid of Threads = Work Group
- 1D, 2D or 3D
- Size
  - Specified in GLSL
  - Limited by GPU
    - Typically 1024 – 1536
- 2D example:

```c
layout(
    local_size_x = 5,
    local_size_y = 3,
    local_size_z = 1
) in;
```
Grid of work groups: Dispatch

1D, 2D or 3D

Size specified in OpenGL call

2D example:

```c
glDispatchCompute( 4 /* x */, 8 /* y */, 1 /* z */ );
```
Thread Location

- GLSL built-in variables
- Have type `uvec3`
- 2D example:
  - `gl_WorkGroupID` is \((2, 4, 0)\)
  - `gl_LocalInvocationID` is \((3, 0, 0)\)
  - `gl_GlobalInvocationID` is \((13, 12, 0)\)

(0,0)
Example (parallel image processing)

```glsl
#version 430 core

layout(local_size_x=16, local_size_y=16, local_size_z=1) in;

layout(r32ui, binding=0) uniform uimage2D valueImage;

uniform uvec2 globalSize;

void main(void)
{
    uvec2 gid = gl_GlobalInvocationID.xy;
    if (gid.x < globalSize.x && gid.y < globalSize.y)
    {
        uint value = imageLoad(valueImage, ivec2(gid)).x;
        uint newValue = value + 1;
        imageStore(valueImage, ivec2(gid), uvec4(newValue, 0, 0, 1));
    }
}
```

Every work group processes a 16x16 tile

Bounds check for unneeded threads

Example: 300x500 image
- 300 / 16 = 18.75
- 500 / 16 = 31.25
Need 19x32 work groups → 304x512 threads
Overview

- Thread Hierarchy
- Memory Resources
- Shared Memory & Synchronization
Memory Resources I

- No input from generic vertex attributes
  - `layout(location=0) in vec4 position;`

- No output to draw buffers / depth buffer
  - `layout(location=0) out vec4 outColor;`
**Memory Resources II**

- **Inputs**
  - Random access to
    - Images
    - SSBOs
  - Uniforms
  - Samplers

- **Outputs**
  - Random access to
    - Images
    - SSBOs

SSBO: Shader Storage Buffer Object
Images I

- An **image** is a single mipmap layer of a texture
  - No mipmapping
  - No bilinear filtering
  - No texture wrap modes

- Access in shader with pixel coordinates
- No support for 3-component images (e.g. rgb8; use rgba8 instead)
Images II

- In shader code
  - Define `image` variable
  - Access with `imageLoad`, `imageStore`

- In application
  - Bind a layer of the texture with `glBindImageTexture`(...)
Images III (Example)

```glsl
layout(binding=0, rgba8) 
uniform readonly image2D colImage;
layout(binding=1, r8ui)
uniform writeonly uimage2D bwImage;
...
vec4 val = imageLoad(colImage,
    ivec2(gl_GlobalInvocati

imageStore(bwImage,
     ivec2(gl_G

uvec4(mix(0,1,val.r > 0.5), 0,0,1));
```

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>image2D</td>
<td>floating point</td>
</tr>
<tr>
<td>iimage2D</td>
<td>signed integer</td>
</tr>
<tr>
<td>uimage2D</td>
<td>unsigned integer</td>
</tr>
</tbody>
</table>
SSBOs I

- **SSBO** = **Shader** **Storage** **Buffer** **Object**
  - Continuous chunk of GPU memory
  - Use for large data structures

- Structure of SSBO defined in GLSL
- Supports unsized arrays
- Read/Write like local variable
SSBOs II

Initialization in C++
- Buffer target is `GL_SHADER_STORAGE_BUFFER`
- Works like a VBO (`glGenBuffers`, `glBindBuffer`, `glBufferData ...`)

In shader code
- Define `buffer` block with data layout

Binding
- Bind with `glBindBufferBase(...)`
Example: Particles

GLSL

```cpp
struct PStruct {
    vec3 position;
    vec3 velocity;
    // .x = total life span, .y = life span left
    vec2 lifeSpan;
};

layout(std430, binding=0)
buffer Particles {
    PStruct particles[];
};
```
Example: Particles

```glsl
uniform uint numThreads;
uniform float dt;
uniform vec3 gravity;

void main(void) {
    uint gid = gl_GlobalInvocationID.x;
    if (gid < numThreads) {
        vec3 vel = particles[gid].velocity;
        vec3 pos = particles[gid].position;
        vel += dt * gravity;
        pos += vel * dt;
        particles[gid].velocity = vel;
        particles[gid].position = pos;
        particles[gid].lifeSpan.y -= dt;
    }
}
```

```glsl
struct PStruct {
    vec3 position;
    vec3 velocity;
    vec2 lifeSpan;
};

layout(std430, binding=0)
buffer Particles {
    PStruct particles[];
}
```
Example: Particles

- Upload initial particle data to SSBO
  - Create equivalent `struct` in C++
  - Fill with data in C++
  - Upload with `glBufferData` or `glBufferSubData`
Example: Particles

C++

```cpp
struct PStruct
{
    glm::vec3 position;
    glm::vec3 velocity;
    glm::vec2 lifeSpan;
};

PStruct* particles = new PStruct[100];
```
Example: Particles

GLSL

```glsl
struct PStruct {
    vec3  position;
    vec3  velocity;
    vec2  lifeSpan;
};

layout(std430, binding=0) buffer Particles {
    PStruct particles[];
};
```

Example:

```
particles[0].position = (0, 4, 8);
particles[0].velocity = (16, 20, 24);
particles[0].lifeSpan = (32, 36);
```

```
particles[1].position = (48, 52, 56);
particles[1].velocity = (64, 68, 72);
particles[1].lifeSpan = (80, 84, 88);
```

= 4 bytes
Example: Particles

### GLSL

```glsl
struct PStruct {
  vec3 position;
  vec3 velocity;
  vec2 lifeSpan;
};

layout(std430, binding=0)
buffer Particles {
  PStruct particles[];
};
```

### C++ (correct)

```c++
struct PStruct {
  glm::vec3 position;
  float padding0;
  glm::vec3 velocity;
  float padding1;
  glm::vec2 lifeSpan;
  glm::vec2 padding2;
};
```
When you match a C++ struct to an SSBO
- Look up the **data alignment rules**!
- Remember to add **padding**!

Can use SSBO as VBO to render!
- So we could calculate new positions
- And then use those positions as vertices

Rules for memory layout:
[OpenGL 4.5, Section 7.6.2.2, page 137](#)
Overview

- Thread Hierarchy
- Memory Resources
- Shared Memory & Synchronization
Memory

- Local memory
  - Per thread
  - Variables declared in `main()`
- Shared memory (SM)
  - Per work group
  - Declared before `main()` with `shared`
  - Compute shaders only
- Global memory
  - Textures, buffers, etc.

This is what makes compute shaders so special.
Use it why?
- SM has less delay than global memory

Use it when?
- If many threads require the same data from global memory
  - Read data from global memory once
  - Share it with other threads in SM
- If threads must communicate
Shared Memory: Example

- Horizontal Average Blur
- 7-pixel kernel

\[
\text{avg}(\quad )
\]
**Shared Memory: Example**

- **Naïve**: consider 1 thread independently

  1 thread
  → 7 *imageLoads*

- **Better**: consider 1 work group (e.g. 8 threads)

  1 work group = 8 threads
  → 14 *imageLoads*
Algorithm (per work group)

- Distribute the 14 `imageLoad` across 8 threads
- Write 14 samples to shared memory
- Synchronize!
- Every thread reads its 7 samples from shared memory
- Calculate average and `imageStore`
#version 430 core

layout(local_size_x=8,
      local_size_y=1,
      local_size_z=1) in;

layout(binding=0, rgba8) uniform readonly image2D inputImage;
layout(binding=1, rgba8) uniform writeonly image2D outputImage;

uniform ivec2 size;

shared vec4 sharedTexels[14];
void main(void)
{
    ivec2 gid = ivec2(gl_GlobalInvocationID.xy);
    uint lid = gl_LocalInvocationID.x;

    ivec2 coords =
        ivec2(clamp(gid.x-3,0,size.x-1), gid.y);
    sharedTexels[lid] = imageLoad(inputImage, coords);

    if (lid < 6)
    {
        coords.x = clamp(gid.x+5,0,size.x-1);
        sharedTexels[lid+8] =
            imageLoad(inputImage, coords);
    }
}
....

memoryBarrierShared();
barrier();

if (gid.x < size.x)
{
  vec4 finalColor = vec4(0,0,0,0);
  for (uint i=0;i<7;i++)
    finalColor += sharedTexels[lid+i];
  finalColor *= (1.0/7.0);

  imageStore(outputImage, gid, finalColor);
}

flush writes to shared memory
synchronize work group
Synchronization II

- Synchronization in GLSL is twofold
  - Invocation Control
    - `barrier()` – stalls execution until all threads in the work group have reached it
  - Memory Control
    - `memoryBarrier*()` – new values visible to all threads in dispatch
memoryBarrier()

- Makes writes visible to threads in the same dispatch

What about the next dispatch?
- Visibility not guaranteed

Call API function between dispatches
- glMemoryBarrier(GLbitfield mask);
- Various bits for operations like buffer access, image access, etc.
Example: image processing

// grayscale filter
// image0 -> image1

```c
glDispatchCompute(16,16,1);
glMemoryBarrier(
    GL_SHADER_IMAGE_ACCESS_BARRIER_BIT);
```

// gauss filter

// image1 -> image2

```c
glDispatchCompute(16,16,1);
```
Further Material I

- Atomic Counters

- Parallel Reduction on GPU (E.g. parallel scalar product)

- Warps and Memory Bank Conflicts
  - [https://www.youtube.com/watch?v=CZgM3DEBplE](https://www.youtube.com/watch?v=CZgM3DEBplE)
Further Material II

- Data alignment rules for $\text{std140} / \text{std430}$
  - OpenGL Specification 4.5, Section 7.6.2.2 (Standard Uniform Block Layout)
- Synchronization with $\text{glMemoryBarrier}$
  - OpenGL Specification 4.5, Section 7.12.2 (Shader Memory Access Synchronization)
- NVIDIA presentation about OpenGL 4.3 with Gauß Filter Compute Shader Code
  - [http://de.slideshare.net/Mark_Kilgard/siggraph-2012-nvidia-opengl-for-2012](http://de.slideshare.net/Mark_Kilgard/siggraph-2012-nvidia-opengl-for-2012)
Further Material III

- OpenGL Timer Queries – Measure your compute shader performance

- Everything about images (formats, atomics)
  - https://www.opengl.org/wiki/Image_Load_Store

- Buffer Textures
  - https://www.opengl.org/wiki/Buffer_Texture