GPGPU
Assignment 1

1 Introduction

The ever increasing number of applications that can fully use the huge parallelism offered by GPUs makes its understanding and mastery a crucial selling point in many industries. From fluid simulations to finances passing of course by machine learning, GPUs rule over real time and interactive applications as well as an ever increasing number of offline applications.

For its generality and versatility, Vulkan is the API of choice for modern computing. While it has less extensive libraries than CUDA, its integration to modern graphics pipelines is transparent and many leading company in the industry are making a switch to Vulkan.

This lecture will focus on typical techniques and optimizations that fully utilize the power of modern GPUs. The assignment topics will cover typical algorithms, often used in practice as we will see, that happen to be an ideal test bed for optimizations techniques.

This first assignment will walk you through the fundamentals of Vulkan. Creating high performance applications requires the knowledge of the GPU architecture and how to interact with it.

2 Parallel Programming

Parallel processing can differ widely in its nature. This course will focus on Single Instruction Multiple Data (SIMD) parallelism, being the one found in GPUs. Note that while Vulkan was build as a graphics API, it allows for task level parallelism by design (contrary to limited support in OpenGL), and mechanisms for this such as semaphores, fences and barrier are at the core of high performance applications.

Programming for SIMD requires a dramatic shift in programming. Most notably, only a subset of problems can be parallelized. Most require a reformulation of the sequential algorithm, sometimes involving additional steps.

Among the most important things to keep in mind:

1. Problem granularity: subdivide the problem to separate easily parallelizeable parts and the others. More often than not, a pipeline will have both sequential and parallel sections.
2. Minimal workload: GPUs are incredibly powerful, but only from a certain problem size. Below this threshold, memory transfer takes over on the computation time. Simple CPU parallelization can often be enough.
3. Memory is flat: The GPU accesses memory in 256 bytes long transactions. When reading an array of N integers (4bytes) vertically, it means each line needs its own transaction. Horizontally, and it needs only $\frac{4*N}{256}$. This is memory coalescing.
4. Bandwidth is narrow: Some applications are so bandwidth limited that a whole new area of computation emerged: in situ computation. I.e. instead of computing and saving the data, then loading it to display or use it, it is sometimes faster to compute it 'in situ', at the same time as the analysis. This is particularly the case when the amount of data generated is particularly large.
5. Optimize the bottleneck: Profiling is a large part of optimization on larger projects. Knowing what to optimize is half the work. [Amdahl's law, Gustafson's law] Sometimes, not the entire problem can be made parallel, and if the bottleneck is the sequential part to a significant degree, optimization resources should not necessarily be placed in the parallel part.

2.1 Historical context and legacy

GPUs started as very specialized hardware for graphics, and as such, graphics APIs were the standard way to make use of GPUs: OpenGL and DirectX. After 2001, GPU manufacturers added support for programmable shading, allowing game developers to adjust rendering algorithms to produce customized results. GPUs were equipped with conditional statements, loops, unordered accesses to the memory (gather and later scatter operations), and became moderately programmable devices. Such features enabled first attempts to exploit the graphics dedicated hardware for computing non-graphical tasks. The true revolution in the design of graphics cards came in 2007, when both market leading vendors, NVIDIA and ATI, dismissed the idea of separate specialized (vertex and fragment) shaders and replaced them with a single set of unified processing units. Instead of processing vertices and fragments at different units, the computation is nowadays performed on one set of unified processors only. Furthermore, the simplified architecture allows less complicated hardware design, which can be manufactured with shorter and faster silicon technology. Rendering of graphics is carried out with respect to the traditional graphics pipeline, where the GPU consecutively utilizes the set of processing units for vertex operations, geometry processing, fragment shading, and possibly some others. Thanks to the unified design, porting general tasks is nowadays less restrictive. Note the historical back-evolution: though highly advanced, powerful, and much more mature, modern GPUs are in some sense conceptually very similar to graphics accelerators manufactured before 1995.

Because it started as specialized graphics devices, the programming tools were created to support the development of graphic application, even after the unification of processing units. This led to suboptimal design. OpenCL and CUDA were primarily designed with the idea of general purpose computation and provide a very C-like syntax, making it convenient to programmer outside the graphics community.

Vulkan is a recent development in the domain of high performance computing and is quickly becoming the industry norm. Its interface exposes almost every step of the pipeline, allowing for fine grained tuning. Because it is very modular in nature, Vulkan can be very efficiently used for general purpose computation as well as high performance graphics. This allows for seamless interoperability and a unified framework. These reasons make it the framework of choice for an education that actually prepares to work both in the indus-

try and in research, both in graphics and general purpose computing.

2.2 Architecture of GPUs

Writing generic code for GPU that just outputs the correct answer is in general very suboptimal. Knowing the architecture of GPUs, how the computation is organized and how the data flows allows for impressive increase in throughput. This will be very apparent in the second task of this assignment. This section describes the architecture of modern GPUs. We will start by a global description of the GPU and will progress toward the individual compute unit.

First of all, GPUs are often described by the amount and type of dedicated memory they contain. The amount of memory can range from a few GB to just below 100GB. This memory is referred as global memory. The type of memory relates to the bandwidth of the connection between the CPU and GPU and its clock. GDDR6 supports memory clock between 1365-1770MHz with a bandwidth 336-672GB/s. The actual available bandwidth also depends on the different connectors (PCIe...).

2.2.1 Streaming Multiprocessor

The computing part of modern GPUs is made of several physically separated units: Streaming Multiprocessors (SMs). Low end models can have just one or two, while top end models can have more than thirty. Each SM has several caches:

1. *shared* or *local* memory for sharing data between threads
2. Constant cache
3. Texture cache
4. L1 cache to reduce latency to local or global memory

They also contain several thousands of registers that are split between threads. SMs contain a finite number (often 1024) streaming processors (SPs) which perform the computation. These SPs are grouped in *wavefronts* (AMD) of 64 SPs or *warps* (NVIDIA) of 32 SPs. These are operated by warp schedulers that can quickly switch them in and out to hide the global memory latency.

When code is placed on the GPU, called a kernel, it is dispatched as blocks. When computing the addition of two vectors of size 2^{19}, the computation is split into blocks of for instance 64 or 128 elements. These blocks are executed in any order, which allows for optimization from the scheduler.

2.2.2 Shared memory

Shared memory, or local memory, is convenient to use as a buffer on which to carry the computation instead of using directly the global memory. Shared memory is split among allocated blocks. Different blocks can not access each others' memory. Shared memory is split in 32 banks of 4 bytes each. The access to the banks is transparent from the API, but careless management can cause bank conflicts. We will cover bank conflicts in the second assignment.

Shared memory has on the order of 48 KB/threadblock. It has 15x the bandwidth and 20x to 40x less latency compared to global memory (Volta architecture). In general, it take 100-600 cycles for global memory read, and less than 10 cycles for shared memory. We use the name shared memory rather than local because the keyword to use it in GLSL is `shared`.

2.2.3 Streaming Processor

SPs contain execution cores for typical operations:

1. integer and single-precision floating point operations
2. double-precision floating point
3. Special Function Units (SFUs): sin, cos, exp, etc...

They can access registers directly.

2.2.4 Warps/Wavefronts

Warps are fixed groupements of SPs. They perform the exact same instructions at the same time. This implies that branching (e.g. if statement) can heavily impact performance. When there is different code paths within one warp, both paths are executed sequentially by the whole warp, and threads that don't participate are masked out.

2.2.5 Global memory

Global memory is very large but also very slow compared to all other memories. This is where all the data from the host is copied to. Memory bandwidth is oftentimes the main bottleneck of GPU applications. Memory is read from Global memory in transactions of 32, 64 or 128 bytes. This implies that memory accesses should be locally coherent, the dedicated word is coalescing.

2.2.6 Nomenclature confusion

Sometimes global memory is called shared memory, sometimes shared memory is called local, sometimes local memory is used for other kinds of memory. This is all very confusing, so we will try to use the same word for the same type of memory, i.e., those in the title of these subsections.

3 Vulkan set up

The Vulkan API provides no default values. This means every operation that CUDA or OpenCL usually does for you now needs to be explicitly created. This allows a great amount of flexibility and performance. We will first focus on the initialization of the core of the Vulkan application, then we will proceed to the creation of the compute pipeline.

Note that you should often look at the documentation yourself. It is good practice to know what the arguments we don't talk about can do. This document will often reference the name of enums, functions or structures for you to more easily check the documentation.

A visual help to understand where all these concepts fit in the pipeline is available in the supplemental material.

3.1 Initialization

The initialization contains a lot of boilerplate that you only need to know exists.

Instance

The instance holds every resources that we will create. It is possible to have several instances.

Validation Layers

Validation Layers help with catching errors occuring during the creation, utilization and destruction of the diverse structures and objects allocated. They can catch errors such as incoherent arguments, out of order destruction of objects and many others.

Physical Device

The Physical device list can be obtained using the instance. The command vulkaninfo displays all the properties of the device.

Logical Device

The logical device is created on a physical device. One can have several logical devices for one physical device. It is used for the creation of almost every resource from here on.

Queues

Queues are FIFO containers in which we submit command buffers to execute. There are different types of queue, mainly: transfer, compute and graphics. Note that a compute queue necessarily allows transfer, and a graphics queue necessarily allows compute too. Using separated queues for different concurrent work can be more efficient.

Command Pool

The command pool is created from a queue. It is an allocator for command buffers.

Command Buffers

Command buffers record a list of commands to be executed, and is to be submitted the queue corresponding to the command pool it has been created with. The CommandBuffer also has to be bound (during the registration of the commands) to the Pipeline, the DescriptorSets, the PushConstants and sometimes more.

3.2 Pipeline initialization

A visual description of the following concepts is available in the supplemental document.

ShaderModule

This is where the shader SPIR-V code goes. Vulkan only reads SPIR-V code. GLSL shader code needs to be compiled to it either prior to execution (glslc), or during execution (glslang). In the Shader code, we specify the inputs and outputs and give them integer identifiers called binding number. This will be used to bind buffers to them using Descriptors.

Buffers and Memory

Buffers are structures that are created with a specific size and usage. The usage informs on whether the buffer will be used as a destination or source for a copy, a uniform or storage buffer, see *VkBufferUsageFlags*. The memory is allocated depending on the memory type we want: device coherent or device only. Host coherent memory can be mapped and accessed directly, which allows us to fill the buffers with data from the host. Device only memory has better performance. Full list in the *VkMemoryPropertyFlagsBits*.

DescriptorSetLayout

The DescriptorSetLayout contains the description of the different buffers that will be used in the shader. It mainly contains an array of bindings describing individual buffers' types and flags with *VkDescriptorType*.

DescriptorPool

The DescriptorPool is an allocator for the DescriptorSet. Upon creation, it needs to be given information about the buffer types and number.

DescriptorSet

Finally, the DescriptorSet is created from the relevant pool and layout. We then need to bind our buffers to them referring again to their binding id.

PushConstants

This is a very convenient and versatile way to pass a small amount of data to the kernel. The data can change easily and is arguably a better way to pass small amounts of data than uniform buffers.

PipelineLayout

The PipelineLayout is needed to connect the Pipeline to the DescriptorSetLayout and the PushConstants.

Pipeline

Finally, the pipeline is created from additional shader structure and the PipelineLayout. The Pipeline is specific to one (3D) workgroup size, bound to one function in one shader file and to one (array of) DescriptorSetLayout.

3.3 This project

The framework exposes 3 main structures:

1. AppResources: structures that can be used for multiple tasks. Defined in `initialization.cpp/h`
2. TaskResources: each Task has its TaskResources, hosting temporary structures. Defined in `example_template.cpp/h`
3. Task: Where local structures such as buffers, shader parameters are hosted, as well as dedicated function that you need to fill. ex: `A1task1.cpp/h`

In this first assignment you will have to touch most files. This will not be the case for future assignments, as it is mostly about the setup. Note that all methods called from `device` such as `device.destroyPipeline(...)` are from the vulkan API, and methods called from `task` or `app` are created within our framework (which we encourage you to explore). The project uses a convenient way to time the performance of your tasks by using a `CSVWriter` class. It is very easy to use and allows you to setup your own files to easily plot the results with Python/R/Excel...

4 Vector addition

This task will guide you through the creation of a simple addition performed on GPU. We will see all the concepts discussed previously in action.

4.1 Setting Up Vulkan

Vulkan is well known for its vast amount of boilerplate code. Even though this is less true for compute only pipelines, we will not cover it in detail.

You can find in the initialization.cpp file the function initApp. This function prepares all the common resources needed for the tasks:

1. Instance
2. Validation Layers (in Debug mode only)
3. Physical Device
4. Compute and Transfer Queues
5. Logical Device
6. Compute and Transfer Command Pools

Knowing how to set everything up is not necessary for the assignments but if you are curious, you can read the code. The comments have been written assuming no background knowledge. You can also uncomment the line `printDeviceCapabilities(..)` and look inside about how to interact with the objects. Similar information can be obtained with the command vulkaninfo.

Once the App structure is populated, we can create our task.

4.2 Layout and Pipeline

In this section, we will prepare all the layouts. This is equivalent to telling Vulkan the overall shape of what will take place, so that we can put the data where it needs to be. Then we will bind it together in a Pipeline. In order, we will:

1. add buffer bindings
2. create the DescriptorSetLayout from the bindings
3. prepare the PushConstantRange
4. create the PipelineLayout from 2 & 3
5. create the ShaderModule from SPIR-V file
6. specify Specialization Constants (layout and values)
7. create the PipelineShaderStage from 5 & 6
8. assemble the Pipeline with the ShaderStage structure

The bindings serve to connect our buffers to the computation kernels, but only with id numbers. These bindings are all we need for the `DescriptorSetLayout`.

```
task.bindings.push_back(vk::
    DescriptorSetLayoutBinding(0, vk::
    DescriptorType::eStorageBuffer, 1U, vk::
    ShaderStageFlagBits::eCompute));
```

As we can see, the `DescriptorType` matches our situation: a storage buffer for a compute shader. The only parameter that is interesting to us is `binding`. We will need this ID in the shader (see supplemental material). Because this will almost always be the case, let's encapsulate this for readability. Modify the `addStorage(...)` function in `exercise_template.cpp`. Usage:

```
Cmn::addStorage(task.bindings, 0);
```

We have 3 buffers: input1, input2 and output, so we will need 3 bindings. Use this function 3 times with bindings 0, 1 and 2.

We now have a binding vector ready to become a `descriptorSetLayout`.

```
vk::DescriptorSetLayoutCreateInfo layoutInfo({},
    static_cast<uint32_t>(task.bindings.size()),
    task.bindings.data());
task.descriptorSetLayout = app.device.
    createDescriptorSetLayout(layoutInfo);
```

If we want to, we can take this clutter away by filling this function and simply calling it:

```
Cmn::createDescriptorSetLayout(app.device, task.
    bindings, task.descriptorSetLayout);
```

This object holds the shape and usage of our buffers, but not the buffers themselves. The structure that will directly hold them is the `DescriptorSet` and we will see them later.

This object has to be destroyed in `exercise_template.cpp`:

```
device.destroyDescriptorSetLayout(this->
    descriptorSetLayout);
```

As we said at the beginning, the computation is done on warps/wavefronts. This implies that the smallest unit of computation is the warp size, i.e. 32 or 64. The vectors that we want to process would have to be a multiple of 32 otherwise we will necessarily try to read/write unauthorized memory. The simplest way to deal with this is something like:

```
if(index > vectorSize)
  return;
```

This means we need a way to pass the size of the vector to the shader. We could use a dedicated buffer of size one to do this, but this is not very appropriate. A buffer is not supposed to be used for such small amounts of data, and we would like our value to be cached as much as possible since every thread will use it. Using a uniform buffer (read-only) is a possibility, but there is a more convenient alternative: PushConstants. We will also talk about Specialization Constants that are also a possibility, but behave in a different way.

PushConstants are easy to setup and can be modified on the fly without rebuilding everything (here we want to send the size of the buffers, so we would obviously need to resize our buffers...). This is the only object we need to create for our PushConstants. We will pass our data inside a `struct`, so we need to specify its size.

```
vk::PushConstantRange pcr(vk::ShaderStageFlagBits
    ::eCompute, 0, sizeof(PushStruct));
```

We now combine our `DescriptorSetLayout` and `PushConstant` together inside a `PipelineLayout`:

```
vk::PipelineLayoutCreateInfo pipInfo(vk::
    PipelineLayoutCreateFlags(), 1U, &task.
    descriptorSetLayout, 1U, &pcr);
task.pipelineLayout = app.device.
    createPipelineLayout(pipInfo);
```

This object has to be destroyed in `exercise_template.cpp`:

```
device.destroyPipelineLayout(this->pipelineLayout)
    ;
```

It is time to import the actual GPU code in Vulkan. We first create our GLSL file, and then compile it to SPIR-V. For the moment, the shader file is located in the shaders directory, and contains an empty `main()`. The GLSL file is automatically compiled thanks to some CMake magic, but you can also create your own shader file (and you should try out things) and compile it manually with `glslc`.

Fill the `createShader(...)` function in the `exercise_template.cpp`:

```
std::vector<char> cshader = readFile(filename);
vk::ShaderModuleCreateInfo smi({}, static_cast<
    uint32_t>(cshader.size()),
        reinterpret_cast<const uint32_t*>( cshader
            .data() ));
shaderModule = device.createShaderModule(smi);
```

We first read the file as an array of `char`. We need to cast it to `uint32_t` for the module to parse it. Note that the code is not yet put on the graphics card, this happens during the `Pipeline` creation. We also need to destroy the previous shader module in case we call this function multiple times. Simply call it this way:

```
std::string compute = "./build/shaders/"+file+".
    comp.spv";
app.device.destroyShaderModule(task.cShader);
Cmn::createShader(app.device, task.cShader,
    compute);
```

You could eventually create all the different possible shader modules in the first phase, and only use the relevant shader module here. For simplicity, we keep it this way. That being said, we encourage you to experiment with Vulkan as much as possible, and implement a more clever mechanism if you want to.

Since the workload is much bigger than the size of one streaming multiprocessor, we need to split the work into manageable chunks, called workgroups. We discussed that the smallest amount of dispatched work is the size of a warp, 32 or 64, so we preferably choose a workgroup size equal or higher than this.

It is interesting to see how the performance varies with the workgroup size, we will make it a parameter of our shader. The process to modify these values is a little cumbersome but allows for specialization constant of any nature. These constants allow for instance to change shared memory buffer size. Inside the `compute(...)` function, add:

```
std::array<vk::SpecializationMapEntry, 1>
    specEntries = std::array<vk::
    SpecializationMapEntry, 1>{ {{0U, 0U, sizeof(
    int)}} };
std::array<int, 1> specValues = {int(dx)};
vk::SpecializationInfo specInfo = vk::
    SpecializationInfo(CAST(specEntries),
    specEntries.data(), CAST(specValues) * sizeof(
    int), specValues.data());
```

In the first line, the three arguments `0U, 0U, sizeof(int)` are constantID, offset in bytes, size in bytes. The constant ID refers to the ID to refer in the shader. Specifying values for the workgroup uses a special value:

```
layout( local_size_x_id = 0) in;
//layout( local_size_y_id = 1) in;
//layout( local_size_z_id = 2) in;
```

But if you want to pass your own specialization constants that are not related to the workgroups, you use:

```
// ---------- C++ ----------
std::array<vk::SpecializationMapEntry, XX>
    specEntries = std::array<vk::
    SpecializationMapEntry, XX>{... {17, 16*sizeof
    (int)}, sizeof(int)} };
std::array<int, XX> specValues = { ... , int(
    cppArraySize)};
//{...}
// ---------- GLSL ----------
//{...}
layout(constant_id = 17) const int arraySize = 12;
//cppArraySize will overwrite the default value of
     arraySize
```

In our case, we use `dx` as our workgroup size. We will need to compute the number of groups depending on the workload size and workgroup size later.

We have created our specialization constants, and we need to bind it to our `ShaderModule`. This is done using the `PipelineShaderStageCreateInfo` structure:

```
vk::PipelineShaderStageCreateInfo stageInfo(vk::
    PipelineShaderStageCreateFlags(),vk::
    ShaderStageFlagBits::eCompute, task.cShader, "
    main", &specInfo);
```

The `ShaderStageFlagBits` would be different when creating a graphics pipeline: Vertex, Geometry, Fragment, etc. We are not using these features, but keep in mind that what we are doing now is quite similar for a graphics pipeline. Currently, there is no support for multiple entry points, so we need to use several files with each a "main" function, used as an entry point.

We finally bind this inside the `Pipeline`:

```
vk::ComputePipelineCreateInfo computeInfo(vk::
    PipelineCreateFlags(), stageInfo, task.
    pipelineLayout);
task.pipeline = app.device.createComputePipeline(
    nullptr, computeInfo, nullptr).value;
```

We should be able to run several times the compute function with different parameters, so we want to destroy the pipeline in case it was already created. Place this before the pipeline creation.

```
app.device.destroyPipeline( task.pipeline );
```

This object has to be destroyed in `exercise_template.cpp`, before the `PipelineLayout`:

```
device.destroyPipeline(this->pipeline);
```

4.3 Shader

Before actually sending the computation to the GPU, we need to write the code that will be placed on the GPU. This code will be written in GLSL, in the `shader/vectorAddition.comp`. GLSL behaves very much like C++. It also implements vector and matrices, we will see these later on.

We have a few things to do before writing the actual compute code:

1. Specialization Constants
2. DescriptorSets and Buffers
3. PushConstant

For the specialization constants, we only bound one, the workgroup size in the first dimension. In GLSL, the workgroup size has a special specialization constant as explained earlier:

```
layout( local_size_x_id = 0) in;
```

Note the 0 indicates the id of the constant as specified in the array of specialization constants.

The push constants always require a structure. The syntax is the following:

```
layout(push_constant) uniform PushStruct {
  uint size;    // total array size
} params;
```

Be careful with the types, if you put `float`, the data will *not* be casted to a `float` but the binary reinterpreted as `float`.

The buffers also use identifiers like specialization constant:

```
layout(binding = 0) buffer input1 {uint v1[];};
layout(binding = 1) buffer input2 {uint v2[];};
layout(binding = 2) buffer outBuf {uint v3[];};
```

Be careful to put the correct indices of the bindings corresponding to what you do in your computations.

Let's see how to access these buffers inside the main function:

```
int v = v1[gID];
```

Implement a simple addition, being careful not to write in unauthorized memory segments using an if statement.

4.4 Data

In our case, we need 3 buffers: 2 input buffers, 1 output buffer. Alternatively, we could use only 2 and write the result in one of the input buffers. It's a good test of understand to try to figure out what to change.

We start with Buffer creation. Note that there is a simple Buffer struct defined in utils.h:

```
struct Buffer{
  vk::Buffer buf;
  vk::DeviceMemory mem;
};
```

The `BufferCreateInfo` structure that holds the size of the buffer in bytes and its usage. Search the documentation for `VkBufferUsageFlagBits` for full list of usage. The main ones we will focus on are `TransferDst` or `Src`, `StorageBuffer`.

```
vk::BufferCreateInfo inBufferInfo({}, workloadSize
    *sizeof(int), vk::BufferUsageFlagBits::
    eTransferDst | vk::BufferUsageFlagBits::
    eStorageBuffer);
this->inBuffer1 = device.createBuffer(inBufferInfo
    );
```

The flag `eTransferDst` indicates that we will transfer data *to* this buffer.

In our case we will allocate memory to each Buffer individually. Note that it is possible to allocate one big segment of memory and split it directly using size and offset attributes. It is actually the recommended way for game engines or large projects, and the library VulkanMemoryAllocator is often used in this case. It is also not possible to resize a buffer once bound. Creating a new buffer is often the only possibility. If the buffers are recreated, the descriptorSets have to be recreated as well, which is why we only allow for static sizes in our framework. In more complex projects, dedicated functions would take care of the resizing operations, potentially taking care of synchronization issues, which is out of the scope of this assignment.

For the memory, we need to specify the memory properties. Search for `VkMemoryPropertyFlagBits`. The host coherent memory can be directly mapped to a host pointer and filled from C++. The device local memory is faster but requires a *staging* buffer, i.e. a temporary buffer accessible from the host whose only purpose is to get memory from the host to target buffer.

```
vk::MemoryRequirements memReq = app.device.
    getBufferMemoryRequirements(this->inBuffer1.
    buf);
vk::MemoryAllocateInfo allocInfo(memReq.size,
    findMemoryType(memReq, memoryTypeBits, vk::
    MemoryPropertyFlagBits::eHostCoherent, app.
    pDevice));
this->inBuffer1.mem = app.device.allocateMemory(
    allocInfo);
app.device.bindBufferMemory(this->inBuffer1.buf,
    this->inBuffer1.mem, 0U);
```

The `eDeviceLocal` means that we will be using a staging buffer. The lines 1 and 2

Since we need to create 3 buffers like this, we may want to have a `createBuffer` function. Have a look in `utils.cpp`, and fill the `createBuffer` function following the prototype. Then, create the three buffers `inBuffer1`, `inBuffer2`, `outBuffer` with this function. Note that the `BufferUsageFlagBits` for the output buffer differ!

Let's add these buffers to the `cleanup` function:

```
task.destroy( app.device );
auto Bclean = [&](Buffer &b){
    app.device.destroyBuffer(b.buf);
    app.device.freeMemory(b.mem);};
Bclean(inBuffer1);
Bclean(inBuffer2);
Bclean(outBuffer);
```

After having created our host data, we want to fill our two input buffers. It is done in three steps:

1. map device memory a host pointer
2. copy the host buffer data to this pointer
3. unmap the memory

```
void *data = device.mapMemory(this->inBuffer1.mem,
     0, inputVec.size() * sizeof(int),vk::
    MemoryMapFlags());
memcpy(data, inputVec.data(), static_cast<size_t>(
    inputVec.size() * sizeof(int)));
device.unmapMemory(this->inBuffer1.mem);
```

Because this will be a very common procedure, we have created a templated version of this function in `utils.h`. The exact same procedure can be applied to read the memory of the GPU, buf replacing the source and destination pointers in the `memcpy`. We have also a templated version of this function (C++17 allows us to not specify the `function<type>(args)` when it can be inferred).

```
fillDeviceBuffer(app.device, this->inBuffer1.mem,
    inputVec);
fillHostBuffer(app.device, this->outBuffer.mem,
    outputVec);
```

Fill the two input buffers with their respective data.

For the `DescriptorPool`, we will directly fill the `createDescriptorPool(...)` function, as now the creation pattern is quite apparent.

```
vk::DescriptorPoolSize descriptorPoolSize = vk::
    DescriptorPoolSize(vk::DescriptorType::
    eStorageBuffer, bindings.size() *
    numDescriptorSets);
vk::DescriptorPoolCreateInfo descriptorPoolCI = vk
    ::DescriptorPoolCreateInfo(vk::
    DescriptorPoolCreateFlags(), numDescriptorSets
    , 1U, &descriptorPoolSize);
descPool = device.createDescriptorPool(
    descriptorPoolCI);
```

`numDescriptorSets` is 1 by default. We will not need multiple `DescriptorSets` as we continue this assignment, but it is an optional parameter. Here we describe the kind of `DescriptorSets` we are about to create. The arguments of `DescriptorPoolCreateInfo` are compatible with an array, which means we can create different types of `DescriptorSets` with the same pool.

This object has to be destroyed in `exercise_template.cpp` before the `DescriptorSetLayout`:

```
device.destroyDescriptorPool(descriptorPool);
```

We now allocate the `DescriptorSets` that we will be able to bind to our buffers.

Fill the `allocateDescriptorSet(...)` function in `exercise_template.cpp`:

```
vk::DescriptorSetAllocateInfo descAllocInfo(
    descPool, 1U, &descLayout);
descSet = device.allocateDescriptorSets(
    descAllocInfo)[0];
```

We allocate 1 layout (the 1 in the first line). Since the function in line 2 returns an array, we take only the first (and only) element.

The `DescriptorSets` is allocated, we now bind it to our buffers. Fill the `bindBuffers`:

```
vk::DescriptorBufferInfo descInfo(b, 0ULL,
    VK_WHOLE_SIZE);
//      Binding index in the shader   V
vk::WriteDescriptorSet write(set, binding, 0U, 1U,
     vk::DescriptorType::eStorageBuffer, nullptr,
    &descInfo);
device.updateDescriptorSets(1U, &write, 0U,
    nullptr);
```

`VK_WHOLE_SIZE` specifies that we want to bind the entire buffer. It is possible to specify a size and an offset, allowing to split the buffer into different uses. We see that we again need to specify the descriptor type, this must be coherent with the `DescriptorSetLayout` when it is bound during computation. Technically, we can create an array of `WriteDescriptorSet` and use `updateDescriptorSets` only once.

After filling your buffers, the calls should look like:

```
// ||| fills the descriptorLayoutBindings  |||
Cmn::addStorage(task.bindings, 0);
Cmn::addStorage(task.bindings, 1);
Cmn::addStorage(task.bindings, 2);

// ### Create intermediate structures ###
Cmn::createDescriptorSetLayout(app.device, task.
    bindings, task.descriptorSetLayout);
Cmn::createDescriptorPool(app.device, task.
    bindings, task.descriptorPool);
Cmn::allocateDescriptorSet(app.device, task.
    descriptorSet, task.descriptorPool, task.
    descriptorSetLayout);
// ### DescriptorSet is created but not bound yet
    ###

// ||| Bind buffers to descriptor set |||
Cmn::bindBuffers(app.device, inBuffer1.buf, task.
    descriptorSet, 0);
Cmn::bindBuffers(app.device, inBuffer2.buf, task.
    descriptorSet, 1);
Cmn::bindBuffers(app.device, outBuffer.buf, task.
    descriptorSet, 2)
```

The `DescriptorSet` does *not* need to be destroyed or freed this is managed by the `DescriptorPool`

We have now completed the preparation for the task. In summary, we have:

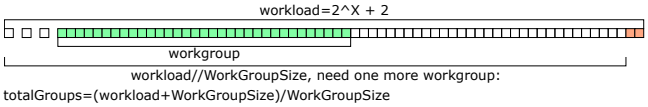
1. created a DescriptorSetLayout 4.2
2. prepared our PushConstant 4.2
3. created the PipelineLayout 4.2
4. created the ShaderModule 4.2
5. created Specialization Constants 4.2
6. created the Pipeline 4.2
7. created 3 Buffers 4.4
8. filled the 2 input Buffers with host data 4.4
9. created a DescriptorPool 4.4
10. allocated one DescriptorSet 4.4
11. bound our 3 buffers to the DescriptorSet 4.4

4.5 Dispatch

We now want to run the computation. For this we call the `dispatchWork` function with the number of groups in each dimension as parameters, as well as the value of the PushConstants.

In our case, we need to create the PushConstant and calculate the groupCount:

```
uint32_t groupCount = (workloadSize+dx-1) / dx;
PushStruct push{workloadSize};
```



And call the function that we will fill just after:

```
this->dispatchWork( groupCount, 1U, 1U, push );
```

In this section, we will finally carry out the computation. This will happen in `A1task1.cpp` in the `dispatchWork` function. We allocate and fill the command buffer. Then we submit it to the compute queue and wait for the result using a fence. We can then free the command buffer and get our result.

```
vk::CommandBufferAllocateInfo allocInfo(app.
    computeCommandPool, vk::CommandBufferLevel::
    ePrimary, 1U);
vk::CommandBuffer cb = app.device.
    allocateCommandBuffers( allocInfo )[0];
```

This allocates the the command buffer using the command pool that was created during the initialization. The command pool is called compute because it has been created from a compute *queue*. You can see how this works in `initialization.cpp` in the `getComputeAndTransferQueues` function.

We add our commands to the command buffer between a `begin` and `end` function:

```
vk::CommandBufferBeginInfo beginInfo(vk::
    CommandBufferUsageFlagBits::eOneTimeSubmit);
cb.begin(beginInfo);
// commands
cb.end();
```

The necessary actions we want to perform are:

1. send the current value our `PushConstant`
2. bind the `Pipeline`
3. bind the `DescriptorSets`
4. dispatch the workgroups

We also want to time the execution. This mechanism is a bit more advanced, you don't need to care about it for now, simply copy:

```
cb.resetQueryPool(app.queryPool, 0, 2);
cb.writeTimestamp(vk::PipelineStageFlagBits::
    eAllCommands, app.queryPool, 0);
cb.pushConstants(task.pipelineLayout, vk::
    ShaderStageFlagBits::eCompute, 0, sizeof(
    PushStruct), &pushConstant);
cb.bindPipeline(vk::PipelineBindPoint::eCompute,
    task.pipeline);
cb.bindDescriptorSets(vk::PipelineBindPoint::
    eCompute, task.pipelineLayout, 0U, 1U, &task.
    descriptorSet, 0U, nullptr);
cb.dispatch(dx, dy, dz);
cb.writeTimestamp(vk::PipelineStageFlagBits::
    eAllCommands, app.queryPool, 1);
```

The command buffer is filled with all the structures required for computing. Note the presence of the queries and timestamps. This is a mechanism used to obtain an accurate timing. We can now send the computation on the compute queue. We also prepare the fence so that our code can wait for the end of the computation.

```
vk::SubmitInfo submitInfo = vk::SubmitInfo(0,
    nullptr, nullptr, 1, &cb);
vk::Fence fence = app.device.createFence(vk::
    FenceCreateInfo());
app.computeQueue.submit({submitInfo}, fence);
vk::Result haveIWaited = app.device.waitForFences
    ({fence}, true, uint64_t(-1));
app.device.destroyFence(fence);
```

The parameters of submitInfo allow for more complex synchronisation mechanisms using `Semaphore`. This is very powerful and allows to explicitly wait for some operation to happen before running another one, e.g., waiting for data transfer or precomputation.

Let's not forget to free our command buffer at the end of the function:

```
app.device.freeCommandBuffers(app.
    computeCommandPool, 1U, &cb);
```

To get the data out, you only need to map the memory again, but reversing the order of the parameters in the `memcpy`. Your result is automatically checked by the `checkDefaultValues` function.

4.6 Evaluation

1. Understanding of the concepts: Device, Queues,... (2pts)
2. Setup and understanding of buffers and memory (2pts)
3. Correct computation of vector addition (3pts)
4. Using 4 different vector sizes and 6 different work-group sizes, compile the timings in appropriate graphs.(We recommend line graphs or any graphs convenient to show the differences) (2pts)

For the timings, repeat the same computation several times (10-50) and take the average. Make sure nothing else is heavily using the graphics card (video games, conference call...).

5 Matrix Multiplication

This task has almost the same preparation as the previous one. You can copy most of the preparation code, and appreciate the usefulness of the helper functions we created along the way.

That said, we will refine the copy of the buffers from host to device. We used host coherent buffers, which means they are always accessible by the host. This is not ideal in terms of performance. We will now use some device only memory. This changes the call to `createBuffer`, we now use `vk::MemoryPropertyFlagBits::eDeviceLocal` instead of `eHostCoherent`. This means we can no longer use the `fillDeviceBuffer` function.

The strategy using a staging buffer is as follows:

1. create a host coherent buffer of the same size (with its memory)
2. copy our data to it
3. copy the content of the staging buffer to the device local buffer
4. free the staging buffer and its memory

You already know how to do all the steps except for the third. Copying data from GPU to GPU is a command. As such it needs to be placed in a command buffer and submitted to a queue. A function handling most of it is written in `utils.h`, the `fillDeviceWithStagingBuffer` function. Read it carefully to understand what is happening.

The line that we do not know how to do is the `copyBuffer` call. Find the definition of this function and complete it:

```
vk::CommandBuffer commandBuffer =
    beginSingleTimeCommands(device, commandPool);
vk::BufferCopy copyRegion(0ULL, 0ULL, byteSize);
commandBuffer.copyBuffer(srcBuffer, dstBuffer, 1,
    &copyRegion);
endSingleTimeCommands(device, q, commandPool,
    commandBuffer);
```

The first line creates the command buffer and the last line places it in the transfer queue, submits and waits for the device to return.

The two other lines are straightforward: specify the size of the region to copy and add the copy command to the command buffer.

With this helper function, it is convenient to copy data to an existing buffer using a staging buffer. Add to the end of the `defaultValues` function:

```
fillDeviceWithStagingBuffer(app.pDevice, app.
    device, app.transferCommandPool, app.
    transferQueue, inBuffer, inputVec);
```

You can try it out on the first task!

5.1 Matrix Rotation

We want to implement a 2D matrix rotation. You can see the CPP implementation (not optimized, for readability) in the function `rotateCPU` in the file `A1task2.h`.

You will implement a simple naïve version, and a more optimized version using local memory. Refer to the first few sections for a reminder on local memory. Each thread will be responsible for writing one element of the matrix in its rotated coordinates.

In both cases, you will use the same `prepare` function, and only indicate the function to be used in the file in the `compute` function. You will need to create only one input buffer and one output buffer. Use device local memory for the buffers. You will need to update the Specialization Constants:

```
specEntries =  std::array<vk::
    SpecializationMapEntry, 2>{vk::
    SpecializationMapEntry{0U, 0U, sizeof(int)},vk
    ::SpecializationMapEntry{1U, sizeof(int),
    sizeof(int)}};
specValues = {int(dx), int(dy)}
```

You have to adjust the number of groups accordingly:

```
n_xgroups = (workloadSize_w + dx - 1) / dx;
n_ygroups = (workloadSize_h + dy - 1) / dy;
```

You also need to change the push constant's structure to pass the width and height of the input.

```
struct A1_Task2 {
    struct PushStruct
    {
        uint32_t size_x;
        uint32_t size_y;
    };
    //{...}
```

5.1.1 Naïve implementation

In the naïve version, we will read from global memory and write to global memory directly. Because we have to rotate the indices, we notice that we will either read or write once horizontally and once vertically, see fig. 1. It is inefficient and we will see in the optimized version how to improve it.

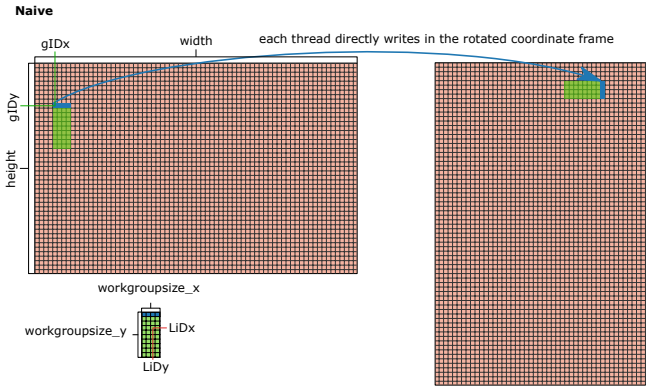


Figure 1: We don't use the local id just yet. Don't forget to protect the memory with an if statement in case the input is not a multiple of the workgroup size.

5.1.2 Optimized implementation

The optimization here involves local memory. Not that each workgroup will use the same shared memory. To use shared memory, declare it in your shader:

```
// below your the push constant
// either larger size than necessary
shared int[32*32] shared_ints;
// or just the right size
shared int[gl_WorkGroupSize.x*gl_WorkGroupSize.y]
    shared_ints;
```

The local memory is fast and efficient. The plan is to read from global memory horizontally (i.e. for neighbouring

gIDx), and write it in local memory. Then we can write in global memory horizontally as well, as the reading order doesn't matter in local memory.

There are multiple ways to tackle this problem. One could be to directly write in the transposed location in local memory, then read horizontally from local and write horizontally in global (it still requires some index magic!). Another would be to load the data as is, and rotate it using a second local memory buffer, then write it. Finally, copying as it is in local memory and writing in the transposed coordinates of both local and global memory. Figuring out the indices is a bit tricky, you may want to use another language, easier to debug to write out the algorithm.

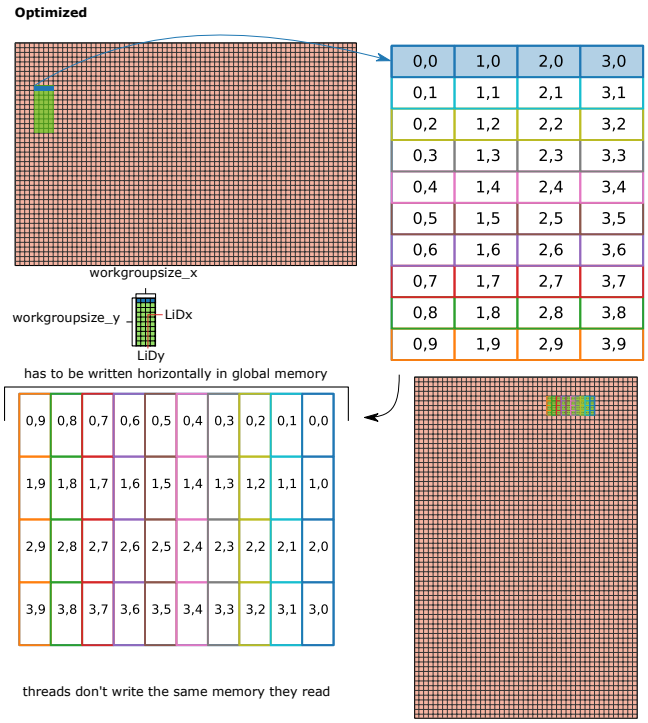


Figure 2: Visualization of the indices during the rotation. All the methods are not shown here. The workgroup ID corresponds to the top left item.

We suggest starting with the local memory filled like this:

```
// global IDs
uint gIDx = gl_GlobalInvocationID.x;
uint gIDy = gl_GlobalInvocationID.y;
// work group sizes
uint lSizeX = gl_WorkGroupSize.x;
uint lSizeY = gl_WorkGroupSize.y;
//  local IDs
uint lIDx = gl_LocalInvocationID.x;
uint lIDy = gl_LocalInvocationID.y;

// p.w and p.h are members of the push constant p
if(!(gIDx >= p.w || gIDy >= p.h))
    shared_ints[ lIDy * lSizeX + lIDx ] = v1[ gIDy
        * p.w + gIDx ];
barrier();
```

The barrier allows for a synchronisation point. Because thread A may read what thread B has written, we need to be sure that thread B has finished writing before reading.

5.2 Evaluation:

1. Implement the naïve algorithm. (4pt)
2. Implement the Optimized version. (4pts)
3. Rotation of non square matrices. (1pt)
4. Gather your timings into a graph with different total sizes and different workgroup sizes. Note: No 3D charts or bar plots. (1pt)

6 Building and Debug

6.1 Build

Because each OS has its specificities regarding dynamic loading of libraries (necessary to integrate RenderDoc), we can only support a limited number of platforms.

This project uses C++17, you will need a compiler that supports it. On Windows you can opt for MINGW or Visual Studio for instance. On Linux, you can use gcc or clang. A CMakeFile provides library management and automated recompilation of shaders, a recent version of CMake is therefore required.

In any case, you will need to download the Vulkan SDK. It contains everything you need to get started regarding Vulkan. The Vulkan SDK should be in the system environment variables: VK_SDK_PATH, and points to something like `C:/VulkanSDK/1.2.18`. This should be automatic though.

- Download and install the Vulkan SDK: https://vulkan.lunarg.com/
- check if installed: type `vulkaninfo --summary` in console
- Idem: RenderDoc https://renderdoc.org/
- Idem: C++17 compiler

6.1.1 Windows

1. Download CMake, either via chocolatey or direct download
2. Install it and check that it is in your PATH. I.e. fire up a terminal (WIN+X, I) and type `cmake --version`
3. `cd Assignment1`
4. `mkdir build && cd build`
5. Depending on your IDE, you may need to specify a generator. It can be `cmake -G"Unix Makefiles"..` or `cmake -G"MingGW Makefiles"..` or `"Visual Studio 16 2019"`...
6. Finally, use the compile command of the generator you chose. `make` for MINGW, directly in the interface for Visual Studio

6.2 RenderDoc

RenderDoc is a piece of software allowing you to see the order and parameters of the commands you've given to the Vulkan API. This can be interesting for debugging all kinds of problem. For instance, you can verify the values of the specialization constants. You can also directly read GPU memory buffers and output the data to check where things went wrong.

6.2.1 Checking Specialization Constants and Push Constants

- open RenderDoc
- Lauch Application tab
- fill Working Directory and Executable Path appropriately
- Launch button
- Pipeline State tab
- CS (last in the list)
- in the Event Browser (should be on the left) click on Dispatch(n1,n2,n3)
- in the Pipeline State tab, Uniform Buffers subwindow, click on the arrow at the end of the Specialization Constants line
- same for Push Constants
- in the Ressources subwindow, you can access the values of the entire buffers

Note: n1,n2,n3 correspond to the number of workgroups in each dimension.

To improve the usability of RenderDoc, we implemented a simple function `setObjectName(vk::Device d, <your vulkan object>, std::string name)`. As its name implies, it allows you to name any Vulkan component that depend on the `vk::device`. This name will be picked up by RenderDoc instead of "buffer1", "buffer2". Of course in this assignment, everything is relatively straightforward, but in bigger projects, this comes in very handy. Don't hesitate to name your objects!

6.3 Debugging

It is important to know how to debug a C++ application. Nowadays, most compilers are shipped with a debugger. Check how yours is set up and make it work. You should be able to place breakpoints easily and access the variables. Of course, check that you are in debug mode. The validation layers can only catch a small subset of mistakes, and obviously not C++ related mistakes.