Compute Shaders

Christian Hafner

Institute of Computer Graphics and Algorithms
Vienna University of Technology
Overview

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

- Use parallel processing power of GPU for General Purpose (GP) computations
- Great for image processing, particles, simulations, etc.
- Implement any parallel SPMD algorithm!
  - Single Program, 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)
- Part of OpenGL ES 3.1
- Supported on
  - Nvidia GeForce 400+
  - Nvidia Quadro x000, Kxxx
  - AMD Radeon HD 5000+
  - Intel HD Graphics 4600
OpenGL 4.3 with Compute Shaders

Old Pipeline

Compute Shaders

From Application
- Vertex Puller
- Vertex Shader
- Tessellation Control Shader
- Tessellation Primitive Gen.
- Tessellation Eval. Shader
- Geometry Shader
- Transform Feedback
- Transform Feedback Buffer
- Rasterization
- Fragment Shader
- Per-Fragment Operations
- Framebuffer

From Application
- Dispatch Indirect Buffer
- Dispatch
- Image Load / Store
- Atomic Counter
- Shader Storage
- Texture Fetch
- Uniform Block

From Application
- Pixel Assembly
- Pixel Operations
- Pixel Pack

Legend
- Fixed Function Stage
- Programmable Stage
- b – Buffer Binding
- t – Texture Binding
- Arrows indicate data flow

Christian Hafner
How to use it?

- Write compute shader in GLSL
  - Define memory resources
  - Write `main()` function
- Initialization
  - Allocate GPU memory (buffers, textures)
  - Compile shader, link program
- Run it
  - Bind buffers, textures, images, uniforms
  - Call `glDispatchCompute(...)`
Overview

- Introduction
- Thread Hierarchy
- Memory Resources
- Shared Memory & Synchronization
Smallest execution unit:
Thread = Invocation
Grid of Threads: Work Group

1D, 2D or 3D

Size specified in shader code

2D example:

```cpp
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
void dispatchCompute(
    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)
Limits on work group size per dimension:

```c
int dim = 0; /* 0=x, 1=y, 2=z */
int maxSizeX;
glGetIntegeri_v(
    GL_MAX_COMPUTE_WORK_GROUP_SIZE,
    dim, &maxSizeX);
```

Limit on total number of threads per work group:

```c
int maxInvoc;
glGetIntegeri(
    GL_MAX_COMPUTE_WORK_GROUP_INVOCATIONS,
    &maxInvoc);
```
Overview

- Introduction
- 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
Images I

- An **image** is a **single mipmap layer** of a texture
  - No mipmapping
  - No advanced sampling
  - Can have array layers
- Access in shader with integer pixel coordinates
- No support for 3-component images (e.g. *rgb8*; use *rgba8* instead)
In shader code

- Define `image` variable
- Access with `imageLoad`, `imageStore`, or atomic operations

Initialization in C++

- Allocate texture with corresponding format

Binding

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

```cpp
layout(binding=0, r8)
    uniform readonly image2D colImage;
layout(binding=1, r8ui)
    uniform writeonly uimage2D bwImage;
...
float val = imageLoad(colImage,
    ivec2(gl_GlobalInvocationID.xy)).r;
imageStore(bwImage,
    ivec2(gl_GlobalInvocationID.xy),
    uvec4(mix(0,1,val>0.5), 0,0,1));
```
SSBOs I

- SSBO = Shader Storage Buffer Object
  - Continuous, large chunk of GPU memory
  - Definition in shader similar to `struct` in C++
  - Supports unsized arrays
  - Random access + atomic operations
SSBOs II

- In shader code
  - Define `buffer` block with data layout
  - Access like local variable
- Initialization in C++
  - Buffer target is `GL_SHADER_STORAGE_BUFFER`
  - Upload initial contents in `glBufferData` (optional)
- Binding
  - Bind with `glBindBufferBase(...)`
SSBOs (Example I)

### GLSL

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

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

- Let’s upload initial data into the buffer

For a `GL_ARRAY_BUFFER` with vertex positions

- Provide pointer to `glm::vec3` array

For our particle buffer

- Declare `struct PStruct` in C++
- Fill array of structs with data and upload in `glBufferData`
```cpp
struct PStruct
{
    glm::vec3 position;
    glm::vec3 velocity;
    glm::vec2 lifeSpan;
};

PStruct* particles = new PStruct[100];
```
**SSBOs (Example IV)**

## GLSL

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

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

<table>
<thead>
<tr>
<th></th>
<th>particles[0].position</th>
<th>particles[0].velocity</th>
<th>particles[0].lifeSpan</th>
<th>particles[1].position</th>
<th>particles[1].velocity</th>
<th>particles[1].lifeSpan</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>16</td>
<td>32</td>
<td>48</td>
<td>64</td>
<td>80</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>20</td>
<td>36</td>
<td>52</td>
<td>68</td>
<td>84</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>24</td>
<td>40</td>
<td>56</td>
<td>72</td>
<td>88</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>28</td>
<td>44</td>
<td>60</td>
<td>76</td>
<td>92</td>
</tr>
</tbody>
</table>

= 4 bytes
**GLSL**

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

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

**C++**

```cpp
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**!
Overview

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

- Local memory
  - Per thread
  - Variables declared in `main()`

- Shared memory (SM)
  - Per work group
  - Declared globally 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
Example: 1D average filter with 7-pixel kernel

8 Threads per work group
Synchronization I

- Thread 31
  ```
  ivec2 coords = ivec2(clamp(gid.x-3, 0, size.x-1), gid.y);
  sharedTexels[lid] = imageLoad(inputImage, coords);
  if (lid < 6)
    {
      coords.x = clamp(coords.x + 64, 0, size.x-1);
      sharedTexels[lid+64] = imageLoad(inputImage, coords);
    }
  
  vec4 finalColor = vec4(0,0,0,0);
  for (int i=0; i<7; i++)
    finalColor += sharedTexels[lid+i];
  finalColor *= (1.0/7.0);

  imageStore(outputImage, gid, finalColor);
  ```

- Thread 33
  ```
  ivec2 coords = ivec2(clamp(gid.x-3, 0, size.x-1), gid.y);
  sharedTexels[lid] = imageLoad(inputImage, coords);
  if (lid < 6)
    {
      coords.x = clamp(coords.x + 64, 0, size.x-1);
      sharedTexels[lid+64] = imageLoad(inputImage, coords);
    }
  
  vec4 finalColor = vec4(0,0,0,0);
  for (int i=0; i<7; i++)
    finalColor += sharedTexels[lid+i];
  finalColor *= (1.0/7.0);

  imageStore(outputImage, gid, finalColor);
  ```

- Execution order between threads undefined
- What if Thread 33 is dependent on value written by Thread 31?
Synchronization in GLSL is twofold

- Invocation Control
- Memory Control
Invocation Control

- Control relative execution order of threads in the same work group
  - (No mechanism to control execution order across work groups)
- GLSL function `barrier()`
  - `barrier()` stalls execution until all threads in the work group have reached it
- However, this is not enough!
What happens when a thread calls `imageStore` or writes to shared memory?

- Momentarily: **nothing**
- At some **undefined** point in the future: the value is written

An OpenGL implementation has the freedom to **cache and delay**

- writes to SM
- random access writes to images, buffers
Memory Control II

- Memory Control
  - Flush all writes
  - Make new values visible to other threads
- `memoryBarrier*()`
  - New values visible to all threads in dispatch
- `groupMemoryBarrier()`
  - New values visible to all threads in work group
Memory Control III

- `memoryBarrier m()`

- \( m \in \{ \text{Shared}, \text{Buffer}, \text{Image}, \epsilon \} \)

  - Works only on buffers / images defined with keyword `coherent`
It seems we need

- `barrier()` for execution order
- `memoryBarrierShared()` to make SM writes visible

1. `barrier();`  
2. `memoryBarrierShared();`  
   `barrier();`

OR ?
Memory Control V

- `memoryBarrier()`
  - Values visible to threads in dispatch
- What about the next dispatch?
  - **Visibility not guaranteed**
- Call API function between dispatches
  - `glMemoryBarrier(GLbitfield mask);`
  - Various bits for different operations like buffer access, image access, etc.
Example: image processing

// grayscale filter
// image0 -> image1

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

```c
glMemoryBarrier(
    GL_SHADER_IMAGE_ACCESS_BARRIER_BIT);
```

// gauss filter

// image1 -> image2

```c
glDispatchCompute(16,16,1);
```
List of Stuff

- Buffer textures / Buffer images
- Atomic operations
  - On images, SSBOs, and SM
  - GL Core: only on integer variables
  - Atomic counters
- Warps and memory bank conflicts
- Further material on the last slides!

Christian Hafner
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=CZgM3DEBpIE
Data alignment rules for std140 / std430

- OpenGL Specification 4.5, Section 7.6.2.2 (Standard Uniform Block Layout)

Synchronization with 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
Further Material III

- OpenGL Timer Queries – Measure your compute shader performance

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

- Buffer Textures
  - [https://www.opengl.org/wiki/Buffer_Texture](https://www.opengl.org/wiki/Buffer_Texture)