Sample Code

Encoding indirect command buffers on the GPU

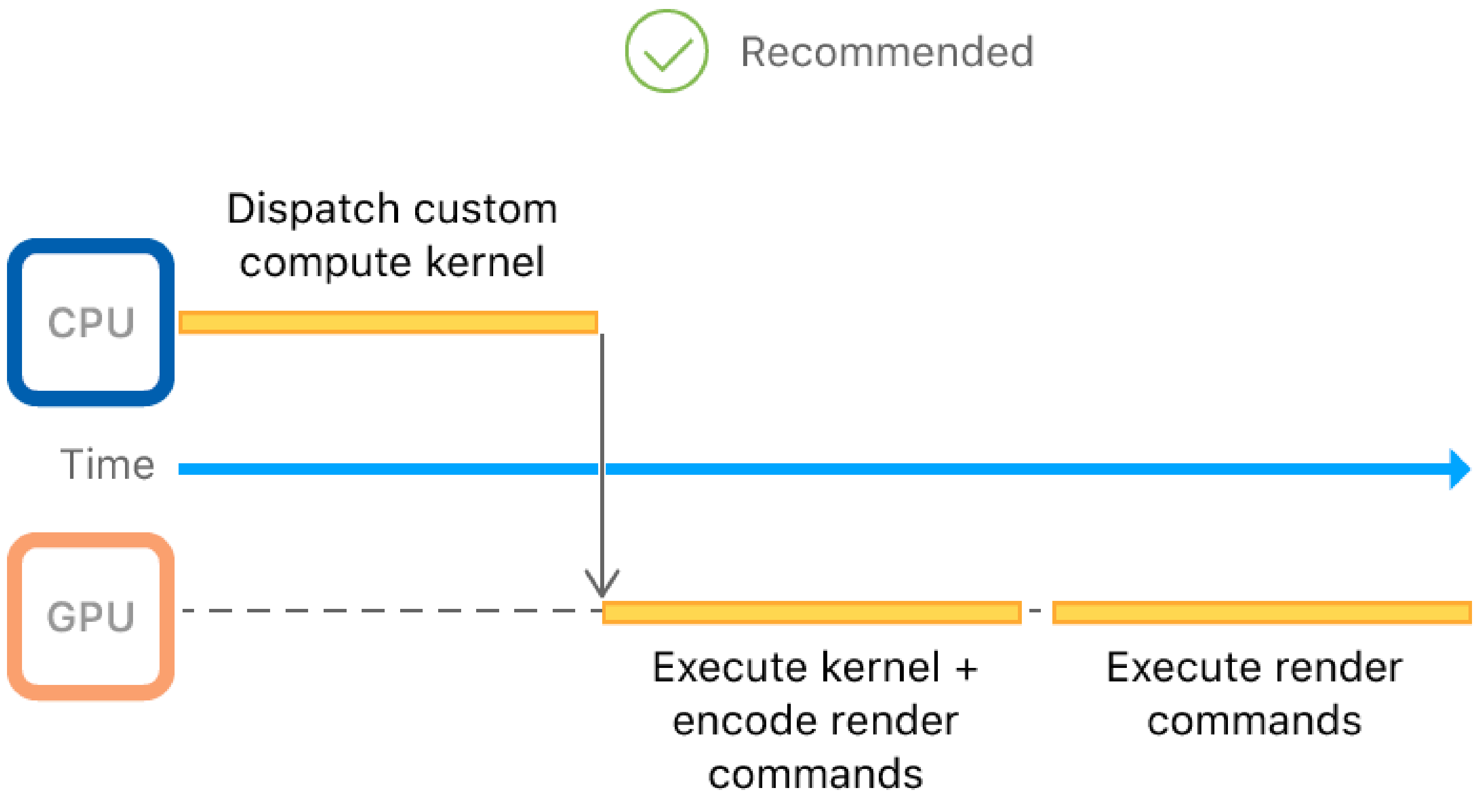
Maximize CPU to GPU parallelization by generating render commands on the GPU.

Download

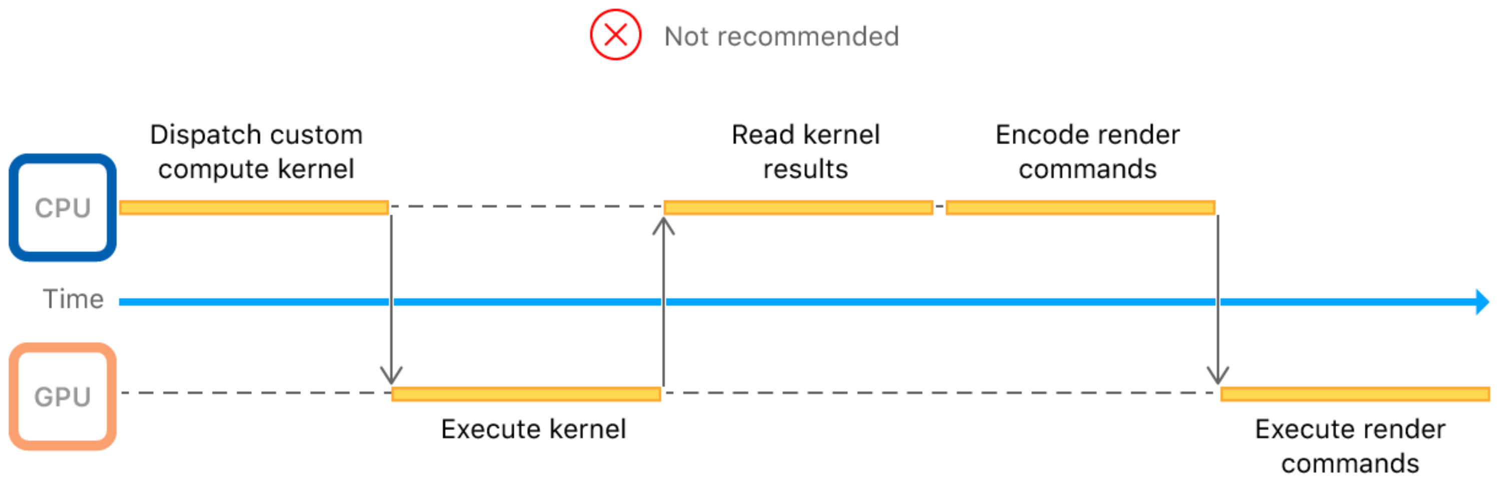
iOS 12.0+ | iPadOS 12.0+ | macOS 10.14+ | Xcode 12.3+

Overview

This sample app demonstrates how to use *indirect command buffers* (ICB) to issue rendering instructions from the GPU. When you have a rendering algorithm that runs in a compute kernel, use ICBs to generate draw calls based on your algorithm's results. The sample app uses a compute kernel to remove invisible objects submitted for rendering, and generates draw commands only for the objects currently visible in the scene.


Recommended
Dispatch custom compute kernel
CPU
Time
GPU
Execute kernel + encode render commands
Execute render commands

Without ICBs, you can’t submit rendering commands on the GPU. Instead, the CPU waits for your compute kernel’s results before generating the render commands. Then, the GPU waits for the rendering commands to make it across the CPU to GPU bridge. The following diagram shows how this creates a slower round trip:



The sample code project, Encoding Indirect Command Buffers on the CPU introduces ICBs by creating a single ICB to reuse its commands every frame. While the former sample saved expensive command-encoding time by reusing commands, this sample uses ICBs to effect a GPU-driven rendering pipeline.

The techniques shown by this sample include issuing draw calls from the GPU, and the process of executing a select set of draws.

Getting started

This project contains targets for macOS and iOS. Run the iOS scheme on a physical device because Metal isn't supported in the simulator.

The sample calls an `MTLComputeCommandEncoder` instances's `dispatchThreads(_:threadsPerThreadgroup:)` method, which is available to a GPU that supports the following feature sets and later:

- MTLFeatureSet_iOS_GPUFamily4_v2
- MTLFeatureSet_macOS_GPUFamily2_v1

Define the data read by the ICB

In an ideal scenario, you store each mesh in its own buffer. However, on iOS, kernels running on the GPU can only access a limited number of data buffers per execution. To reduce the number of buffers needed during the ICBs execution, you pack all meshes into a single buffer at varying offsets. Then, use another buffer to store the offset and size of each mesh. The process to do this follows.

At initialization, create the data for each mesh:

```
for(int objectIdx = 0; objectIdx < AAPLNumObjects; objectIdx++)
{
    // Choose the parameters to generate a mesh so that each one is unique.
    uint32_t numTeeth = random() % 50 + 3;
    float innerRatio = 0.2 + (random() / (1.0 * RAND_MAX)) * 0.7;
    float toothWidth = 0.1 + (random() / (1.0 * RAND_MAX)) * 0.4;
    float toothSlope = (random() / (1.0 * RAND_MAX)) * 0.2;

    // Create a vertex buffer and initialize it with a unique 2D gear mesh.
    tempMeshes[objectIdx] = [self newGearMeshWithNumTeeth:numTeeth
                                                innerRatio:innerRatio
                                                toothWidth:toothWidth
                                                toothSlope:toothSlope];
}
```

Count the individual and accumulated mesh sizes and create the container buffer:

```
size_t bufferSize = 0;

for(int objectIdx = 0; objectIdx < AAPLNumObjects; objectIdx++)
{
    size_t meshSize = sizeof(AAPLVertex) * tempMeshes[objectIdx].numVerts;
    bufferSize += meshSize;
}

_vertexBuffer = [_device newBufferWithLength:bufferSize options:0];
```

Finally, insert each mesh into the container buffer while noting its offset and size in the second buffer:

```
for(int objectIdx = 0; objectIdx < AAPLNumObjects; objectIdx++)
{
    // Store the mesh metadata in the `params` buffer.

    params[objectIdx].numVertices = tempMeshes[objectIdx].numVerts;

    size_t meshSize = sizeof(AAPLVertex) * tempMeshes[objectIdx].numVerts;

    params[objectIdx].startVertex = currentStartVertex;

    // Pack the current mesh data in the combined vertex buffer.

    AAPLVertex* meshStartAddress = ((AAPLVertex*)_vertexBuffer.contents) + currentSt

    memcpy(meshStartAddress, tempMeshes[objectIdx].vertices, meshSize);

    currentStartVertex += tempMeshes[objectIdx].numVerts;

    free(tempMeshes[objectIdx].vertices);

    // Set the other culling and mesh rendering parameters.

    // Set the position of each object to a unique space in a grid.
    vector_float2 gridPos = (vector_float2){objectIdx % AAPLGridWidth, objectIdx / A
    params[objectIdx].position = gridPos * AAPLObjecDistance;

    params[objectIdx].boundingRadius = AAPLObjectSize / 2.0;
}
```

Update the data read by the ICB dynamically

By culling non-visible vertices from the data fed to the rendering pipeline, you save significant rendering time and effort. To do that, use the same compute kernel that encodes the ICB's commands to continually update the ICB’s data buffers:

```
// Check whether the object at 'objectIndex' is visible and set draw parameters if s
// Otherwise, reset the command.
kernel void
cullMeshesAndEncodeCommands(uint                         objectIndex   [[ thread_pos
                            constant AAPLFrameState     *frame_state   [[ buffer(AAP
                            device AAPLObjectPerameters *object_params [[ buffer(AAP
                            device AAPLVertex           *vertices      [[ buffer(AAP
                            device ICBContainer         *icb_container [[ buffer(AAP
{
    float2 worldObjectPostion  = frame_state->translation + object_params[objectInde
    float2 clipObjectPosition  = frame_state->aspectScale * AAPLViewScale * worldObj

    const float rightBounds =  1.0;
    const float leftBounds  = -1.0;
    const float upperBounds =  1.0;
    const float lowerBounds = -1.0;

    bool visible = true;

    // Set the bounding radius in the view space.
    const float2 boundingRadius = frame_state->aspectScale * AAPLViewScale * object_

    // Check if the object's bounding circle has moved outside of the view bounds.
    if(clipObjectPosition.x + boundingRadius.x < leftBounds  ||
       clipObjectPosition.x - boundingRadius.x > rightBounds ||
       clipObjectPosition.y + boundingRadius.y < lowerBounds ||
       clipObjectPosition.y - boundingRadius.y > upperBounds)
    {
        visible = false;
    }
    // Get indirect render commnd object from the indirect command buffer given the
    // index to set parameters for drawing (or not drawing) the object.
    render_command cmd(icb_container->commandBuffer, objectIndex);

    if(visible)
    {
        // Set the buffers and add a draw command.
        cmd.set_vertex_buffer(frame_state, AAPLVertexBufferIndexFrameState);
        cmd.set_vertex_buffer(object_params, AAPLVertexBufferIndexObjectParams);
        cmd.set_vertex_buffer(vertices, AAPLVertexBufferIndexVertices);
        cmd.draw_primitives(primitive_type::triangle,
                            object_params[objectIndex].startVertex,
                            object_params[objectIndex].numVertices, 1,
                            objectIndex);
    }

    // If the object isn't visible, Metal doesn't set a draw command as long as the
    // the indirect command buffer commands with a blit encoder before encoding the
```

The parallel nature of the GPU partitions the compute task for you, resulting in multiple offscreen meshes getting culled concurrently.

Pass an ICB to a compute kernel using an argument buffer

To get an ICB on the GPU and make it accessible to a compute kernel, you pass it through an argument buffer, as follows:

Define the container argument buffer as a structure that contains one member, the ICB:

```
// This is the argument buffer that contains the ICB.
struct ICBContainer
{
    command_buffer commandBuffer [[ id(AAPLArgumentBufferIDCommandBuffer) ]];
};
```

Encode the ICB into the argument buffer:

```
id<MTLArgumentEncoder> argumentEncoder =
    [GPUCommandEncodingKernel newArgumentEncoderWithBufferIndex:AAPLKernelBufferInde

_icbArgumentBuffer = [_device newBufferWithLength:argumentEncoder.encodedLength
                                          options:MTLResourceStorageModeShared];
_icbArgumentBuffer.label = @"ICB Argument Buffer";

[argumentEncoder setArgumentBuffer:_icbArgumentBuffer offset:0];

[argumentEncoder setIndirectCommandBuffer:_indirectCommandBuffer
                                  atIndex:AAPLArgumentBufferIDCommandBuffer];
```

Pass the ICB (`_indirectCommandBuffer`) to the kernel by setting the argument buffer on the kernel's compute command encoder:

```
[computeEncoder setBuffer:_icbArgumentBuffer offset:0 atIndex:AAPLKernelBufferIndexC
```

Because you pass the ICB through an argument buffer, standard argument buffer rules apply. Call `useResource` on the ICB to tell Metal to prepare its use:

```
[computeEncoder useResource:_indirectCommandBuffer usage:MTLResourceUsageWrite];
```

Encode and optimize ICB commands

Reset the ICB's commands to their initial before beginning encoding:

```
[resetBlitEncoder resetCommandsInBuffer:_indirectCommandBuffer
                              withRange:NSMakeRange(0, AAPLNumObjects)];
```

Encode the ICB's commands by dispatching the compute kernel:

```
[computeEncoder dispatchThreads:MTLSizeMake(AAPLNumObjects, 1, 1)
          threadsPerThreadgroup:MTLSizeMake(threadExecutionWidth, 1, 1)];
```

Optimize your ICB commands to remove empty commands or redundant state by calling `optimizeIndirectCommandBuffer:withRange:`:

```
[optimizeBlitEncoder optimizeIndirectCommandBuffer:_indirectCommandBuffer
                                         withRange:NSMakeRange(0, AAPLNumObjects)];
```

This sample optimizes ICB commands because redundant state results from the kernel setting a buffer for each draw, and encoding empty commands for each invisible object. By removing the empty commands, you can free up a significant number of blank spaces in the command buffer that Metal otherwise spends time skipping at runtime.

Note

If you optimize an indirect command buffer, you won't be able to call `executeCommandsInBuffer:withRange:` with a range that starts in the optimized region. Instead, specify a range staring at the beginning and finishing within or at the end of the optimized region.

Execute the ICB

Draw the onscreen meshes by calling `executeCommandsInBuffer` on your render command encoder:

```
[renderEncoder executeCommandsInBuffer:_indirectCommandBuffer withRange:NSMakeRange(
```

While you can encode an ICB's commands in a compute kernel, you call `executeCommandsInBuffer` from your host app to encode a single command that contains all of the commands encoded by the compute kernel. By doing this, you choose the queue and buffer that the ICB's commands go into. When you call `executeIndirectCommandBuffer` determines the placement of the ICB's commands among any other commands you may also encode in the same buffer.

See Also

Indirect command buffers

Creating an indirect command buffer

Configure a descriptor to specify the properties of an indirect command buffer.

Specifying drawing and dispatch arguments indirectly

Use indirect commands if you don't know your draw or dispatch call arguments when you encode the command.

{} Encoding indirect command buffers on the CPU

Reduce CPU overhead and simplify your command execution by reusing commands.

`protocol MTLIndirectCommandBuffer`

A command buffer containing reusable commands, encoded either on the CPU or GPU.

`class MTLIndirectCommandBufferDescriptor`

A configuration you create to customize an indirect command buffer.

`struct MTLIndirectCommandType`

The types of commands that you can encode into the indirect command buffer.

`struct MTLIndirectCommandBufferExecutionRange`

A range of commands in an indirect command buffer.

`func MTLIndirectCommandBufferExecutionRangeMake(UInt32, UInt32) -> MTLIndirectCommandBufferExecutionRange`

Creates a command execution range.