Get the most out of the new OpenGL ES 3.1 API

Hans-Kristian Arntzen
Software Engineer

Content

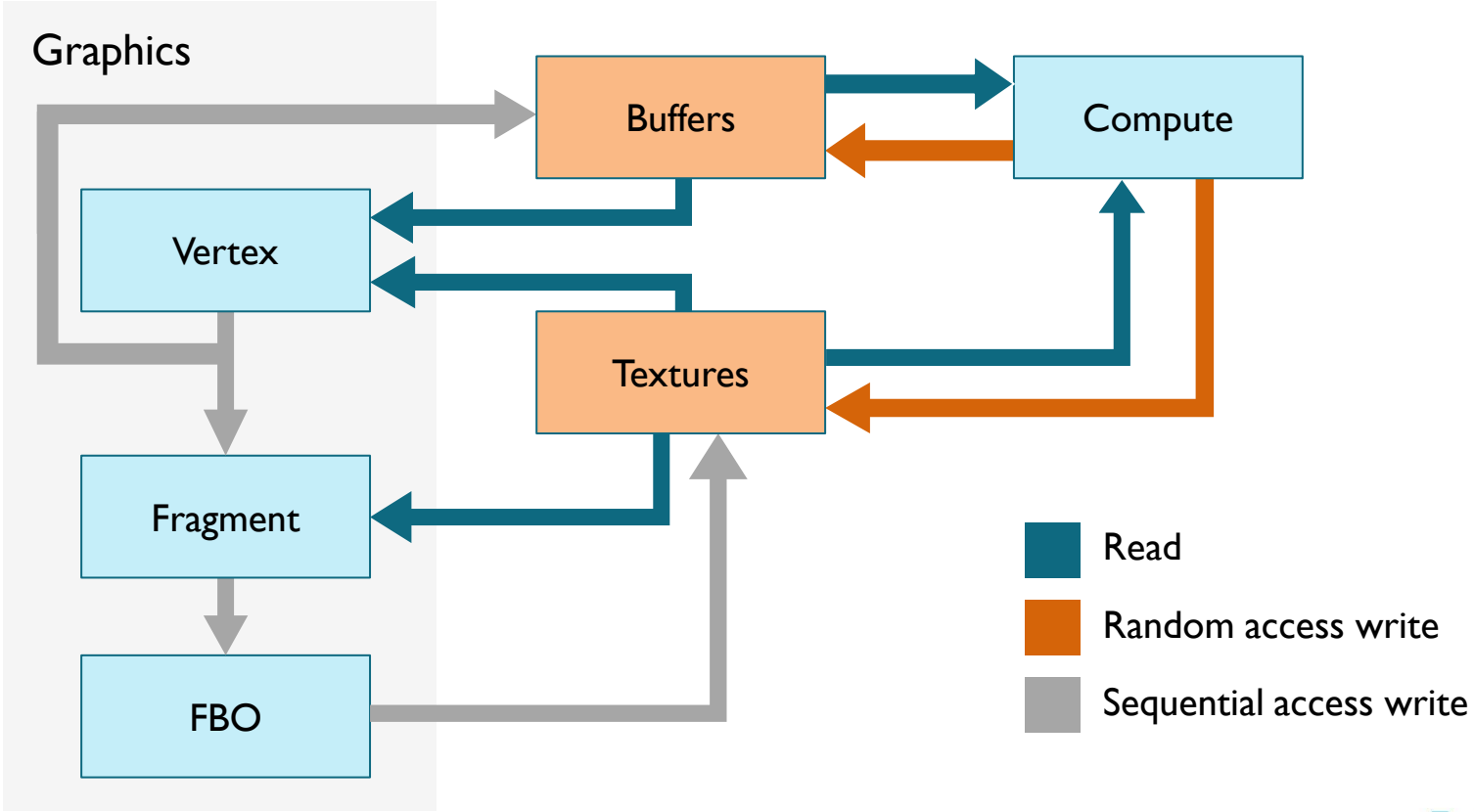
- Compute shaders introduction
  - Shader storage buffer objects
  - Shader image load/store
  - Shared memory
  - Atomics
  - Synchronization
- Indirect commands
- Best practices on ARM® Mali™ Midgard GPUs
- Use cases
- Example

Introduction to Compute Shaders

- Brings some OpenCL™ functionality to OpenGL ES
- Familiar Open GLSL syntax

- Random access writes to buffers and textures
- Sharing of data between threads

Introduction to Compute Shaders (cont.)

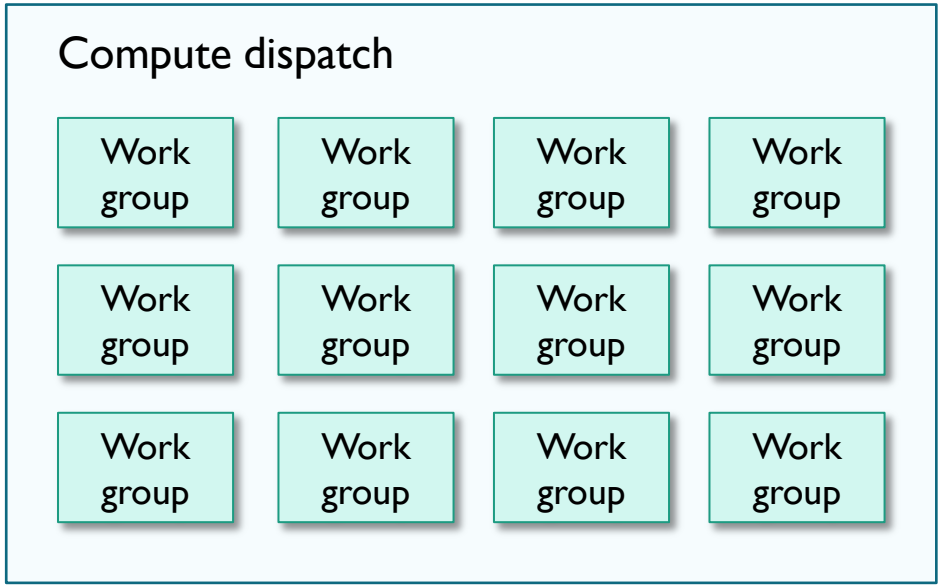


Compute model

- Traditional graphics pipeline
  - No random access write
  - Implicit parallelism
  - No synchronization between threads
- Compute
  - Random access writes to buffers and textures
  - Explicit parallelism
  - Full synchronization and sharing between threads in same **work group**

Compute model (cont.)

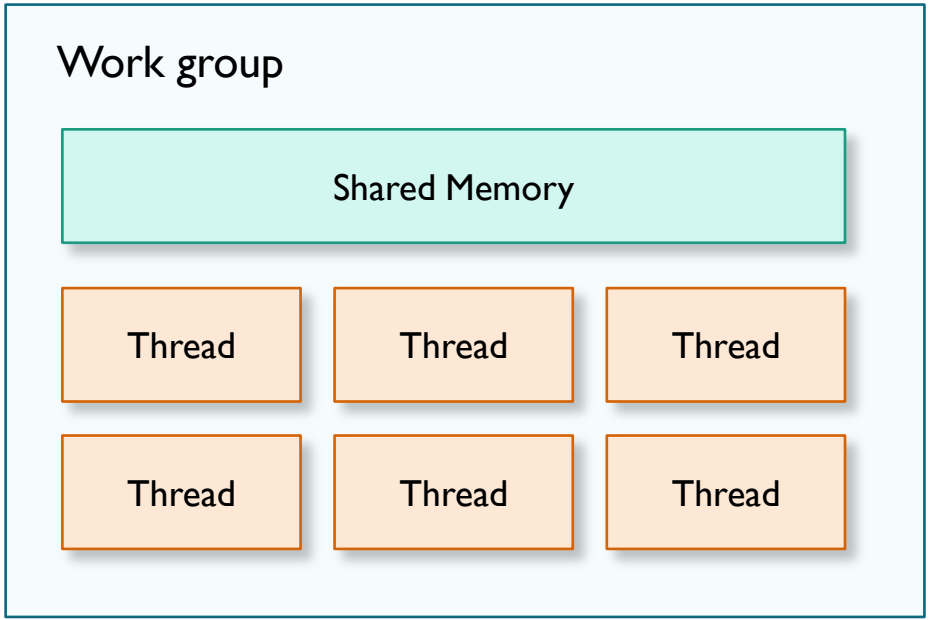
- Work group – the compute building block
  - Independent
  - Up to three dimensions



ARM

Compute model (cont.)

- Work group
  - Shared memory
  - Concurrent threads
  - Synchronization
- Unique identification
  - gl_LocalInvocation{ID,Index}
  - gl_GlobalInvocationID
  - gl_WorkGroupID



ARM

Hello compute world

```
#version 310 es
layout(local_size_x = 1) in;

layout(std430, binding = 0) buffer Output {
    writeonly float data[];
} output;

void main() {
    uint ident = gl_GlobalInvocationID.x;
    output.data[ident] = float(ident);
}
```


ARM

Compiling and executing a compute shader

```
GLuint shader = glCreateShader(GL_COMPUTE_SHADER);
// ... Compile, attach and link here.

glUseProgram(program);
glDispatchCompute(work_groups_x,
    work_groups_y,
    work_groups_z);
```

Shader storage buffer objects (SSBO)

- ”Writeable uniform buffers”
- Minimum required size **128 MiB**
- Can be unsized in shader
- New buffer layout for SSBOs (std430), better packing than std140

```
glBindBufferBase(GL_SHADER_STORAGE_BUFFER,
    binding, buffer_object);

layout(std430, binding = 0) buffer SomeData {
    float data[];
};
```

Shader image load/store

- Raw read/write texel access
- Layering support
- Atomics support in OES_shader_image_atomic

```
glBindImageTexture(0, tex, level, layered, layer,
    access, format);

layout(r32f, binding = 0) uniform writeonly image2D myImage;
imageStore(myImage, ivec2(x, y), color);
```

Shared memory

- Same as ”local” address space in OpenCL™
- Shared between threads in same work group
- Coherent
- Limited in size
  - GL_MAX_COMPUTE_SHARED_MEMORY_SIZE
  - Implementations must support at least 16 KiB

Atomic operations

- Dedicated atomic counters

```
glBindBufferBase(GL_ATOMIC_COUNTER_BUFFER, 0, atomic);

layout(binding = 0, offset = 0) uniform atomic_uint myCounter;
void main() {
    uint unique = atomicCounterIncrement(myCounter);
}
```

- SSBOs and shared memory
  - Add, Min/Max, Exchange, CompSwap, etc

```
shared uint sharedVarible;
uint previous = atomicMax(sharedVariable, 42u);
```

Memory qualifiers

- Applies to SSBOs and images
- coherent
  - Writes can be read by other shader invocations in the same command
  - Ensure visibility with shader language memory barrier
  - Writes only visible if they have **actually happened**
  - Shared memory implicitly declared coherent
- readonly / writeonly
- volatile / restrict
  - Same meaning as in C

Synchronization

- OpenGL ES synchronizes GL commands for you
  - Appears to operate as-if everything is in-order
  - Random access writes are unsynchronized
  - Ensure visibility to other GL commands with API memory barrier

Synchronization (cont.)

- glMemoryBarrier()
  - Ensures shader writes are visible to subsequent GL calls
  - Specify how data is **read** after the barrier

```
glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 0, vbo);
glDispatchCompute(groups_x, groups_y, groups_z);

// Non-blocking call! Flush caches on GPU, etc.
glMemoryBarrier(GL_VERTEX_ATTRIB_ARRAY_BARRIER_BIT);

// Draw using updated VBO contents.
glDrawElements(...);
```

Synchronization (cont.)

- groupMemoryBarrier(), memoryBarrier*()
  - Ensures coherent writes are visible to other shader invocations
  - Writes below barrier not visible before writes above barrier

- barrier()
  - All threads in work group must reach barrier before any thread can continue
  - Must be called from **dynamically uniform control flow**
  - **Does not order memory**
  - memoryBarrierShared() before barrier()

Synchronization (cont.)

```
#version 310 es
layout(local_size_x = 128) in;
shared float sharedData[128];

void main() {
  sharedData[gl_LocalInvocationIndex] = 0.0;

  // Ensure shared memory writes are visible to work group
  memoryBarrierShared();

  // Ensure all threads in work group
  // have executed statements above
  barrier();

  // Entire buffer now cleared for every thread
}
```

Indirect commands

- Three new indirect commands
  - glDrawArraysIndirect
  - glDrawElementsIndirect
  - glDispatchComputeIndirect
- Draw/dispatch parameters sourced from buffer object
- Lets GPU feed itself with work
  - Very useful when draw parameters are not known by CPU
  - Avoids CPU/GPU synchronization point

Indirect commands (cont.)

```
struct IndirectCommand {
    GLuint count;
    GLuint instanceCount;
    GLuint firstIndex;
    GLuint baseVertex;
    GLuint reservedMustBeZero;
};

glBindBuffer(GL_DRAW_INDIRECT_BUFFER, command);
// Update instanceCount on GPU.
glDrawElementsIndirect(GL_TRIANGLES,
    GL_UNSIGNED_SHORT, NULL);
```

Indirect commands (cont.)

```
struct IndirectDispatch {
    GLuint num_groups_x;
    GLuint num_groups_y;
    GLuint num_groups_z;
};

glBindBuffer(GL_DISPATCH_INDIRECT_BUFFER, command);
// Update dispatch buffer on GPU.
glDispatchComputeIndirect(0);
```

Best practices on ARM® Mali™ Midgard

- Global memory just as fast as shared memory
  - Avoid reads from global to shared memory just for caching
  - Use shared if sharing of computation is needed
- Atomics on SSBOs just as efficient as atomic counters
  - SSBOs cleaner anyways
- Cheap branching
  - Branch on gl_LocalInvocationIndex == 0u for expensive once-per-work-group code paths

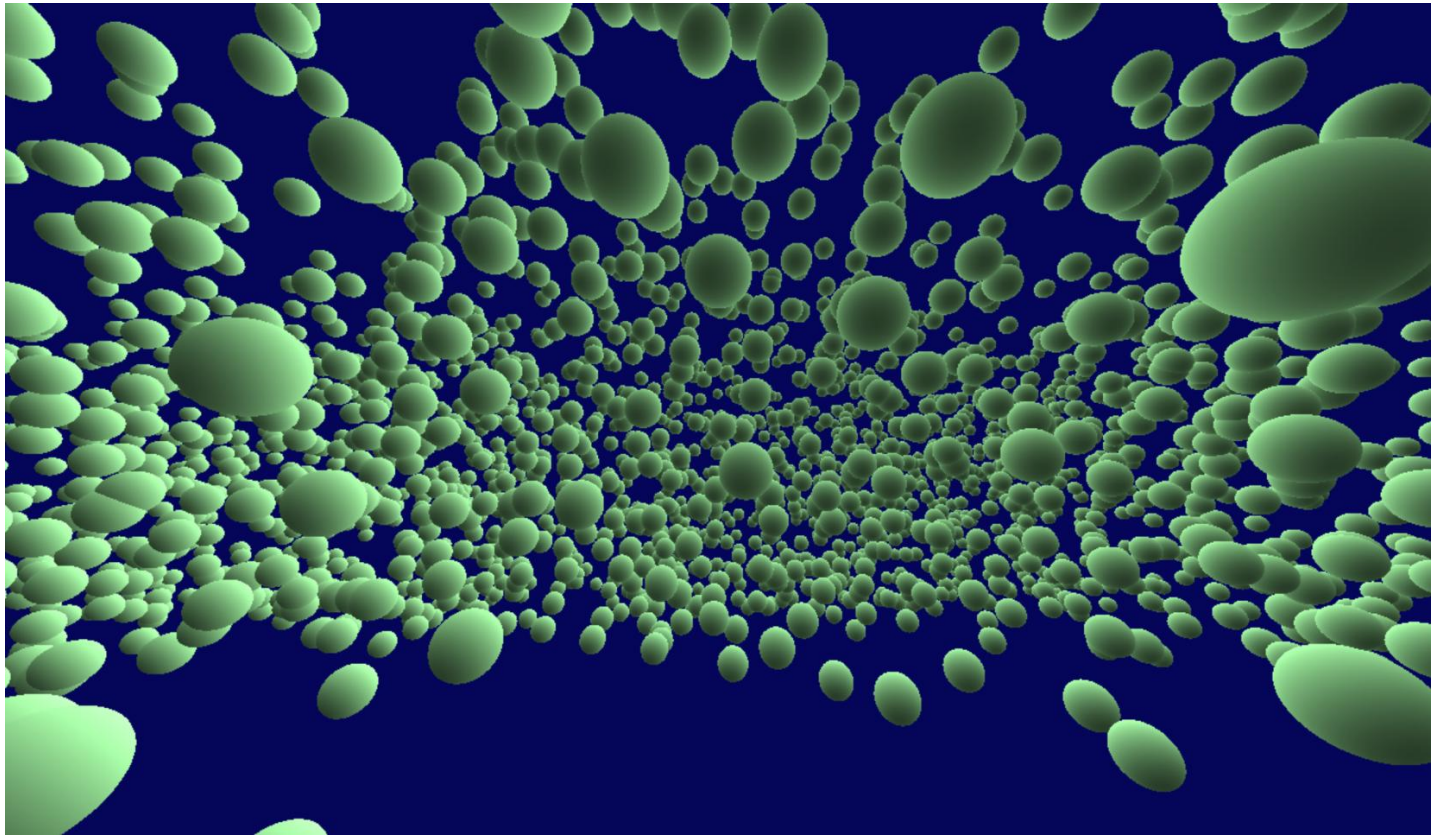
Best practices on ARM® Mali™ Midgard (cont.)

- Many small indirect draws are expensive
  - instanceCount of 0 often just as expensive as 1
  - Use sparingly
  - Ideal case is instancing
- Avoid tiny work groups
  - Limits maximum number of concurrent threads
  - Work group of 128 threads recommended

Use cases for compute

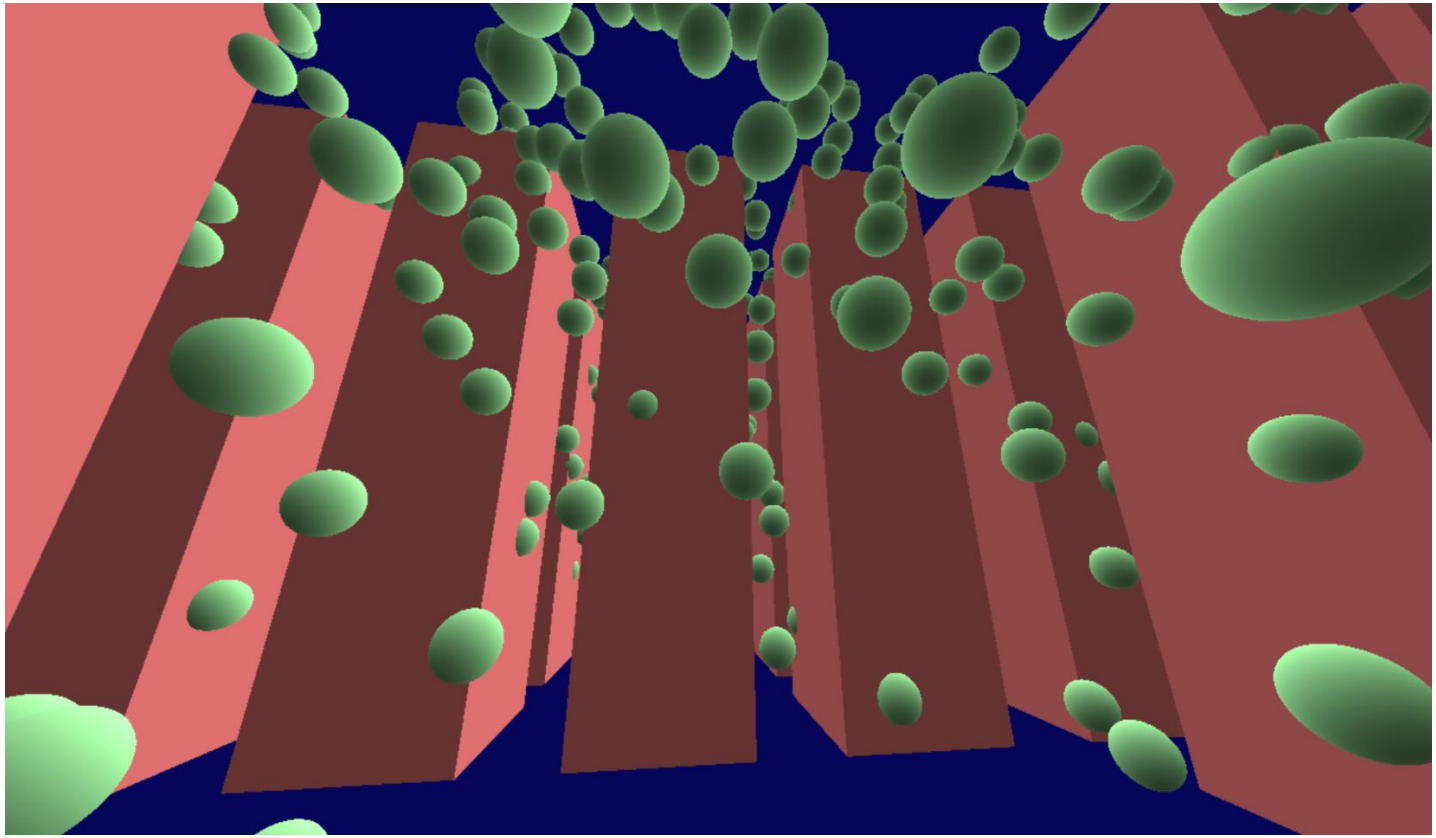
- Wave simulation
- **Occlusion culling**
- Physics
- Particle effects
- Image processing

Occlusion culling with compute



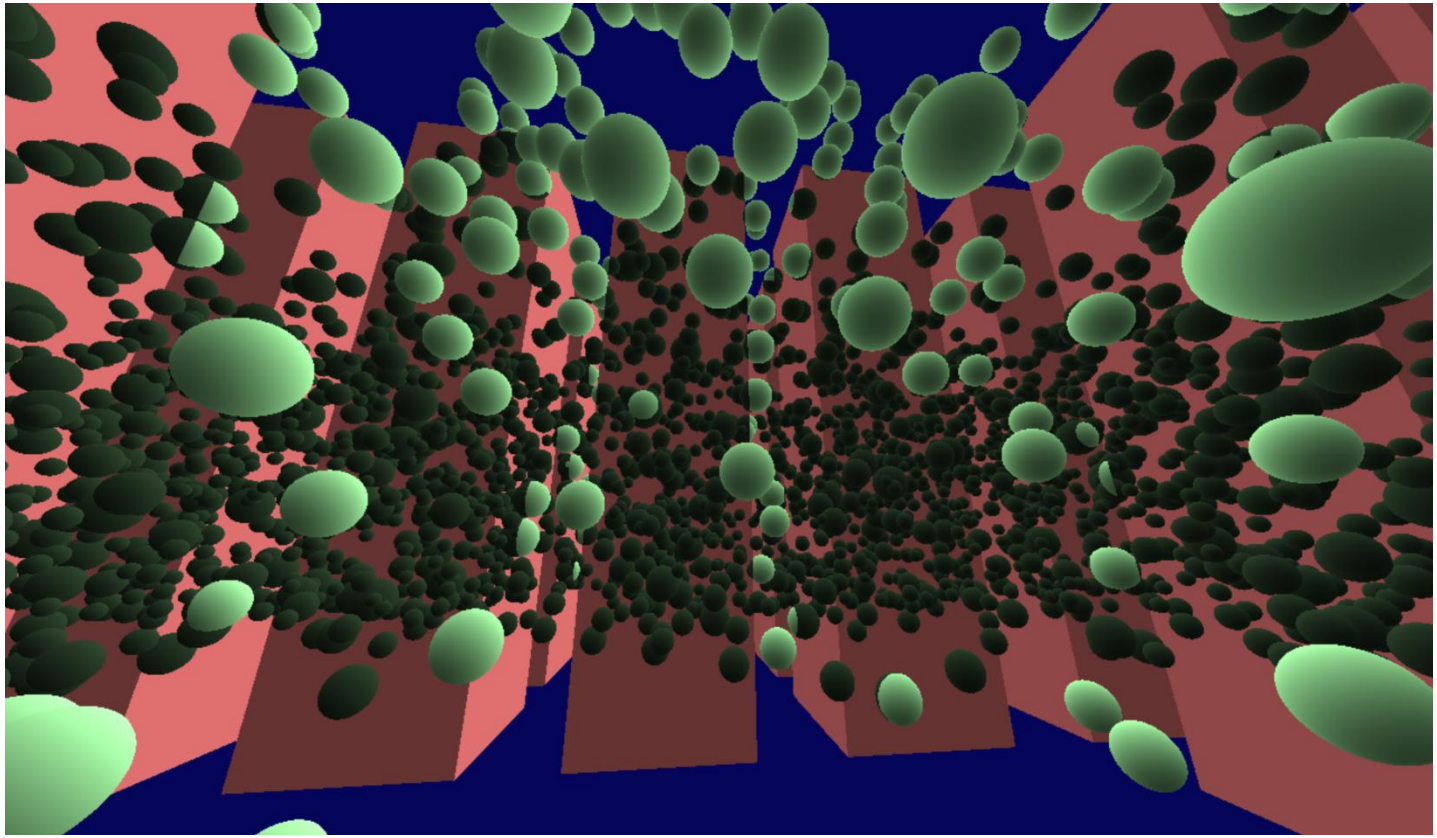
ARM

Occlusion culling with compute (cont.)



ARM

Occlusion culling with compute (cont.)



ARM

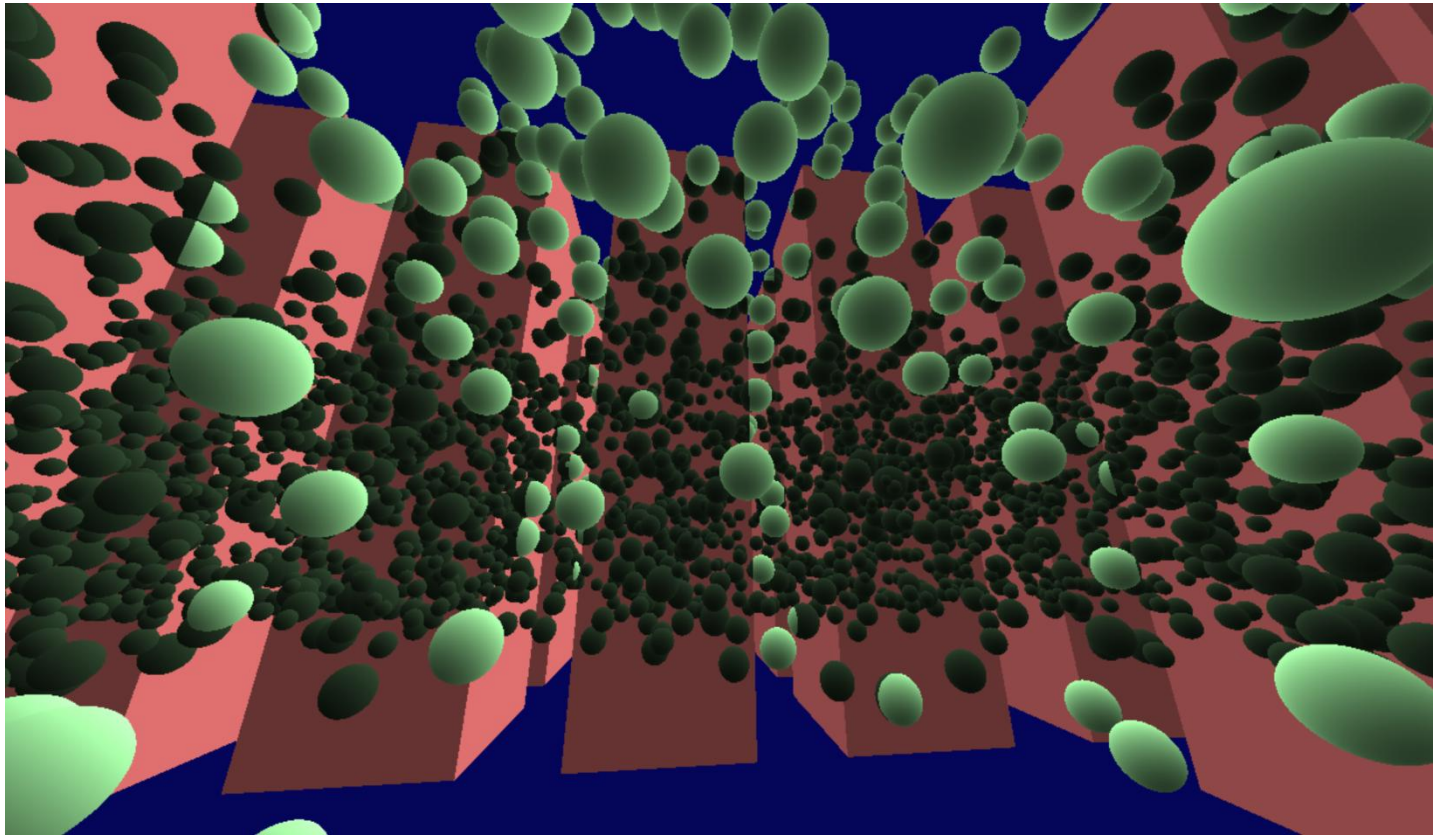
Occlusion culling with compute (cont.)

- Only want to draw visible meshes
- Frustum culling not enough
- OpenGL ES 3.0 occlusion query too inefficient
  - Doesn't support instancing
  - CPU readback required
- Traditional CPU methods not always viable
  - Instance data updated every frame by GPU
  - CPU already busy with other tasks

Hierarchical-Z occlusion culling

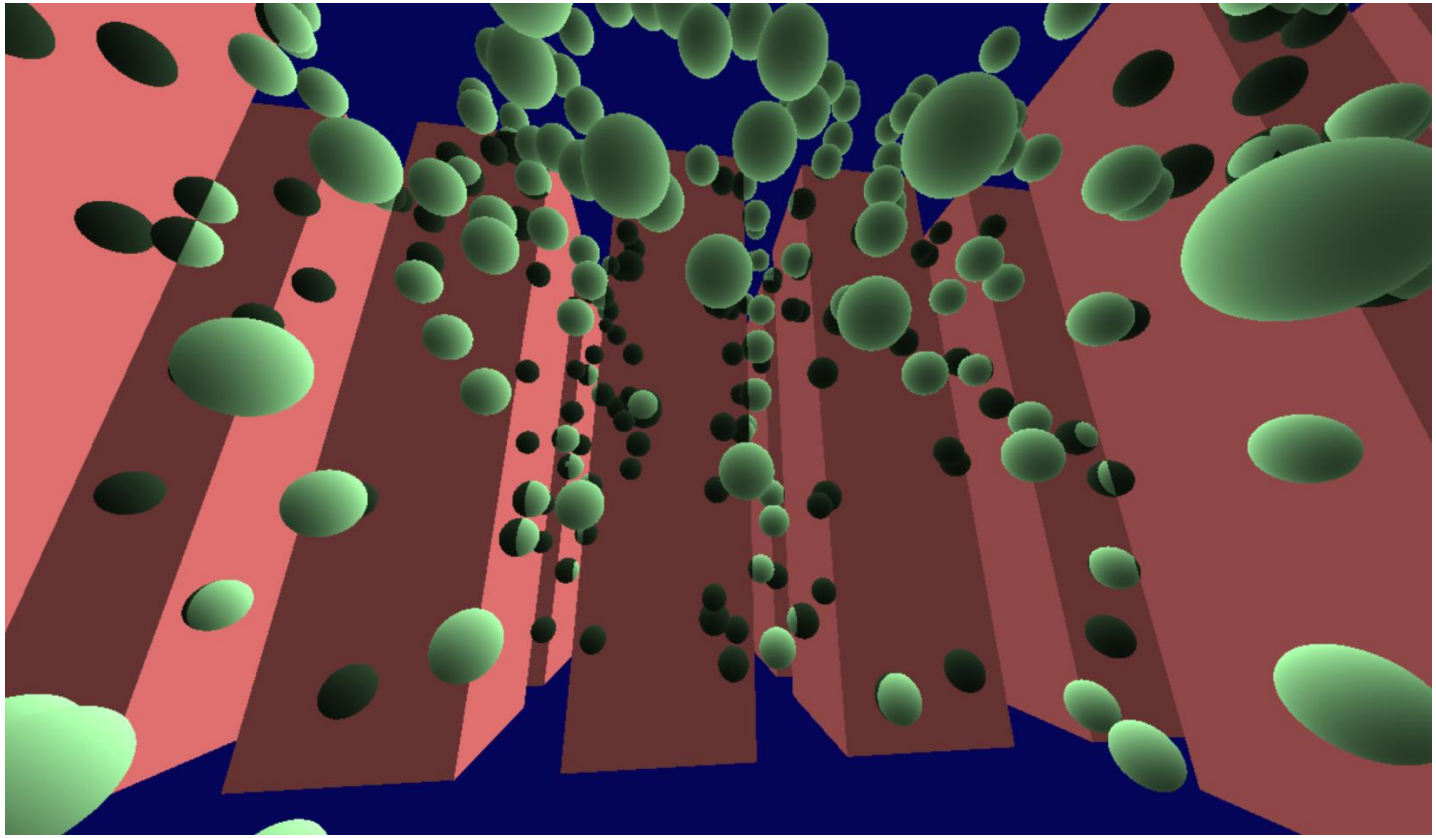
- Rasterize occluders to depth map
  - Simplified occluder meshes
  - Reduced resolution good enough (256x256)
  - Mipmap depth manually with max() filter
- Test every bounding volume in parallel with compute
  - Find screen space bounding box
  - Sample depth at appropriate LOD
- Append visible per-instance data to buffer
  - Atomic counter to increment instanceCount
- Indirect Draw

Results (without culling)



ARM

Results (with culling)



ARM

Performance, Mali™-T604

- Vertex bound
- ~100 vertices per sphere
- ~114k spheres in scene

Method	Frame time (ms)	Culling time (ms)	Vertices (per frame)
Hi-Z culling	**10.8**	**3.3**	**186k**
Frustum culling	120.5	1.3	2837k
No culling	246.9	N/A	11271k

ARM

Closing

- Compute shaders is a very useful addition to OpenGL ES
- Indirect features allow GPU to feed itself with work