Multithreading in Vulkan

By: Victor Li

All code examples Copyright © Sascha Willems 2016, with modifications

Multithreading in OpenGL

OpenGL:

- Technically “multithreadable”, but a nightmare to work with
- Can create multiple threads
- Each thread does graphics work in its own context
- How do we refer to the work done in each context?
  - Contexts aren’t “shareable”
  - Can’t show them all independently!
- Graphics work isn’t thread-safe
- How do we avoid race conditions without proper synchronization tools?
  - Hmm……
Multithreading in Vulkan

- Command buffer submission thread-safe across multiple queues
  - We can do whatever we want before we submit buffers to the queue(s)!
- Handy dandy synchronization tools
- Vulkan designed functions for multithreading!

When to multi-thread...

- NVIDIA recommends these three cases:
- Pipeline construction
- Updating resources
  - Change vertex buffers, do instanced animations, apply transformations, etc.
  - Updating uniform buffers
- Rendering ← we'll focus on this!
Different ways of multi-threading rendering

- Single-pass, single queue
- Single-pass, multiple queues

We're going to look at:

- Single-pass, single queue

Single-pass, single queue

- Worker threads
  - Construct individual command buffers, each factoring into our scene
- Main thread
  - Assemble worker threads’ completed command buffers into one command buffer for queue submission

Command buffer types differ between worker threads and the main thread!
### Command Buffers: 2 types!

**Primary command buffers (main thread)**
- Render pass begin
- Bind pipelines
- Draw calls
- End render pass
- Submit to queue

**Secondary command buffers (worker thread)**
- Bind pipelines
- Draw calls

**REMEMBER:** Primary/secondary buffers NOT related to front and back buffering (Vulkan uses a hi!

### Main Thread/Program Structures

**Single-threaded**
- **Setup**: pipeline creation, push constants, etc. etc.
  - Set up resources for our threading
- Begin a render pass with PCB
- Update data for PCB
- Draw with PCB
- End the render pass
- Submit to queue
- Destroy everything

**Multi-threaded**
- **Setup**: pipeline creation, push constants, etc. etc.
  - Set up resources for our threading
- Begin a render pass with PCB
- Update data for SCB
- Draw with SCBs
  - Construct SCBs using a thread pool
  - Collect SCBs
- Assemble and execute SCBs inside PCB
- End the render pass
- Submit to queue
- Destroy everything
What is a thread pool?

- A thread pool is a collection of worker threads
  - Each thread can be assigned work to do in its “work queue”
  - A thread pool can be waited on, meaning we execute its jobs and wait on their completion
- In our case, we use a very simple class-based implementation of thread pool functionality using the C++ `<thread>` library

  https://github.com/SaschaWillems/Vulkan/blob/master/base/threadpool.hpp

Licensed with the MIT license, meaning you can use it in anything so long as you credit copyright. No warranty here!

---

Setup

- Initialize the thread pool
  To use the thread pool, we must first set it up within our Vulkan initialization functions
  - As our thread pool is a class, we must first create a new instance of it
    `ThreadPool myThreadPool;` // put in global scope
  - Then, we need to initialize it. Easiest way is to write an initialization function called immediately after all of our pipeline initialization is finished
    `void InitXXThreadPoolPrepare()` // declare and call after
  - Inside our initialization function, we need to supply the thread pool the number of threads we want it to have
    `void setThreadCount(uint32_t count)`
Setup

- Set some global variable equal to hardware the result of hardware concurrency (we'll use this in other places!)
  
  ```cpp
  int maxConcurrentThreads = std::thread::hardware_concurrency();
  ```

- Set the thread pool thread count equal to this number
  
  ```cpp
  myThreadPool.setThreadCount(maxConcurrentThreads)
  ```

---

Setup

- How many threads is ideal?
- This is a very subjective question that literally millions cannot agree on
- Good rule of thumb?
  - #include <thread>
  ```cpp
  std::thread::hardware_concurrency()
  std::thread::hardware_concurrency() == num_of_cores * threads_per_core
  ```
  - Max number of threads our system can run concurrently!
Setup

- Define a struct that will hold thread data (for thread pool results and reassembly)
  ```cpp
  struct ThreadData {
    VkCommandPool commandPool;
    std::vector<VkCommandBuffer> commandBuffer;
    std::vector<ThreadPushConstantBlock> pushConstBlock;
    std::vector<ObjectData> objectData;
  };
  ``

- Declare an instance of this struct
  ```cpp
  std::vector<ThreadData> myThreadData;
  ``

- Resize based on our thread count
  ```cpp
  myThreadData.resize(maxConcurrentThreads);
  ```

Setup

- Initialize our vector of thread data
  ```cpp
  for (uint32_t i = 0; i < maxConcurrentThreads; i++) {
    // Create a command pool for each thread
    ThreadData *thread = &myThreadData[i];
    VkCommandPoolCreateInfo cmdPoolInfo = vks::initializers::commandPoolCreateInfo();
    cmdPoolInfo.queueFamilyIndex = swapChain.queueNodeIndex;
    cmdPoolInfo.flags = VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT;
    vkCreateCommandPool(device, &cmdPoolInfo, nullptr, &thread->commandPool);
    thread->commandBuffer.resize(numObjectsPerThread);

    // Allocate our secondary command buffers
    VkCommandBufferAllocateInfo scbAllocateInfo;
    scbAllocateInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
    scbAllocateInfo.commandPool = thread->commandPool;
    scbAllocateInfo.level = VK_COMMAND_BUFFER_LEVEL_SECONDARY;
    scbAllocateInfo.commandBufferCount = thread->commandBuffer.size();
  }
  ```
- Initialize our vector of thread data (cont’d)

```c
// Allocate all of our secondary command buffers
vkAllocateCommandBuffers(device, &secondaryCmdBufAllocateInfo, thread->commandBuffer.data());

// Resize each thread data element’s internal # of push constant blocks and objectData structs
thread->pushConstBlock.resize(numObjsPerThread);  //
thread->objectData.resize(numObjsPerThread);
```

---

**Setup**

- Write secondary command buffer draw function(s)

```c
void mySCBDrawFunction(uint32_t threadIndex, uint32_t cmdBufferIndex, VkCommandBufferInheritanceInfo inheritanceInfo) {
    ThreadData *thread = &myThreadData[threadIndex];    // grab reference to where we
    ObjectData *objectData = &thread->objectData[cmdBufferIndex];  // grab reference to any object data we need to
draw

    VkCommandBufferBeginInfo vcscbbi;     // create info
    VkCommandBufferBeginInfo vcscbbi1;    // specify our inheritance info argument
    vcscbbi.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;
    vcscbbi.flags = VK_COMMAND_BUFFER_USAGE_RENDER_PASS_CONTINUE_BIT;
    vcscbbi.pInheritanceInfo = &inheritanceInfo;

    VkCommandBuffer cmdBuffer = thread->commandBuffer[cmdBufferIndex]; // write our SCB to our globalized thread data
    vkBeginCommandBuffer(cmdBuffer, &vcscbbi1)    // begin buffer!
    // write draw commands to cmdBuffer
    vkEndCommandBuffer(cmdBuffer);  // end buffer
}
```
Setup

- Write secondary command buffer draw function(s) (cont’d)
  
  ```
  // push constants to shader
  thread->pushConstBlock[cmdBufferIndex].mvp = matrices.projection * matrices.view * objectData->model;
  
  vkCmdPushConstants(cmdBuffer, pipelineLayout, VK_SHADER_STAGE_VERTEX_BIT, 0, sizeof(ThreadPushConstantBlock),
                        &thread->pushConstBlock[cmdBufferIndex]);
  
  // Set up viewport...
  // Set up scissor...
  // Bind pipeline & draw (bind vertex buffers/index buffers/etc.; make these global for easy access
  
  vkEndCmdBuffer(cmdBuffer);
  ```

Drawing

- Start the render pass
  
  ```
  vkCmdBeginRenderPass(primaryCommandBuffer, &renderPassBeginInfo,
                       VK_SUBPASS_CONTENTS_SECONDARY_COMMAND_BUFFERS);
  
  The VK_SUBPASS_CONTENTS_SECONDARY_COMMAND_BUFFERS flag states that we will
  construct our primary command buffer out of secondary command buffers

  May affect your program if you’re using multi-pass rendering!
  ```
Drawing

- Prepare SCB inheritance info
  Secondary command buffers must know what render pass they're working in, AND know what
  swap image they're going to render to inside the render pass

  Easiest way to do this is to supply them inheritance information on creation

  `VkCommandBufferInheritanceInfo vrpbi;`

  `vrpbi.framebuffer = swapchain[nextImageIndex]; // supply the current swap image we're writing to`

  `vrpbi.renderPass = currentRenderPass;  // the global render pass variable`

  /* Everything else should be the same */

Drawing

- Add jobs to our thread pool
  Using the given C++ implementation, call addJobs() on a particular thread in the thread pool
  Supply this function 1 argument: the function() that constructs your SCB drawing code

  ```cpp
  for (uint32_t thread = 0; thread < maxConcurrentThreads; thread++)
    for (uint32_t obj_instance = 0; obj_instance < numObjectsPerThread; obj_instance++) {
      myThreadPool.threads[thread]->addJob([=] { mySCBDrawFunction(thread, obj_instance, SCBInheritanceInfo); });
    }
  }

  Execute our jobs
  Call wait() on the thread pool when we've finished adding jobs to it

  myThreadPool.wait()```
Drawing

- Create vector to hold all of the work done by the SCBs
  std::vector<VkCommandBuffer> collectedCommandBuffers;

- Push our constructed SCBs onto the vector
  Refer to the work done by secondary command buffers (saved in the threadData vector) and use `push_back()`

```cpp
for (uint32_t t = 0; t < maxConcurrentThreads; t++) {
    for (uint32_t i = 0; i < numObjectsPerThread; i++) {
        collectedCommandBuffers.push_back(myThreadData[t].commandBuffer[i]);
    }
}
```

- Executing our SCBs inside a PCB
  We need to assemble all of our constructed SCBs as one PCB, so we can submit it to queue

```cpp
vkCmdExecuteCommands(primaryCommandBuffer, collectedCommandBuffers.size(), collectedCommandBuffers.data());
```

- End the render pass
  `vkCmdEndRenderPass(primaryCommandBuffer);`

- Submit to queue (same way we've been doing)
  We're done!
Destroy Stuff

- Clear out our thread pool
  We have to free all of the command buffers stored inside the thread data structs, as they contain data that needs to be freed

```cpp
for (auto& thread : myThreadData) {
    vkFreeCommandBuffers(device, thread.commandPool, thread.commandBuffer.size(), thread.commandBuffer.data());
    vkDestroyCommandPool(device, thread.commandPool, nullptr);
}
```

Multi-threaded Performance Benefits

- Desktop:
  - Reduced program load time (mileage may vary)
  - Vastly reduced render time (mileage may vary)
  - All that FPS!
- Mobile:
  - All of the above... probably
  - Reduced power consumption!
    - If we convert a mobile game to Vulkan:
      - Reduced CPU usage for the same amount of work
      - 10-15% power savings over OpenGL ES by most estimates
      - Play Pokemon GO for longer!
Some Thoughts...

- Multithreading can reduce loading and rendering time
- Not terribly complicated to get it working around existing code
- Though multithreading is a great resource, you don't have to multithread!
- Simple applications should be kept simple
  - Not all that much point in rendering Space Invaders with 8 cores and 16 threads
- Apps that rely on lots of high-definition assets will benefit from Vulkan even without multithreading

Some resources...

NVIDIA:

Android multithreading examples: [https://developer.nvidia.com/vulkan-android#samples](https://developer.nvidia.com/vulkan-android#samples)
Gameworks samples (ThreadedRenderingVk directory):
[https://github.com/NVIDIAGameWorks/GraphicsSamples/tree/master/samples/vk10-kepler](https://github.com/NVIDIAGameWorks/GraphicsSamples/tree/master/samples/vk10-kepler)

Sascha Willems's multithreading example:


Look here for more Vulkan examples, including: N-body collision, raytracing, shadow mapping, radial blur... etc.
Any questions?