Advanced Metal Shader Optimization
Forging and polishing your Metal shaders
Session 606

Fiona Assembly Alchemist
Alex Kan GPU Software
Metal at WWDC This Year
A look at the sessions

Adopting Metal

Part One
• Fundamental Concepts
• Basic Drawing
• Lighting and Texturing

Part Two
• Dynamic Data Management
• CPU-GPU Synchronization
• Multithreaded Encoding
Metal at WWDC This Year
A look at the sessions

What’s New in Metal

Part One
• Tessellation
• Resource Heaps and Memoryless Render Targets
• Improved Tools

Part Two
• Function Specialization and Function Resource Read-Writes
• Wide Color and Texture Assets
• Additions to Metal Performance Shaders
Metal at WWDC This Year

A look at the sessions

Advanced Shader Optimization

- Shader Performance Fundamentals
- Tuning Shader Code
Optimizing Shaders
An overview

There’s a lot you can do to make your code faster
Including things specific to A8 and later GPUs!
And major performance pitfalls to watch for…
Do high-level optimizations before low-level
For experienced shader authors
Metal Pipeline

Flowchart:
- Buffer
- Texture
- Sampler
- Function

Buffer -> Vertex Fetch -> Vertex Processing -> Rasterization -> Fragment Processing -> Framebuffer Write

- Color Attachments
- Depth Attachment

Present
Metal Pipeline

- Vertex Fetch
- Buffer
- Texture
- Sampler
- Function
- Vertex Processing
- Rasterization
- Fragment Processing
- Framebuffer Write
- Color Attachments
- Depth Attachment
- Present
Overview

Shader performance fundamentals
Tuning shader code
Shader Performance Fundamentals
Shader Performance Fundamentals
Things to check before digging deeper

Address space selection for buffer arguments
Buffer preloading
Fragment function resource writes
Compute kernel organization
GPUs have multiple paths to memory
Designed for different access patterns
Explicitly developer-controlled in shading language
Address Spaces
Device memory

Read-write
No size restrictions
Flexible alignment restrictions
Address Spaces
Constant memory

- Read-only
- Limited size
- Alignment restrictions
- Optimized for reuse
Address Spaces

Picking an address space

1. Start
2. How much data? (variable size)
3. How many times will each item be read? (few, many)
4. Device
5. Constant
Address Spaces

Example: vertex data

<table>
<thead>
<tr>
<th>Variable</th>
<th>Number of items</th>
<th>Amount of reuse</th>
<th>Address space</th>
</tr>
</thead>
<tbody>
<tr>
<td>positions</td>
<td>variable number of vertices</td>
<td>one</td>
<td>device</td>
</tr>
</tbody>
</table>

```
vertex float4 simpleVertex(uint vid [[ vertex_id ]]),
    const device float4 *positions [[ buffer(0) ]])
{
    return positions[vid];
}
```
vertex float4 simpleVertex(uint vid [[ vertex_id ]]),

```
const device float4 *positions [[ buffer(0) ]])
{
    return positions[vid];
}
```
## Address Spaces

**Example: projection matrix**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Number of items</th>
<th>Amount of reuse</th>
<th>Address space</th>
</tr>
</thead>
<tbody>
<tr>
<td>transform</td>
<td>one</td>
<td>all</td>
<td>constant</td>
</tr>
</tbody>
</table>

```cpp
vertex float4 transformedVertex(uint vid [[ vertex_id ]]),
    const device float4 *positions [[ buffer(0) ]],
    constant matrix_float4x4 &transform [[ buffer(1) ]])
{
    return transform * positions[vid];
}
```
Address Spaces

Example: projection matrix

<table>
<thead>
<tr>
<th>Variable</th>
<th>Number of items</th>
<th>Amount of reuse</th>
<th>Address space</th>
</tr>
</thead>
<tbody>
<tr>
<td>transform</td>
<td>one</td>
<td>all</td>
<td>constant</td>
</tr>
</tbody>
</table>

```c
vertex float4 transformedVertex(uint vid [[ vertex_id ]]),
    const device float4 *positions [[ buffer(0) ]],
    constant matrix_float4x4 &transform [[ buffer(1) ]])
{
    return transform * positions[vid];
}
```
Example: skinning matrices

<table>
<thead>
<tr>
<th>Variable</th>
<th>Number of items</th>
<th>Amount of reuse</th>
<th>Address space</th>
</tr>
</thead>
<tbody>
<tr>
<td>skinningMatrices</td>
<td>fixed number of bones</td>
<td>all vertices using bone</td>
<td>constant</td>
</tr>
</tbody>
</table>

```
struct SkinningMatrices {
    matrix_float4x4 position_transforms[MAXBONES];
};

vertex float4 skinnedVertex(uint vid [[ vertex_id ]]),
    const device Vertex *vertices [[ buffer(0) ]],
    constant SkinningMatrices &skinningMatrices [[ buffer(1) ]]
{
    ...
    for (ushort i = 0; i < NBONES; ++i) {
        skinnedPosition += (skinningMatrices.position_transforms[vertices[vid].boneIndices[i]] * 
    ...
}
```
Address Spaces

Example: skinning matrices

<table>
<thead>
<tr>
<th>Variable</th>
<th>Number of items</th>
<th>Amount of reuse</th>
<th>Address space</th>
</tr>
</thead>
<tbody>
<tr>
<td>skinningMatrices</td>
<td>fixed number of bones</td>
<td>all vertices using bone</td>
<td>constant</td>
</tr>
</tbody>
</table>

```cpp
struct SkinningMatrices {
    matrix_float4x4  position_transforms[MAXBONES];
};

vertex float4 skinnedVertex(uint vid [[ vertex_id ]]),
    const device Vertex *vertices [[ buffer(0) ]],
    constant SkinningMatrices &skinningMatrices [[ buffer(1) ]]
{
    ...
    for (ushort i = 0; i < NBONES; ++i) {
        skinnedPosition += (skinningMatrices.position_transforms[vertices[vid].boneIndices[i]] * ...
    }
}
**Address Spaces**

**Example: per-instance data**

<table>
<thead>
<tr>
<th>Use case</th>
<th>Number of items</th>
<th>Amount of reuse</th>
<th>Address space</th>
</tr>
</thead>
<tbody>
<tr>
<td>instanceTransforms</td>
<td>variable number of instances</td>
<td>all vertices in instance</td>
<td>device</td>
</tr>
</tbody>
</table>

```c
vertex float4 instancedVertex(uint vid [[ vertex_id ]],
                               uint iid [[ instance_id]],
                               const device float4 *positions [[ buffer(0) ]],
                               const device matrix_float4x4 *instanceTransforms [[ buffer(1) ]])
{
    return instanceTransforms[iid] * positions[vid];
}
```
### Address Spaces

**Example: per-instance data**

<table>
<thead>
<tr>
<th>Use case</th>
<th>Number of items</th>
<th>Amount of reuse</th>
<th>Address space</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>instanceTransforms</code></td>
<td>variable number of instances</td>
<td>all vertices in instance</td>
<td>device</td>
</tr>
</tbody>
</table>

```cpp
vertex float4 instancedVertex(uint vid [[ vertex_id ]],
                            uint iid [[ instance_id]],
                            const device float4 *positions [[ buffer(0) ]],
                            const device matrix_float4x4 *instanceTransforms [[ buffer(1) ]])
{
    return instanceTransforms[iid] * positions[vid];
}
```
Buffer Preloading

Buffer loads can be hoisted to dedicated hardware

- Constant buffers
- Vertex buffers

Depending on

- Access patterns in the shader
- Address space buffer resides in
Constant Buffer Preloading

Direct loads
- Known address/offset
- No indexing

Indirect loads
- Unknown address/offset
- Buffer must be explicitly sized
Constant Buffer Preloading

Direct loads
• Known address/offset
• No indexing

Indirect loads
• Unknown address/offset
• Buffer must be explicitly sized
Constant Buffer Preloading

Use constant address space when appropriate

Statically bound your accesses
• Pass single struct arguments by reference
• Pass bounded arrays in a struct, rather than via a pointer

```c
typedef struct {
    uint count;
    Light data[MAX_LIGHTS];
} LightData;

fragment float4 litFragment(
    const device Light *l [[ buffer(0) ]],
    const device uint *count [[ buffer(1) ]],
    LitVertex vertex [[ stage_in ]]);
```

```c
fragment float4 litFragment(
    constant LightData &lights [[ buffer(0) ]],
    LitVertex vertex [[ stage_in ]]);
```
Constant Buffer Preloading

Use constant address space when appropriate

Statically bound your accesses

- Pass single struct arguments by reference
- Pass bounded arrays in a struct, rather than via a pointer

```c
typedef struct {
    uint count;
    Light data[MAX_LIGHTS];
} LightData;

fragment float4 litFragment(
    constant LightData &lights [[ buffer(0) ]],
    LitVertex vertex [[ stage_in ]]);
```

```c
fragment float4 litFragment(
    const device Light *l [[ buffer(0) ]],
    const device uint *count [[ buffer(1) ]],
    LitVertex vertex [[ stage_in ]]);
```
Constant Buffer Preloading

Use constant address space when appropriate

Statically bound your accesses
- Pass single struct arguments by reference
- Pass bounded arrays in a struct, rather than via a pointer

```c
typedef struct {
    uint count;
    Light data[MAX_LIGHTS];
} LightData;

fragment float4 litFragment(
    constant LightData &lights [[ buffer(0) ]],
    LitVertex vertex [[ stage_in ]]);

fragment float4 litFragment(
    const device Light *l [[ buffer(0) ]],
    const device uint *count [[ buffer(1) ]],
    LitVertex vertex [[ stage_in ]]);
```
Constant Buffer Preloading

Use constant address space when appropriate

Statically bound your accesses

- Pass single struct arguments by reference
- Pass bounded arrays in a struct, rather than via a pointer

```c
typedef struct {
    uint count;
    Light data[MAX_LIGHTS];
} LightData;

fragment float4 litFragment(
    constant LightData &lights [[ buffer(0) ]],
    LitVertex vertex [[ stage_in ]]);
```
Constant Buffer Preloading

Use constant address space when appropriate

Statically bound your accesses

- Pass single struct arguments by reference
- Pass bounded arrays in a struct, rather than via a pointer

```c
typedef struct {
    uint count;
    Light data[MAX_LIGHTS];
} LightData;

fragment float4 litFragment(
    constant LightData &lights [[ buffer(0) ]],
    LitVertex vertex [[ stage_in ]]);
```
Constant Buffer Preloading

Use constant address space when appropriate

Statically bound your accesses

• Pass single struct arguments by reference
• Pass bounded arrays in a struct, rather than via a pointer

```
typedef struct {
    uint count;
    Light data[MAX_LIGHTS];
} LightData;

fragment float4 litFragment(  
    constant LightData &lights [[ buffer(0) ]],  
    LitVertex vertex [[ stage_in ]]);
```
Constant Buffer Preloading

A practical example: deferred rendering

More than one way to implement a deferred renderer

Not all ways created equal from a performance point of view
Constant Buffer Preloading

A practical example: deferred rendering

One draw call for all lights

• May read all lights
• Unbounded input size

```glsl
fragment float4 accumulateAllLights(
    const device Light *allLights [[ buffer(0) ]],
    LightInfo tileLightInfo [[ stage_in ]]);
```
Constant Buffer Preloading
A practical example: deferred rendering

One draw call for all lights
- May read all lights
- Unbounded input size

```glsl
fragment float4 accumulateAllLights(
    const device Light *allLights [[ buffer(0) ]],
    LightInfo tileLightInfo [[ stage_in ]]);
```
Constant Buffer Preloading
A practical example: deferred rendering

One draw call per light

• Bounded input size — can be in constant address space
• Takes advantage of constant buffer preloading

```glsl
fragment float4 accumulateAllLights(
    const device Light *allLights [[ buffer(0) ]],
    LightInfo tileLightInfo [[ stage_in ]]);
```
Constant Buffer Preloading
A practical example: deferred rendering

One draw call per light
- Bounded input size — can be in constant address space
- Takes advantage of constant buffer preloading

```cpp
fragment float4 accumulateAllLights(
    const device Light *allLights [[ buffer(0) ]],
    LightInfo tileLightInfo [[ stage_in ]]);
```

```cpp
fragment float4 accumulateOneLight(
    constant Light &currentLight [[ buffer(0) ]],
    LightInfo lightInfo [[ stage_in ]]);
```
Constant Buffer Preloading
A practical example: deferred rendering

One draw call per light

• Bounded input size — can be in constant address space
• Takes advantage of constant buffer preloading

```c
fragment float4 accumulateAllLights(
    const device Light *allLights [[ buffer(0) ]],
    LightInfo tileLightInfo [[ stage_in ]]);
```

```c
fragment float4 accumulateOneLight(
    constant Light &currentLight [[ buffer(0) ]],
    LightInfo lightInfo [[ stage_in ]]);
```
Constant Buffer Preloading
A practical example: deferred rendering

One draw call per light
• Bounded input size — can be in constant address space
• Takes advantage of constant buffer preloading

```glsl
fragment float4 accumulateAllLights(
    const device Light *allLights [[ buffer(0) ]],
    LightInfo tileLightInfo [[ stage_in ]]);
```

```glsl
fragment float4 accumulateOneLight(
    constant Light &currentLight [[ buffer(0) ]],
    LightInfo lightInfo [[ stage_in ]]);
```
Vertex Buffer Preloading

Fixed-function vertex fetching is handled by dedicated hardware Buffer loads will be handled by dedicated hardware for buffer loads if:

- Indexed by vertex/instance ID
- Including divisor math
- With or without base vertex/instance offset
Vertex Buffer Preloading

Use vertex descriptors where possible

If you’re writing your own indexing code

• Lay out data linearly to simplify buffer indexing
• Lower-granularity data can still be hoisted if access is linear
Fragment Function Resource Writes

Resource writes in fragment shaders partially defeat hidden surface removal

- Can’t be occluded by later fragments
- Can be removed by failing depth/stencil test with 

[[ early_fragment_tests ]]

NEW
Fragment Function Resource Writes

Use `[[ early_fragment_tests ]]` to maximize rejection

- Draw after opaque objects
- Sort front-to-back if updating depth/stencil

Similar to objects with discard/per-pixel depth
Compute Kernel Organization

Per-thread launch overhead
Barriers
Compute Kernel Organization

Amortizing compute thread launch overhead

Process multiple work items per compute thread

Reuse values across work items
kernel void sobel_1_1(/* ... */
  ushort2 tid [[ thread_position_in_grid ]])
{
  ushort2 gid = ushort2(tid.x,tid.y);
  ushort2 dstCoord = ...
  ...

  // read 3x3 region of source
  float2 c = ...
  float r0 = src.sample(sam, c, int2(-1,-1)).x;
  // read r1-r8

  // apply Sobel filter
  float gx = (r2-r0) + 2.0f*(r5-r3) + (r8-r6);
  float gy = (r0-r6) + 2.0f*(r1-r7) + (r2-r8);
  float4 g = float4(sqrt(gx * gx + gy * gy));
  dst.write(g, static_cast<uint2>(dstCoord));
}
kernel void sobel_1_1(/* ... */
    ushort2 tid [[ thread_position_in_grid ]])
{
    ushort2 gid = ushort2(tid.x,tid.y);
    ushort2 dstCoord = ...
    ...

    // read 3x3 region of source
    float2 c = ...
    float r0 = src.sample(sam, c, int2(-1,-1)).x;
    // read r1-r8

    // apply Sobel filter
    float gx = (r2-r0) + 2.0f*(r5-r3) + (r8-r6);
    float gy = (r0-r6) + 2.0f*(r1-r7) + (r2-r8);
    float4 g = float4(sqrt(gx * gx + gy * gy));
    dst.write(g, static_cast<uint2>(dstCoord));
}
kernel void sobel_1_1(/* ... */
    ushort2 tid [[ thread_position_in_grid ]])
{
    ushort2 gid = ushort2(tid.x,tid.y);
    ushort2 dstCoord = ...
    ...

    // read 3x3 region of source
    float2 c = ...
    float r0 = src.sample(sam, c, int2(-1,-1)).x;
    // read r1-r8

    // apply Sobel filter
    float gx = (r2-r0) + 2.0f*(r5-r3) + (r8-r6);
    float gy = (r0-r6) + 2.0f*(r1-r7) + (r2-r8);
    float4 g = float4(sqrt(gx * gx + gy * gy));
    dst.write(g, static_cast<uint2>(dstCoord));
}
kernel void sobel_1_1(/* ... */
    ushort2 tid [[ thread_position_in_grid ]])
{
    ushort2 gid = ushort2(tid.x, tid.y);
    ushort2 dstCoord = ...
    ...

    // read 3x3 region of source
    float2 c = ...
    float r0 = src.sample(sam, c, int2(-1,-1)).x;
    // read r1-r8

    // apply Sobel filter
    float gx = (r2-r0) + 2.0f*(r5-r3) + (r8-r6);
    float gy = (r0-r6) + 2.0f*(r1-r7) + (r2-r8);
    float4 g = float4(sqrt(gx * gx + gy * gy));
    dst.write(g, static_cast<uint2>(dstCoord));
}
kernel void sobel_1_1(/* ... */
    ushort2 tid [[ thread_position_in_grid ]])
{
    ushort2 gid = ushort2(tid.x*2,tid.y);
    ushort2 dstCoord = ...
    ...

    // read 3x3 region of source for pixel 1
    float2 c = ...
    float r0 = src.sample(sam, c, int2(-1,-1)).x;
    // read r1-r8

    // apply Sobel filter for pixel 1
    float gx = (r2-r0) + 2.0f*(r5-r3) + (r8-r6);
    float gy = (r0-r6) + 2.0f*(r1-r7) + (r2-r8);
    float4 g = float4(sqrt(gx * gx + gy * gy));
    dst.write(g, static_cast<uint2>(dstCoord));

    // continue to pixel 2
kernel void sobel_1_1(/* ... */
    ushort2 tid [[ thread_position_in_grid ]])
{
    ushort2 gid = ushort2(tid.x*2,tid.y);
    ushort2 dstCoord = ...
    ...

    // read 3x3 region of source for pixel 1
    float2 c = ...
    float r0 = src.sample(sam, c, int2(-1,-1)).x;
    // read r1-r8
    // apply Sobel filter for pixel 1
    float gx = (r2-r0) + 2.0f*(r5-r3) + (r8-r6);
    float gy = (r0-r6) + 2.0f*(r1-r7) + (r2-r8);
    float4 g = float4(sqrt(gx * gx + gy * gy));
    dst.write(g, static_cast<uint2>(dstCoord));

    // continue to pixel 2
kernel void sobel_1_1(/* ... */
    ushort2 tid [[ thread_position_in_grid ]])
{
    ushort2 gid = ushort2(tid.x*2,tid.y);
    ushort2 dstCoord = ...
    ...
    // read 3x3 region of source for pixel 1
    float2 c = ...
    float r0 = src.sample(sam, c, int2(-1,-1)).x;
    // read r1-r8
    // apply Sobel filter for pixel 1
    float gx = (r2-r0) + 2.0f*(r5-r3) + (r8-r6);
    float gy = (r0-r6) + 2.0f*(r1-r7) + (r2-r8);
    float4 g = float4(sqrt(gx * gx + gy * gy));
    dst.write(g, static_cast<uint2>(dstCoord));
    // continue to pixel 2
// continue to pixel 2...
dstCoord.x++;
if (dstCoord.x >= params.dstBounds.z)
    return;

// reuse 2x3 region from pixel 1,
// read additional 1x3 region for pixel 2
r0 = r1; r1 = r2; r2 = src.sample(sam, c, int2(2,-1)).x;
r3 = r4; r4 = r5; r5 = src.sample(sam, c, int2(2,0)).x;
r6 = r7; r7 = r8; r8 = src.sample(sam, c, int2(2,1)).x;

// apply Sobel filter for pixel 2
float gx = (r2-r0) + 2.0f*(r5-r3) + (r8-r6);
float gy = (r0-r6) + 2.0f*(r1-r7) + (r2-r8);
float4 g = float4(sqrt(gx * gx + gy * gy));
dst.write(g, static_cast<uint2>(dstCoord));
// continue to pixel 2...

dstCoord.x++;
if (dstCoord.x >= params.dstBounds.z)
    return;

// reuse 2x3 region from pixel 1,
// read additional 1x3 region for pixel 2
r0 = r1; r1 = r2; r2 = src.sample(sam, c, int2(2,-1)).x;
r3 = r4; r4 = r5; r5 = src.sample(sam, c, int2(2,0)).x;
r6 = r7; r7 = r8; r8 = src.sample(sam, c, int2(2,1)).x;

// apply Sobel filter for pixel 2
float gx = (r2-r0) + 2.0f*(r5-r3) + (r8-r6);
float gy = (r0-r6) + 2.0f*(r1-r7) + (r2-r8);
float4 g = float4(sqrt(gx * gx + gy * gy));
dst.write(g, static_cast<uint2>(dstCoord));
// continue to pixel 2...

dstCoord.x++;

if (dstCoord.x >= params.dstBounds.z)
    return;

// reuse 2x3 region from pixel 1,
read additional 1x3 region for pixel 2

r0 = r1; r1 = r2; r2 = src.sample(sam, c, int2(2,-1)).x;

r3 = r4; r4 = r5; r5 = src.sample(sam, c, int2(2,0)).x;

r6 = r7; r7 = r8; r8 = src.sample(sam, c, int2(2,1)).x;

// apply Sobel filter for pixel 2

float gx = (r2-r0) + 2.0f*(r5-r3) + (r8-r6);
float gy = (r0-r6) + 2.0f*(r1-r7) + (r2-r8);
float4 g = float4(sqrt(gx * gx + gy * gy));
dst.write(g, static_cast<uint2>(dstCoord));
// continue to pixel 2...
dstCoord.x++;

if (dstCoord.x >= params.dstBounds.z)
    return;

// reuse 2x3 region from pixel 1,
read additional 1x3 region for pixel 2
r0 = r1; r1 = r2; r2 = src.sample(sam, c, int2(2,-1)).x;
r3 = r4; r4 = r5; r5 = src.sample(sam, c, int2(2,0)).x;
r6 = r7; r7 = r8; r8 = src.sample(sam, c, int2(2,1)).x;

// apply Sobel filter for pixel 2
float gx = (r2-r0) + 2.0f*(r5-r3) + (r8-r6);
float gy = (r0-r6) + 2.0f*(r1-r7) + (r2-r8);
float4 g = float4(sqrt(gx * gx + gy * gy));
dst.write(g, static_cast<uint2>(dstCoord));
// continue to pixel 2...
dstCoord.x++;  
if (dstCoord.x >= params.dstBounds.z)
    return;

// reuse 2x3 region from pixel 1,
// read additional 1x3 region for pixel 2
r0 = r1; r1 = r2; r2 = src.sample(sam, c, int2(2,-1)).x;
r3 = r4; r4 = r5; r5 = src.sample(sam, c, int2(2,0)).x;
r6 = r7; r7 = r8; r8 = src.sample(sam, c, int2(2,1)).x;

// apply Sobel filter for pixel 2
float gx = (r2-r0) + 2.0f*(r5-r3) + (r8-r6);
float gy = (r0-r6) + 2.0f*(r1-r7) + (r2-r8);
float4 g = float4(sqrt(gx * gx + gy * gy));
dst.write(g, static_cast<uint2>(dstCoord));
Compute Kernel Organization

Considerations

Use barriers with the smallest possible scope

- SIMD-width threadgroups make threadgroup_barrier unnecessary
- For thread groups $\leq$ SIMD group size, use simdgroup_barrier

Usually faster than trying to squeeze out additional reuse
Shader Performance Fundamentals

Conclusion
Shader Performance Fundamentals

Conclusion

Pick appropriate address spaces for arguments
Shader Performance Fundamentals

Conclusion

Pick appropriate address spaces for arguments
Structure your data/rendering to leverage buffer preloading
Shader Performance Fundamentals

Conclusion

Pick appropriate address spaces for arguments
Structure your data/rendering to leverage buffer preloading
Use early fragment tests to reduce shading of objects with resource writes
Shader Performance Fundamentals

Conclusion

Pick appropriate address spaces for arguments
Structure your data/rendering to leverage buffer preloading
Use early fragment tests to reduce shading of objects with resource writes
Do enough work in each compute thread to amortize launch overhead
Shader Performance Fundamentals

Conclusion

Pick appropriate address spaces for arguments
Structure your data/rendering to leverage buffer preloading
Use early fragment tests to reduce shading of objects with resource writes
Do enough work in each compute thread to amortize launch overhead
Use the smallest-scoped barrier you can
Tuning Shader Code
GPU Architecture

Focus on the bottleneck to improve performance
Improving non-bottlenecks can still save power
Typical Shader Bottlenecks

- ALU bandwidth
- Memory bandwidth
- Memory issue rate
- Latency/occupancy/register usage
Optimization Opportunities

Data types
Arithmetic
Control flow
Memory access
Data Types

Overview

A8 and later GPUs use 16-bit register units

Use the smallest possible data type

- Fewer registers used → better occupancy
- Faster arithmetic → better ALU usage

Use half and short for arithmetic when possible

- Energy: half < float < short < int
Data Types
Using half and short arithmetic

For texture reads, interpolates, and math, use half when possible

- Not the texture format, the value returned from sample()
- Conversions are typically *free*, even between float and half

Half-precision numerics and limitations are different from float

- Minimum normal value: $6.1 \times 10^{-5}$
- Maximum normal value: 65504
  - Classic bug: writing “65535” as a half will actually give you infinity
Data Types

Using half and short arithmetic

Use `ushort` for local thread IDs, and for global thread IDs when possible.
Data Types
Using half and short arithmetic

Use `ushort` for local thread IDs, and for global thread IDs when possible.

```c
kernel void LocalAdd( ... 
    uint threadGroupID [[ thread_position_in_threadgroup]],
    uint threadGroupGridID [[ threadgroup_position_in_grid ]])
```
Data Types
Using half and short arithmetic

Use **ushort** for local thread IDs, and for global thread IDs when possible.

```
kernel void
LocalAdd( ...
  ushort threadGroupID [[ thread_position_in_threadgroup]],
  ushort threadGroupGridID [[ threadgroup_position_in_grid ]])
```

```
kernel void
LocalAdd( ...
  uint   threadGroupID [[ thread_position_in_threadgroup]],
  uint   threadGroupGridID [[ threadgroup_position_in_grid ]])
```
Data Types

Using half and short arithmetic

Avoid float literals when doing half-precision operations
Data Types

Using half and short arithmetic

Avoid float literals when doing half-precision operations

```c
half foo(half a, half b)
{
    return clamp(a, b, -2.0, 5.0);
}
```
Data Types

Using half and short arithmetic

Avoid float literals when doing half-precision operations

```
half foo(half a, half b)
{
    return clamp(a, b, -2.0h, 5.0h);
}
```
Data Types

Using half and short arithmetic

Avoid `char` for arithmetic if not necessary

- Not natively supported for arithmetic
- May result in extra instructions
Arithmetic

Built-ins

Use built-ins where possible

- Free modifiers: negate, abs(), saturate()
  - Native hardware support
Arithmetic

Built-ins

Use built-ins where possible

• Free modifiers: `negate`, `abs()`, `saturate()`
  - Native hardware support

```c
kernel void
myKernel(...)
{
    // fabs on p.a negation on p.b and clamp of (fabs(p.a) * -p.b * input[threadID]) are free
    float4 f = saturate((fabs(p.a) * -p.b * input[threadID]));
    ...
}
```
A8 and later GPUs are scalar

- Vectors are fine to use, but compiler splits them
  - Don’t waste time vectorizing code when not naturally vector
Arithmetic

ILP (Instruction Level Parallelism) not very important

- Register usage typically matters more
  - Don’t restructure for ILP, e.g. using multiple accumulators when not necessary
Arithmetic

ILP (Instruction Level Parallelism) not very important

- Register usage typically matters more
  - Don’t restructure for ILP, e.g. using multiple accumulators when not necessary

```c
// unnecessary, possibly slower
float accum1 = 0, accum2 = 0;
for (int x = 0; x < n; x += 2) {
    accum1 += a[x] * b[x];
    accum2 += a[x+1] * b[x+1];
}
return accum1 + accum2;
```
Arithmetic

ILP (Instruction Level Parallelism) not very important

- Register usage typically matters more
- Don’t restructure for ILP, e.g. using multiple accumulators when not necessary

```c
// unnecessary, possibly slower
float accum1 = 0, accum2 = 0;
for (int x = 0; x < n; x += 2) {
    accum1 += a[x] * b[x];
    accum2 += a[x+1] * b[x+1];
}
return accum1 + accum2;
```

```c
// better
float accum = 0;
for (int x = 0; x < n; x += 2) {
    accum += a[x] * b[x];
    accum += a[x+1] * b[x+1];
}
return accum;
```
A8 and later GPUs have very fast ‘select’ instructions (ternary operators)

• Don’t do ‘clever’ things like multiplying by 1 or 0 instead
A8 and later GPUs have very fast ‘select’ instructions (ternary operators)

• Don’t do ‘clever’ things like multiplying by 1 or 0 instead

```c
// slow: no need to fake ternary op
if (foo)
   m = 0.0h;
else
   m = 1.0h;
half p = v * m;
```
A8 and later GPUs have very fast ‘select’ instructions (ternary operators)

• Don’t do ‘clever’ things like multiplying by 1 or 0 instead

```c
// slow: no need to fake ternary op
if (foo)
    m = 0.0h;
else
    m = 1.0h;
half p = v * m;
```

```c
// fast: ternary op
half p = foo ? v : 0.0h;
```
Arithmetic

Integer divisions

Avoid division or modulus by denominators that aren’t literal/function constants

```cpp
constant int width [[ function_constant(0) ]];
struct constInputs {
    int width;
};
vertex float4 vertexMain(...)
{
    // extremely slow: constInputs.width not known at compile time
    int onPos0 = vertexIn[vertex_id] / constInputs.width;

    // fast: 256 is a compile-time constant
    int onPos1 = vertexIn[vertex_id] / 256;

    // fast: width provided at compile time
    int onPos2 = vertexIn[vertex_id] / width;
}
```
Avoid division or modulus by *denominators that aren’t literal/function constants*

```c
constant int width [[ function_constant(0) ]];
struct constInputs {
  int width;
};
vertex float4 vertexMain(...)
{
  // extremely slow: constInputs.width not known at compile time
  int onPos0 = vertexIn[vertex_id] / constInputs.width;

  // fast: 256 is a compile-time constant
  int onPos1 = vertexIn[vertex_id] / 256;
  // fast: width provided at compile time
  int onPos2 = vertexIn[vertex_id] / width;
}
```
Fast-math

In Metal, fast-math is on by default

Often >50% perf gain on arithmetic, possibly much more

Uses faster arithmetic built-ins with well-defined precision guarantees

Maintains intermediate precision

Ignores strict NaN/infinity/signed zero semantics
  • but will not introduce new NaNs

Might perform arithmetic reassociation
  • but will not perform arithmetic distribution
Arithmetic

Fast-math

If you absolutely cannot use fast-math:

- Use FMA built-in (fused multiply-add) to regain some performance
  - Having fast-math off prohibits this optimization (and many others)
Arithmetic

Fast-math

If you absolutely cannot use fast-math:

• Use FMA built-in (fused multiply-add) to regain some performance
  - Having fast-math off prohibits this optimization (and many others)

```c
kernel void
myKernel(…)
{
  // d = a * b + c;
  float d = fma(a, b, c);
  …
}
```
Control Flow

Control flow uniform across SIMD width is generally fast
• Dynamically uniform (uniform at runtime) is also fast
Divergence within a SIMD means running both paths
Control Flow

Switch fall-throughs: can create unstructured control flow

- Can result in significant code duplication — avoid if possible

```java
switch (numItems) {
    [...]  
    case 2:
        processItem(1);
        /* fall-through */
    case 1:
        processItem(0);
        break;
}
```
Memory Access

Stack access

Avoid dynamically indexed non-constant stack arrays

- Cost can be catastrophic: 30% due to one 32-byte array in a real-world app

Loops with stack arrays will typically be unrolled to eliminate the dynamic access
Memory Access

Stack access

Avoid dynamically indexed non-constant stack arrays

- Cost can be catastrophic: 30% due to one 32-byte array in a real-world app

Loops with stack arrays will typically be unrolled to eliminate the dynamic access

```c
// bad: dynamically indexed stack array
int foo(int a, int b, int c) {
    int tmp[2] = { a, b };
    return tmp[c];
}
```
Memory Access

Stack access

Avoid dynamically indexed non-constant stack arrays

- Cost can be catastrophic: 30% due to one 32-byte array in a real-world app

Loops with stack arrays will typically be unrolled to eliminate the dynamic access

// bad: dynamically indexed stack array
int foo(int a, int b, int c) {
    int tmp[2] = { a, b };
    return tmp[c];
}

// okay: constant array
int foo(int a, int b, int c) {
    int tmp2[2] = { 1, 2 };
    return tmp2[c];
}
Memory Access

Stack access

Avoid dynamically indexed non-constant stack arrays

- Cost can be catastrophic: 30% due to one 32-byte array in a real-world app

Loops with stack arrays will typically be unrolled to eliminate the dynamic access

```c
// bad: dynamically indexed stack array
int foo(int a, int b, int c) {
    int tmp[2] = { a, b };
    return tmp[c];
}

// okay: loop will be unrolled
int foo(int a, int b, int c) {
    int tmp3[3] = { a, b, c };
    int sum = 0;
    for (int i = 0; i < 3; ++i)
        sum += tmp3[i];
    return sum;
}

// okay: constant array
int foo(int a, int b, int c) {
    int tmp2[2] = { 1, 2 };
    return tmp2[c];
}
```
Memory Access

Loads and stores

One big vector load/store is faster than multiple scalar ones
• The compiler will try to vectorize neighboring loads/stores
One big vector load/store is faster than multiple scalar ones

- The compiler will try to vectorize neighboring loads/stores

```c
struct foo {
    float a;
    float b[7];
    float c;
};

// bad: a and c aren't adjacent.
will result in two scalar loads

float sum_mul(foo *x, int n) {
    float sum = 0;
    for (uint i = 0; i < n; ++i)
        sum += x[i].a * x[i].c;
}```
Memory Access
Loads and stores

One big vector load/store is faster than multiple scalar ones

- The compiler will try to vectorize neighboring loads/stores

```c
struct foo {
    float a;
    float b[7];
    float c;
};

// bad: a and c aren’t adjacent.
// will result in two scalar loads
float sum_mul(foo *x, int n) {
    float sum = 0;
    for (uint i = 0; i < n; ++i)
        sum += x[i].a * x[i].c;
}
```

```c
struct foo {
    float2 a;
    float b[7];
};

// good: a is now a vector, so there
// will be one load.
float sum_mul(foo *x, int n) {
    float sum = 0;
    for (uint i = 0; i < n; ++i)
        sum += x[i].a.x * x[i].a.y;
}
```
One big vector load/store is faster than multiple scalar ones

- The compiler will try to vectorize neighboring loads/stores

```c
struct foo {
    float a;
    float b[7];
    float c;
};

// bad: a and c aren't adjacent.
// will result in two scalar loads
float sum_mul(foo *x, int n) {
    float sum = 0;
    for (uint i = 0; i < n; ++i)
        sum += x[i].a * x[i].c;
}

// good: a is now a vector, so there will be one load.
struct foo {
    float2 a;
    float b[7];
};

float sum_mul(foo *x, int n) {
    float sum = 0;
    for (uint i = 0; i < n; ++i)
        sum += x[i].a.x * x[i].a.y;
}

// also good: compiler will likely be able to vectorize.
struct foo {
    float a;
    float c;
    float b[7];
};

float sum_mul(foo *x, int n) {
    float sum = 0;
    for (uint i = 0; i < n; ++i)
        sum += x[i].a * x[i].c;
}
```
Memory Access

Loads and stores

Use `int` or smaller types for device memory addressing (not `uint`)
Memory Access

Loads and stores

Use `int` or smaller types for device memory addressing (not `uint`)

```c
kernel void Accumulate( const device int *a [[ buffer(0) ]], ... ) {
    int sum = 0;
    for (uint i = 0; i < nElems; i++)
        sum += a[i];
}```
Load and stores

Memory Access

Use `int` or smaller types for device memory addressing (not `uint`).

```c
kernel void Accumulate( const device int *a [[ buffer(0) ]], …) {
    int sum = 0;
    for (int i = 0; i < nElems; i++)
        sum += a[i];
}
```

```c
kernel void Accumulate( const device int *a [[ buffer(0) ]], …) {
    int sum = 0;
    for (uint i = 0; i < nElems; i++)
        sum += a[i];
}
```
Latency/Occupancy

GPUs hide latency with large-scale multithreading
When waiting for something to finish (e.g. a texture read) they run another thread
Latency/Occupancy

The more latency, the more threads you need to hide it
The more registers you use, the fewer threads you have
• The number of threads you can have is called the ‘occupancy’
• Threadgroup memory usage can also bound the occupancy
‘Latency-limited’: too few threads to hide latency of a shader

Measure occupancy in Metal compute shaders using `MTLComputePipelineState maxTotalThreadsPerThreadgroup()`
Memory Access

Latency-hiding: False dependency example
Memory Access

Latency-hiding: False dependency example

// REAL dependency: 2 waits

half a = tex0.sample(s0, c0);
half res = 0.0h;

// wait on ‘a’
if (a >= 0.0h) {
    half b = tex1.sample(s1, c1);

    // wait on ‘b’
    res = a * b;
}


Memory Access
Latency-hiding: False dependency example

// REAL dependency: 2 waits
half a = tex0.sample(s0, c0);
half res = 0.0h;

🔴// wait on ‘a’
if (a >= 0.0h) {
    half b = tex1.sample(s1, c1);
    🔴// wait on ‘b’
    res = a * b;
}

// FALSE dependency: 2 waits
half a = tex0.sample(s0, c0);
half res = 0.0h;

🔴// wait on ‘a’
if (foo) {
    half b = tex1.sample(s1, c1);
    🔴// wait on ‘b’
    res = a * b;
}
Memory Access
Latency-hiding: False dependency example

// REAL dependency: 2 waits
half a = tex0.sample(s0, c0);
half res = 0.0h;

🔴 // wait on 'a'
if (a >= 0.0h) {
    half b = tex1.sample(s1, c1);
    // wait on 'b'
    res = a * b;
}

// FALSE dependency: 2 waits
half a = tex0.sample(s0, c0);
half res = 0.0h;

🔴 // wait on 'a'
if (foo) {
    half b = tex1.sample(s1, c1);
    // wait on 'b'
    res = a * b;
}

// NO dependency: 1 wait
half a = tex0.sample(s0, c0);
half b = tex1.sample(s1, c1);
half res = 0.0h;

🔴 // wait on 'a' and 'b'
if (foo) {
    res = a * b;
}
Summary
Summary

Pick correct address spaces and data structures/layouts

• Performance impact of getting this wrong can be very high
Summary

Pick correct address spaces and data structures/layouts

• Performance impact of getting this wrong can be very high

Work with the compiler — write what you mean

• “Clever” code often prevents the compiler from doing its job
Summary

Pick correct address spaces and data structures/layouts

- Performance impact of getting this wrong can be very high

Work with the compiler — write what you mean

- “Clever” code often prevents the compiler from doing its job

Keep an eye out for pitfalls, not just micro-optimizations

- Can dwarf all other potential optimizations
Summary

Pick correct address spaces and data structures/layouts

- Performance impact of getting this wrong can be very high

Work with the compiler — write what you mean

- "Clever" code often prevents the compiler from doing its job

Keep an eye out for pitfalls, not just micro-optimizations

- Can dwarf all other potential optimizations

Feel free to experiment!

- Some tradeoffs, like latency vs. throughput, have no universal rule
More Information

https://developer.apple.com/wwdc16/606
## Related Sessions

<table>
<thead>
<tr>
<th>Session</th>
<th>Location</th>
<th>Date/Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adopting Metal, Part 1</td>
<td>Nob Hill</td>
<td>Tuesday 1:40PM</td>
</tr>
<tr>
<td>Adopting Metal, Part 2</td>
<td>Nob Hill</td>
<td>Tuesday 3:00PM</td>
</tr>
<tr>
<td>What’s New in Metal, Part 1</td>
<td>Pacific Heights</td>
<td>Wednesday 11:00AM</td>
</tr>
<tr>
<td>What’s New in Metal, Part 2</td>
<td>Pacific Heights</td>
<td>Wednesday 1:40PM</td>
</tr>
<tr>
<td>Labs</td>
<td>Developer Tools Lab B</td>
<td>Time</td>
</tr>
<tr>
<td>---------------------------</td>
<td>-----------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>Xcode Open Hours</td>
<td>Lab B</td>
<td>Wednesday 3:00PM</td>
</tr>
<tr>
<td>Metal Lab</td>
<td>Lab A</td>
<td>Thursday 12:00PM</td>
</tr>
<tr>
<td>Xcode Open Hours</td>
<td>Lab B</td>
<td>Friday 9:00AM</td>
</tr>
<tr>
<td>Xcode Open Hours</td>
<td>Lab B</td>
<td>Friday 12:00PM</td>
</tr>
<tr>
<td>LLVM Compiler, Objective-C, and C++ Lab</td>
<td>Lab C</td>
<td>Friday 4:30PM</td>
</tr>
</tbody>
</table>