Futhark Vulkan Backend

Steffen Holst Larsen (bsc373)

January 6, 2019

Abstract

This paper describes the effort, challenges, and limitations involved in the implementation of a Futhark compiler variant using the Vulkan API version 1.1 for compiling Futhark programs targeting GPUs. Compared to the existing OpenCL backend with the same purpose, the more modern Vulkan API could offer some performance benefits and may extend the scope of supported hardware for the Futhark programming language. The Futhark Vulkan backend comes as an extension upon the Futhark compiler pipeline with no changes to the structure, implementing both a host-to-device host code backend using the Vulkan API as well as an almost complete compiler from the intermediate kernel representation to SPIR-V shaders. Though most Futhark programs compile and run correctly using the Futhark Vulkan backend, there are still bugs and limitations residing in the implementation as well as volatile performance relative to that of the Futhark OpenCL backend tending towards a slowdown. However, now that the base implementation in place, with more work put into the Futhark Vulkan backend and potentially by giving the Vulkan API more time to further mature, the Futhark Vulkan backend may prove a valid alternative to the Futhark OpenCL backend in the future.

Contents

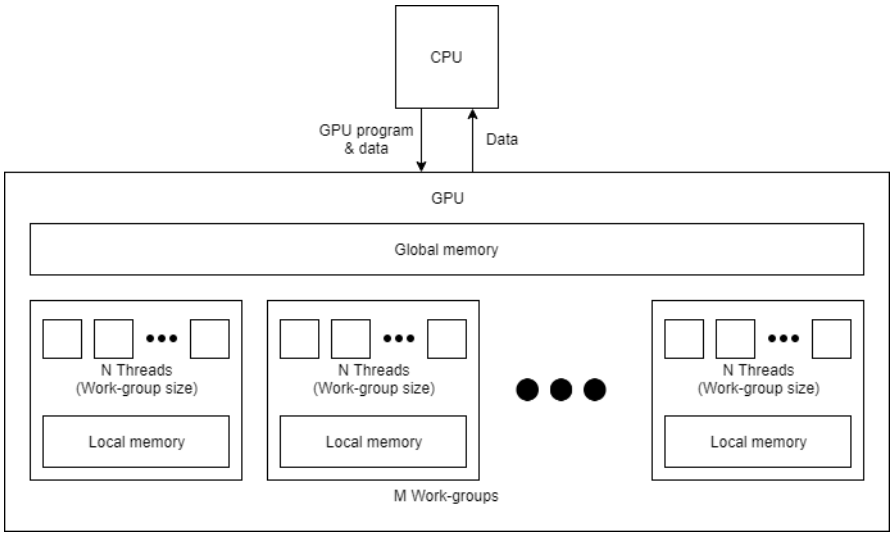

Figure 1: High-level view of the GPU computation model.

1 Introduction

Back in March of 2015, at the annual Game Developers Conference, a new generation of high-performance graphics and compute API was unveiled under the name Vulkan [14]. At that point in time, the Futhark compiler project, a collection of tools for compiling and profiling programs written in the data-parallel functional array language Futhark developed at the University of Copenhagen DIKU, was already two years into its development, so naturally the Vulkan compute API was not considered as the base backend for having Futhark programs target GPUs. Now that Vulkan has had time to mature, it is reasonable to explore the potential benefits of a Futhark compiler variant with a Vulkan backend.

Of particular interest in the implementation of the Futhark Vulkan backend is the Futhark OpenCL backend. To understand both backends better we need to understand the high-level GPU computation model as illustrated in Figure 1. A GPU acts as a co-processor to the CPU with its own memory spaces, of particular interest are the global memory space that all threads can access and the local memory space for each work-group. To do a computation on the GPU, the CPU transfers a program and the necessary data to the GPU. The GPU will then launch a number of work-groups with a number of threads each, the latter of which we will also refer to as the work-group size. All threads in all work-groups execute the same program and those in the same work-group can access the same local memory. As to make executions distinguishable, each thread can access a number of seemingly static values, such as the number of the work-group it is in and its thread number in that work-group. The CPU generally executes in parallel with the GPU, but is typically allowed to synchronize with

the GPU and transfer data from it to its own memory space.

Both the Vulkan API and OpenCL follows this computation model. The Futhark OpenCL backend has a number of different variants using different host languages, but for the purpose of the implementation of a Vulkan backend we will focus on the variant using C host code. The implementation of a Futhark backend using the Vulkan API is not without challenges, as the intermediate representation of Futhark programs targeting GPUs is inspired by OpenCL.

In the report we describe and analyze the implementation of the Futhark Vulkan backend for compiling Futhark programs to be run on a GPU. We first describe the Futhark OpenCL computation model in broad strokes, focusing on the imperative representation of GPU sub-programs called kernels and their general structure. We then shallowly go step-by-step through how the imperative representation is compiled into C using the Vulkan API, followed by a discussion of some challenges in the implementation and how they were handled. The programs that the Vulkan API runs on a GPU must be in the SPIR-V intermediate representation, so after discussing the use of the Vulkan API we shortly describe how we compile the code of a kernel body to SPIR-V, again followed by difficulties and limitations encountered throughout. Lastly we discuss how testing was conducted to ensure generated code quality, followed by a comparison in both compilation time and execution time between Futhark programs compiled with the Vulkan backend and the OpenCL backend variants of the Futhark compiler.

The source code for the Futhark compiler project with the Futhark compiler variant using the Vulkan API, namely the futhark-vulkan compiler, can be found in the vulkan-backend branch on the project Github page [10].

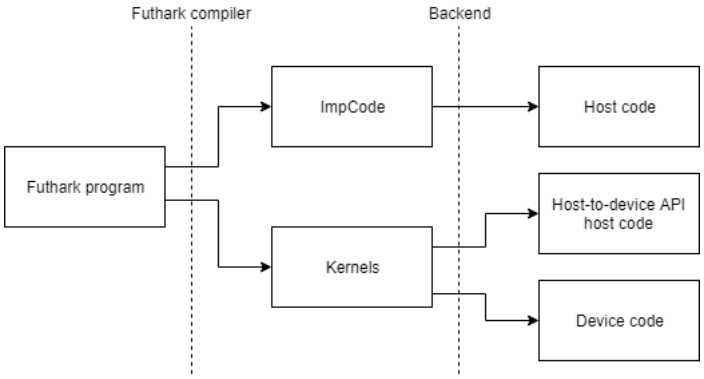

Figure 2: Code-generation steps of GPU targeting Futhark compilers.

2 Futhark OpenCL computation model

When compiling a Futhark program, the backend corresponding to the compiler in use is given an intermediate imperative representation of the program, known inside the compiler as ImpCode. When a compiler that uses a backend which targets a GPU, the ImpCode it is given contains a number of sub-parts intended to be executed on a GPU, known as kernels. Simplified, the code-generation of the Futhark compilers that target the GPU is summarized in Figure 2. In the original GPU targeting compiler, the host code is C with the device code being OpenCL C, a variant of C intended to be executed on the GPU. The host-to-device API host code in this case consists of C, primarily using calls to the OpenCL library for compiling and spawning the kernels as well as general book keeping. The generated OpenCL C program is placed inline in the C host program as a string, to be compiled by the program at runtime.

As an example of the generated ImpCode and kernels, consider a simple Futhark program add_all.fut for adding an input integer to all element of an input list of integers:

```
1 let main (a: i32) (xs: []i32): []i32 =
2   map (+a) xs
```

Using the futhark tool supplied with the Futhark compiler project, we can extract the ImpCode containing kernels resulting from compiling add_all.fut by running

```
1 > futhark --gpu --compile-imperative-kernels .\add_all.
    fut
```

which outputs:

```
1  Function main:
2    Inputs:
3      i64 xs_mem_size_3542
4      mem@device xs_mem_3543
5      i32 size_3524
6      i32 a_3525
7    Outputs:
8      i64 out_memsize_3549
9      mem@device out_mem_3548
10     i32 out_arrsize_3550
11   Arguments:
12     i32 a_3525
13     [i32,size_3524] at xs_mem_3543(xs_mem_size_3542)
           @device
14   Result:
15     [i32,out_arrsize_3550] at out_mem_3548(
           out_memsize_3549)@device
16   Body:
17     var group_size_3532: i32
18     group_size_3532 <- get_size(group_size_3531,
           group_size)
19     var y_3533: i32
20     y_3533 <- sub32 (group_size_3532) (1i32)
21     var x_3534: i32
22     x_3534 <- add32 (size_3524) (y_3533)
23     var num_groups_3535: i32
24     num_groups_3535 <- squot32 (x_3534) (group_size_3532)
25     var num_threads_3536: i32
26     num_threads_3536 <- mul32 (group_size_3532) (
           num_groups_3535)
27     var binop_x_3545: i64
28     binop_x_3545 <- sext_i32_i64 (size_3524)
29     var bytes_3544: i64
30     bytes_3544 <- mul64 (4i64) (binop_x_3545)
31     var mem_3546: mem(@device)
32     mem_3546 <- malloc(bytes_3544)@device
33     kernel {
34       groups {
35         num_groups_3535
36       }
37       group_size {
38         group_size_3532
39       }
40       local_memory {
41
```

```
42        }
43        uses  {
44          scalar_copy(size_3524, i32), scalar_copy(a_3525,
                i32),
45          mem_copy(xs_mem_3543, xs_mem_size_3542), mem_copy
                (mem_3546, bytes_3544)
46        }
47        body  {
48          var wave_size_3551: i32
49          var group_size_3552: i32
50          var thread_active_3553: bool
51          var gtid_3530: i32
52          var global_tid_3537: i32
53          var local_tid_3538: i32
54          var group_id_3539: i32
55          global_tid_3537 <- get_global_id(0)
56          local_tid_3538 <- get_local_id(0)
57          group_size_3552 <- get_local_size(0)
58          wave_size_3551 <- get_lockstep_width()
59          group_id_3539 <- get_group_id(0)
60          gtid_3530 <- global_tid_3537
61          thread_active_3553 <- slt32 (gtid_3530) (
                size_3524)
62          var x_3540: i32
63          var res_3541: i32
64          if thread_active_3553 then {
65            x_3540 <- xs_mem_3543<i32@global>[mul32 (
                  gtid_3530) (4i32)]
66            res_3541 <- add32 (a_3525) (x_3540)
67          } else {
68            skip
69          }
70          if thread_active_3553 then {
71            mem_3546<i32@global>[mul32 (gtid_3530) (4i32)]
                 <- res_3541
72          } else {
73            skip
74          }
75        }
76      }
77      out_arrsize_3550 <- size_3524
78      out_memsize_3549 <- bytes_3544
79      out_mem_3548 <- mem_3546 @@device
```

Of particular interest is the kernel definition in the lines 31-74. Before encountering the body of the kernel we have 4 parameter blocks; groups, group_size,

local_memory, and uses. The groups parameter specifies the number of work-groups to spawn, whereas the group_size parameter specifies the work-group size. The local_memory parameter specifies the amount of local memory to allocate for each work-group in bytes. Lastly we have the uses, which is a collection of data structures, each being one of three kinds; memory-use, scalar-use, and constant-use. A memory-use specifies a block of memory that should be visible to the kernel when executing, whereas scalar-uses and constant-uses specifies values of non-composite type that will be visible to the kernel body, but will remain constant. The latter two may seem equivalent, but whereas the value of a constant-use can be determined early in the execution of a program, a scalar-use must be evaluated just before the execution of a kernel.

The body of the kernel follows close to the same structure as the ImpCode encapsulating the kernel, with some additional operations, e.g. reading values such as the global identifier of the executing thread and the work-group identifier. Additionally, kernels may contain expressions such as memory barriers and atomic operations, which are not illustrated in our example. Therefore, since OpenCL C is a variant of C, the code-generator used for the host code can be reused for the OpenCL C code-generator, with only little extension.

3 Vulkan backend

At the creation of Futhark, Vulkan had not been announced, thus OpenCL was the obvious choice of backend for the Futhark kernels. However, OpenCL is not supported out-of-the-box on platforms such as Android and iOS, whereas Vulkan is supported by Android 7.0 and later[1] and is supported on iOS through the MoltenVK implementation [6]. Additionally, a Vulkan backend for Futhark could offer performance enhancements, e.g. through lower launch overhead. Notice however that the Vulkan backend is not intended to replace the OpenCL backend, but is rather intended to offer more choices to the Futhark programmer.

Due to the modularity of the code-generation steps, as illustrated in Figure 2, the addition of a Vulkan backend using C host code requires only the implementation of a code-generator for the kernels. For this the GPU program will be written in SPIR-V, which will be further discussed in Section 4, and the host-to-device host API code will be in C using the Vulkan API.

Relative to the OpenCL API, the Vulkan API is much more explicit, requiring more steps for setting up a context, compiling shaders, and general book keeping. The context setup of the Vulkan backend can be summarized as:

1. Create a VkInstance containing information about the application. This will initialize the Vulkan library [13, ch. 3.2]. Notice that we use version 1.1 of the Vulkan API, as it has the device extensions VK_KHR_16bit_storage and VK_KHR_storage_buffer_storage_class promoted, i.e. they are enabled by default. These extensions are used to allow fine-grained access to the input memory objects [13, appendix C].

2. Using the newly created instance we find a suitable physical device by iterating through the available devices, selecting the first device that support version 1.1 of the Vulkan API [13, ch. 4.1]. Currently there is no way of selecting specific devices when using the Futhark Vulkan backend, but it should be easy to implement in a similar way to that of the OpenCL backend by extracting the names of the devices through the device properties.

3. Create a logical device, represented by the VkDevice handle. When creating a logical device, we enable the VK_KHR_8bit_storage device extension which, akin to VK_KHR_16bit_storage, allows us to access input buffers by an 8-bit stride. Additionally we enable the shaderInt16, shaderInt64, and shaderFloat64 device features, allowing for the used of 16-bit and 64-bit integers as well as 64-bit floating point numbers in shaders [13, ch. 4.2]. Notice that support for 8-bit integer operations is at the time very limited, which will be discussed further in Section 4.2.

4. To find a command queue, we first need to find a suitable queue family, which is any queue on the physical device that support compute operations. Notice that transfer operations are implicitly enabled on queues with the compute-bit, thus though we need transfer operations we need not check for it. Using this, we get the corresponding command queue,

which will be where we submit the work to, be it kernel shaders or device-side memory copy operations [13, ch. 4.3].

5. Lastly, we create a command pool from the queue family, which will be used for allocating command buffers, and a single descriptor pool.

Notice that, unlike OpenCL, the Vulkan API is not exclusively a GPU compute framework but is also intended on graphics processing. This is likely why the Vulkan API is made as modular as it is, requiring the enabling of extensions and capabilities if needed, such as the device extension VK_KHR_8bit_storage. The optional extensions allow the Vulkan to be implemented on devices that do not necessarily have a use for some of the extension, e.g. they may focus primarily on graphics processing which may not have a use for 8-bit integers. Additionally, the extensions allows the Vulkan API to grow incrementally, potentially promoting the extensions in new versions, requiring hardware to support the extensions to be compliant with that and later versions of the API. Consequently any extensions enabled may exclude devices from using the Futhark compiler with the Vulkan backend, at least until the extension is added to the Vulkan implementation on the device.

In addition to a general context, each kernel shader has context with objects that needs to be initialized early. These objects are:

- A VkShaderModule that in turn contains the shader code of the kernel and the entry point.
- A VkDescriptorSetLayout describing the layout of the descriptor set, i.e. an overview of the input parameters of the kernel.
- A VkPipelineLayout used for creating pipelines. Created pipelines are cached to avoid expensive recompilation of shaders.

When the program seek to execute a kernel it will first find a suitable command queue, create the pipeline, and allocate a descriptor set. It will then update the descriptors and bind both the descriptor set and pipeline to the command buffer. Lastly it records a dispatch of the kernel into the command buffer with the corresponding work-group sizes [13, ch. 5]. The command buffer will then be submitted to the command queue and eventually executed.

The Vulkan API does very little cleanup implicitly, so most objects explicitly created or allocated through the API must be kept until we can safely dispose of them.

3.1 Descriptors and descriptor pools

A descriptor is a representation of a device-local resource, which are grouped into sets allocated from a descriptor pool. We will use a single descriptor set for each kernel, but we must not reuse a descriptor sets before it is done being used. This means that, if we are to run a kernel multiple times, we cannot use the same descriptor set for them all unless we can ensure that the other instances

of the kernel have completed execution before we submit the next. We could do this by synchronizing, but this comes with an unwanted overhead.

Instead, we allocate the descriptor sets from the descriptor pool by need. This is generally inexpensive as the pool is pre-allocated, but it does mean that we cannot know the needed number of descriptors before we actually need them. To accommodate this, we let the size of a descriptor pool be the size of each bulk allocation of command buffers. Whenever we need more command buffers we can allocate another descriptor pool, thereby when the command buffer at index i is needed, we know that there must be a free descriptor in descriptor pool $\lfloor \frac{i}{s} \rfloor$, where s is the number of command buffers allocated in bulk. However, since descriptor sets may vary in size and pools need the total number of descriptors specified at creation, we need to to allocate the maximum number descriptors of the descriptor sets for each descriptor set in the pool. This will often infer over-allocation, which should generally not pose a problem as the Vulkan API specifies the maximum allowed number of descriptors of the type we need to be at least 24^8 and, depending on the implementation on the device, descriptors should be fairly lightweight [13, 30.2.1]. It does however allow for the rare case where the whole descriptor pool occupied by descriptors for the kernel with the largest descriptor set.

Notice that we cannot just bind a single descriptor set to a command buffer as different kernels may need different descriptor set layouts, which are specified when allocating a descriptor set, thus reallocation is needed.

3.2 Command buffer reuse

A command buffer follows a well-defined lifecycle [13, ch. 5.1] wherein a call to vkBeginCommandBuffer will bring it into the Recording state, resetting any previously recorded commands. Therefore, if a command buffer is currently in execuring or waiting in the command queue, we must not record over it. Since a kernel may run multiple times, it is therefore not possible to just assign a command buffer to each shader.

Instead, we keep an array of command buffer contexts, containing a single command buffer and additional objects, such as scalar-use memory blocks, that benefit from being associated with it. Since only memory-use parameters are allocated by the C backend and given that we allocate scalar-uses from the same heap, we know that whenever a command buffer is free, any scalars it holds can be freed. Therefore, scalars are kept with the command buffer. The same goes for descriptor sets.

When a program seeks to execute a kernel, an available command buffer must first be found. Initially, a number of command buffers are allocated in bulk and if at some point in the execution of the program all command buffers are occupied, we extend the allocation accordingly. A command buffer context also contains a fence that is signaled when the command buffer is done executing. Notice that a signaled fence is not enough to conclude whether a command buffer is available, as it may hold ownership of some still active memory (see Section 3.3). Therefore, a command buffer is available only if its fence has been signaled and

it can be determined to have no active ownership. Whenever a command buffer is deemed available, any objects bound to it will be appropriately disposed.

3.3 Ownership synchronization

Whenever the host code accesses device memory, typically either to write data to device-visible space or to read the result of a kernel, it first maps the memory block to the host's address space, ensuring a coherent view of the memory at host-level until unmapped [13, ch. 10.2.1]. However, mapping to host memory does not synchronize with any kernel that accesses it, thus we introduce the notion of ownership.

We say that command buffer owns a memory block only if its descriptor set contains the last descriptor that was assigned the memory block. Though command buffers submitted to the same queue may be executed out of order, fences are first signaled after the corresponding command buffer and any command buffers submitted before it has finished executing [13, ch. 2.2.1]. Therefore, it suffices to synchronize with only the owner of a device memory block before accessing it, rather than waiting for the whole queue to finish.

The ownership strategy is implemented by including, in each device memory reference object, a reference to the command buffer that owns it and a flag indicating if it has no owner. Whenever another command buffer needs to acquire ownership the reference is changed. This is not enough however, as a finished command buffer may be reused by another kernel. To prevent this from happening, each command buffer maintains a count of device memory blocks that it owns, thereby a command buffer must not be reused unless it owns no device memory blocks.

Alternatively, we could keep references in each command buffer back to the deivce memory blocks they own and release ownership when the command buffer's fence is found signaled, thereby allowing for reuse earlier. This would however require more advanced data structures, such as linked lists, to allow for transfer or freeing of device memory blocks, as it would create irregular structure if an simple array is used.

3.4 Specialization and pipeline caching

As described earlier we have three groups of kernel uses, namely memory-uses, scalar-uses, and constant-uses. Whereas memory-uses are fully dynamic, that is the content may change at any time, the scalar-uses and constant-uses are both, to some extent, constant. The distinguishing feature of the latter two is that scalar-uses are constant throughout the execution of a kernel, but may change in the host code, whereas constant-uses remain constant throughout the execution of the program. As an example, consider the Futhark ImpCode example from Section 2 where we have memory-uses for both an input and an result memory block, as well as scalar-uses for the size of both memory blocks and the integer value to add to each element of the input memory. The scalar-uses depend on the input to the main function, so they are not ensured constant until they

reach the kernel, where the memory block has a specific size and the integer to add to the elements will not change. Though scalar-uses could potentially represent the values of constant-uses, the knowledge that the values will remain constant throughout the execution of a program allows shaders to be optimized when compiled at runtime.

Instead of leaving it to the programmer to create placeholders inside the shaders and keeping track of their offset, the Vulkan API allows for shader specialization when creating pipelines [13, ch. 9.7]. Using this we specialize constant-uses, the lock-step width, and the work-group dimensions. Notice that specialization is the primary way to specify work-group dimensions at runtime in the Vulkan API, which is needed for most kernels.

Specialization and compilation is expensive, so created pipelines are cached in a dynamic array with entries containing the pipeline and a key containing the specialization values used to create the pipeline, which is in turn used to check if a cache entry can be reused. A pipeline cache is dynamic and is only emptied at the end of the program. Typically, only a single cache entry is ever needed as the specialization constants are rarely changed between executions of a kernel. Therefore, the original pipeline cache only contained a single pipeline, replacing the pipeline on a cache miss. This strategy was replaced with the dynamic cache as otherwise we would have to destroy the old pipeline after replacement, which in turn requires that it is not in use by another kernel at the time, thus either it would have to wait for the other kernels to finish or defer the deletion to some other time. Notice that scalar-use parameters are not included as a specialization value, though they remain constant through the execution of a kernel, as they are likely to change between kernels, thus would drastically increase the chances of cache misses.

The Vulkan API does offer a pipeline cache object, allowing for better sharing of pipeline construction results [13, ch. 9.6]. This was experimented with, but resulted in worse performance than the current pipeline cache, likely due to some pipeline destruction and recreation overhead. The Vulkan supplied pipeline cache might still prove useful in the future however, as it could potentially reduce the work needed in the rare case where we need to create multiple pipelines in the cache.

4 Generating SPIR-V shaders

The Vulkan API uses shaders written in SPIR-V, an intermediate representation in bytecode form for representing shaders in multiple Khronos APIs, such as Vulkan and OpenGL [7, ch. 1.1]. Therefore, unlike the Futhark OpenCL backend, the Futhark Vulkan backend cannot reuse any already existing code-generator, so a new compiler from kernel ImpCode to SPIR-V is needed. Before getting into the implementation of such compiler, we first need to understand the basic structure of SPIR-V better.

Each instruction in SPIR-V consists of a number of 32-bit words, starting with a single word wherein the 16 high-order bits containing the number of 32-bit words in the instruction, including the first word, and the 16 low-order bits containing the opcode of the instruction. The opcode defines the effect of the adjacent words, often with the first of these being the identifier to contain the result of the instruction [7, ch. 2.3]. SPIR-V follows SSA form, thus there is only ever a single instruction with any given identifier as the result, but does also allow for defining variables, enabling the use of any number of loads and stores to them throughout execution.

A shader module in SPIR-V must follow a strict logical layout [7, ch. 2.4] and must include a header starting with the "magic number" 0x07230203 [7, ch. 3.1]. Additionally, the header must include the SPIR-V version number, an identifier of a generator registered with Khronos [4], and an upper bound on the identifiers in the shader. Notice that Futhark is not a registered application in the Khronos registry, the Vulkan backend uses the 0 identifier reserved for general use. Since SPIR-V must be on SSA form, the compiler keeps a moving upper bound on the identifiers in the shader. Whenever an instruction with a return value is inserted into the generated code, the result will be assigned the value at the upper bound before incrementing the bound by 1.

With this in mind, we can in broad strokes consider how the compiler constructs these shaders. The body of a kernel consists of a series of imperative operations, which is primarily comprised of control structures, such as if-then-else statements and loops, variable declarations, and assignment operations. A SPIR-V shader must be comprised of basic blocks, i.e. a sequence of instructions with a single entry point and a single exit point, so compiling an ImpCode control structure is done by first compiling any expressions needed for determining the path to be taken and then inserting basic blocks containing the corresponding compiled structure body, referenced by a branch instruction. As an example, consider again the ImpCode from Section 2. On the lines 70-74 of the ImpCode example we have an if-then-else statement, which compiled to SPIR-V we get the shader snippet:

```
1 %66 = OpLoad %bool %27
2       OpSelectionMerge %69 None
3       OpBranchConditional %66 %67 %68
4 %67 = OpLabel
5 %70 = OpLoad %uint %33
```

```
6  %71 = OpLoad %uint %28
7  %72 = OpIMul %uint %71 %uint_4
8  %73 = OpUDiv %uint %72 %uint_4
9  %74 = OpAccessChain %_ptr_StorageBuffer_uint %21 %73
10         OpStore %74 %70 None
11         OpBranch %69
12 %68 = OpLabel
13         OpBranch %69
14 %69 = OpLabel
```

Notice that this is a disassembled representation of the SPIR-V shader, using the spirv-dis program from the SPIR-V tools by the Khronos group [8]. The first instruction is the condition of the statement compiled, i.e. the load of the variable thread_active_3553, which resides in the variable with identifier 27. The actual if-then-else statement starts immediately thereafter by the conditional branch instruction followed by the two basic blocks corresponding to the true-branch and the false-branch on lines 4-11 and 12-13 respectively. In the false-branch we have only a skip-operation which is simply omitted when compiled, so the corresponding basic block ends immediately. This empty basic block will be optimized out when compiled by the Vulkan API. The true-branch contains the single write to memory in its body. To make a write to memory, first the expression and the index are compiled. Then an access chain is made to create a pointer into the memory block, which is then used to store the result of the expression through. The astute may have noticed that the index is first multiplied by the constant 4 followed by the result being divided by 4, which is caused by the index in ImpCode being in bytes but the access chain needs to be indexed by an element offset. The Vulkan compiler should optimize this discrepancy accordingly. Notice that the compiled control structure results in an open basic block, which will either be ended by a new control structure or by the end of the shader entry point function.

Compilation of ImpCode expressions to SPIR-V is done by traversing the expression tree, compiling the leafs to stand-alone instructions and propagate the resulting identifier of the node operations up towards the root, where the final identifier will be assigned to the result of the whole expression. This ensures that precedence and order is maintained.

By the logical layout of a SPIR-V shader, declaration of variables inside the entry point function must happen at the very start of the first basic block. The structure of a kernel may not respect this, thus the kernel ImpCode to SPIR-V compiler incorporates a variable declaration pass which finds all inline variable declarations such that it can declare them early.

After compiling the body of the kernel, additional information is added to the start of the SPIR-V shader, following the logical layout. This includes information such as input memory descriptor bindings, type definitions, constants, etc.

4.1 GLSL extension

Many arithmetic operations in the Futhark imperative representation have a direct mapping to a SPIR-V instruction of equivalent semantics, but not all. Examples of missing operations are square-root and the exponential function. To extend the toolkit of operations available, we enable the GLSL extension, which adds a plethora of these missing mathematical operations [17]. Notice that the GLSL extended instruction set for SPIR-V must be supported in all implementations of the Vulkan API version 1.1, so we do not exclude any hardware by enabling this extension [13, appendix A].

The imperative representation of a kernel has a static set of functions that it may call. These functions are on either 32-bit floating point numbers or 64-bit floating point numbers, identified by the bit-size suffixed in the name of the function. The functions on 32-bit floating point numbers mostly have a direct mapping to an instruction in the GLSL extended instruction set, but some of these GLSL instructions support only 32-bit floating point numbers and not 64-bit floating point numbers as needed. To mitigate this we reduce the precision of the operands to 32-bit floating point precision, allowing the use of the 32-bit instruction. This is of course not optimal, but there are no alternatives without enabling another extended instruction set, such as the OpenCL extended instruction set, which may alienate some hardware as it is not a required extended instruction set and would arguably defeat the purpose of the Futhark Vulkan backend.

4.2 8-bit integers

Though the Vulkan API uses SPIR-V shader modules to describe programs to be executed on the a GPU, it does not support all of SPIR-V. For the Futhark Vulkan backend, this becomes relevant in the limited support for 8-bit integers, where like with 16-bit and 64-bit integers there is a capability in SPIR-V intended for enabling the use of 8-bit integers and operations on them, but this capability is not on the list of SPIR-V capabilities supported in the Vulkan API [13, appendix A].

This is partially patched by enabling the VK_KHR_8bit_storage device extension together with the SPIR-V StorageBuffer8BitAccessCapability capability in each shader module, allowing the shaders to read and write to 8-bit input arrays [13, pg. 1483]. This extension does not explicitly require 8-bit operations or 8-bit variables to be implemented in the hardware and could potentially be implemented in hardware by just allowing loads and stores to and from memory, requiring casting to a value of larger bit-size. On the hardware that the Vulkan backend has been tested on, enabling the extension allows for general 8-bit usage, be it variables or operations. Notice however, that this extension may exclude some hardware from using the Vulkan backend, even though they support Vulkan, as it is not currently a required extension. However, since it is an extension made by the Khronos group, it has good chances of being promoted in a future version of Vulkan.

With the Vulkan 1.1.95 specification released in December of 2018, a VK_KHR_shader_float16_int8 device extension was added to the specification, which enables 8-bit integer and 16-bit floating point arithmetic operations in SPIR-V shaders [13, pg. 1523-1524]. This will move the Futhark Vulkan implementation out from the undefined behavior that it is currently utilizing when using 8-bit integer arithmetics. At the time of writing, version 1.1.95 of Vulkan is however not supported by the LunarG SDK and the extension is only supported by 0.36% devices registered in the Vulkan hardware database [11], but in time this extension should be enabled in the Futhark Vulkan backend.

4.3 Pointer conversion

Another example of a SPIR-V capability that would be useful to the Futhark Vulkan backend but is not supported by the Vulkan API is the Addresses capability, which allows for the use of physical addressing when using pointers. Most noticeably the Addresses capability adds the ability to convert a pointer to and from an integer representing the physical address it points to.

In the imperative representation of a kernel, memory blocks are generally not declared with an element size, but rather with a total size and may at any time be accessed with any non-composite element type. This is done to allow for tight packing of different data in the same memory block to save memory, which is often done in a work-group's local memory. Since SPIR-V is strongly typed, the type of the elements in arrays must be declared explicitly so we need to determine an element type at compile-time. Given that SPIR-V does not allow for converting neither arrays nor pointers, we cannot just convert the access type by need. Instead, we add minimum access size pass before compiling the body of a kernel where we determine the size of the smallest element access made to a memory block in the kernel body. The memory blocks are defined to be an array of an integer of the minimum access size found, given that integers may have any of the access sizes. If an array with minimum access size of a is then read as an array with elements of size b, where we must have that $a < b$, exactly $\frac{a}{b}$ integers of size a are read from the array, starting from the given offset. They are then converted into integers of size b, bit-shifted, combined using integer addition, and then converted to the wanted type. Likewise, when writing an element to an array, the value to be written is split up into parts the size of the minimum access size of the target array, and each of these parts are then written. Notice that all scalar types have a byte-size the power of two, thus any access type will be a multiple of the minimum access size, so there must be an exact number of element to read or write.

As an example of this, consider a memory block of 5 bytes accessed as an array of 32-bit floating point numbers and as an array of 8-bit integers. The minimum access size of the memory block is 1 byte, so reading a 32-bit floating point number at offset 1 would be done as follows:

1. Read 8-bit integers at offset 1, 2, 3, and 4.
2. Convert each 8-bit integer to a 32-bit integer.

3. Bit-shift each 32-bit integer s.t. the 8 read bits are at the correct position.
4. Sum all the shifted 32-bit integers.
5. Bit-cast the 32-bit integer sum to a 32-bit floating point number.

Likewise, if we were to write a 32-bit floating point number value to the memory block at offset 1, it would be done as follows:

1. Bit-cast the 32-bit floating point number to a 32-bit integer.
2. Bit-shift the 32-bit integer by 8-bits 4 times, leaving the wanted 8-bit values in the lowest order byte.
3. Convert each shifted value to an 8-bit integer. This will discard all other values than the lowest order byte.
4. Write the 8-bit integer to offsets 1, 2, 3, and 4 respectively.

Notice that boolean values in ImpCode is the size of 1 byte and since that is the smallest size allowed reading and writing a boolean value will always just be a conversion from or to an 8-bit integer. However, in SPIR-V boolean values are size-less, thus bit-casts are not allowed, so conversion is done by having 0 represent false and anything else represent true, like in C.

Most often, especially in the case of input memory, arrays are only ever accessed as an array of elements with size of its minimum access size, so a read or a write is at most a conversion from an integer to an element of the wanted type, which is generally inexpensive. When it is not the case, not only may the additional memory operations prove expensive, the access pattern will mostly be sub-optimal with respect to memory coalescing.

5 Testing

Throughout the implementation of the Futhark Vulkan backend tests were conducted using the Futhark test suite supplied with the Futhark compiler project source code. This was done using the futhark-test program by running the command

```
> futhark-test --compiler=futhark-vulkan --exclude=
    no_vulkan tests
```

from the Futhark source code directory. This will run all tests in the testing suite, reporting the number of successful and failed Futhark program files and test passes. Akin to the no_opencl tag, we have defined a no_vulkan tag to exclude test files that is not supported by the Vulkan backend. Currently, the set of test files with the no_opencl tag is equal to that with the no_vulkan tag, but if tests are added for features currently not supported by the Vulkan backend that is supported by the OpenCL backend, e.g. precision tests on 64-bit floating point number square-root, the no_vulkan tag is useful.

While testing the Futhark Vulkan backend, whenever a test would fail some tools proved crucial for debugging. One such debugging tool is the GDB debugging tool, which requires us to recompile the C file generated by the Futhark compiler using the same GCC command as the compiler, but with the -g flag, to generate debug information to be used by GDB. As an example consider a Futhark program test_a.fut on a machine running Windows 10, which would generate an executable test_a.exe and a C file test_a.c when compiled with the Futhark compiler using the Vulkan backend. To generate an executable with debug information to replace the generated executable test_a.exe, we compile with

```
> gcc -std=c99 -O3 -g -o test_a test_a.c -IC:\VulkanSDK
    \1.1.92.1\Include -LC:\VulkanSDK\1.1.92.1\Lib -lvulkan
    -1
```

assuming LunarG Vulkan SDK version 1.1.92.1 [5] located in C:\VulkanSDK. The executable test_a.exe will then contain the necessary debugging information, which can then be debugged with GDB by runnning the commands

```
> gdb test_a.exe
> run
```

which run the program and given the failing test input will hopefully help isolate the problems.

The GDB debugging tool is helpful if a problem occurs in the host code of the resulting program, but debugging becomes less straight forward when the problem is in a SPIR-V shader. To analyze the generated shaders of a program, we can first extract them using the dump-spirv flag for a Futhark program compiled with the Futhark compiler using the Vulkan backend. Again,

let us consider the test Futhark program in test_a.fut and assume that it contains two kernels when compiled with the Futhark compiler; namely kernel_a and kernel_b. Running the executable test_a.exe generated by the Futhark compiler using the Vulkan backend as

```
1 > ./test_a.exe --dump-spirv ta
```

would generate the files ta.kernel_a.spv and ta.kernel_b.spv, containing the SPIR-V shaders representing kernel_a and kernel_b respectively. Analyzing these shaders can be done using SPIR-V tools made by the Khronos group [8], which are also supplied with the LunarG Vulkan SDK. For this purpose, most noticeable of these are spirv-dis, the SPIR-V disassembler, and spirv-val, the SPIR-V validator. The disassembler converts the shaders into a more human-readable representation, making it easier to gain an overview of the generated shaders. Used in conjunction with the validator, which attempts to validate the shaders based on SPIR-V specification [7], debugging SPIR-V shaders can often be done by using the output of the validator to isolate the problem in the shader, aided by the disassembled shader. Notice however, that these tools are still under development and some potential problems, such as incorrect semantics, are obviously not found by the validator.

Problems with Futhark programs compiled with the Vulkan backend are not always located in either the host code or the device code. In some cases, the problems may occur in the Vulkan API connecting them. To save us when such a problem occurs, the LunarG Vulkan SDK supplies a VK_LAYER_LUNARG_standard_validation layer, which enables a series of other layers for debugging usage of the Vulkan API, writing all problems found to stdout in an often informative manner [12]. As to not limit general debugging of Futhark programs to machines with the LunarG Vulkan SDK installed, the debugging layer is enabled using the lunarg-debug flag when running a program. Notice that one of the layers enabled by the debug layer will output the validation results of the executed SPIR-V shaders, but this may be drowned in other errors that may be caused by the invalid shader or may simply never be reached, so the spirv-val tool is still useful on its own.

At the time of writing, the whole Futhark test suite succeeds with no errors when using the Futhark compiler with the Vulkan backend, even when passing the lunarg-debug flag to the compiler.

6 Comparison

To assess the performance of the backend, we compare both the compilation time and execution time of Futhark programs. This comparison was conducted on the DIKU APL GPU machine with a Intel Xeon E5-2650 v2 CPU clocked at 2.60 GHz and with two NVIDIA GTX 780ti GPUs. The comparison is based on the Futhark benchmarks found at [3].

Notice that, though the Futhark compiler with Vulkan backend runs the whole Futhark test suite successfully, there are still some bugs, some of which occur in the Futhark benchmarks. The programs that causes problems at the time of writing are:

- rodinia/bfs/bfs_asympt_ok_but_slow.fut
- rodinia/bfs/bfs_heuristic.fut
- rodinia/bfs/bfs_iter_work_ok.fut
- rodinia/cfd/cfd.fut
- rodinia/nw/nw.fut
- rodinia/particlefilter/particlefilter.fut
- misc/heston/heston32.fut
- misc/heston/heston64.fut
- finpar/LocVolCalib.fut
- jgf/series/series.fut
- misc/bfast/bfast-cloudy.fut

These programs fails by a variety of errors, such as wrong results, host code segmentation faults, and seemingly stalled execution. Notice that a failure in the host code may still be caused by a wrongful shader result, thus wrong final results of some program may be caused by the same bug that causes segmentation faults in another program.

The failing programs are all large, so isolating the exact issues infers more work than the simple test cases in the test suite. The programs are currently under investigation and the problems should hopefully be fixed in the near future.

6.1 Compilation

From the Futhark benchmark folder we run the commands

```
1 > time for f in $(find . -name "*.fut"); do futhark-
    vulkan $f; done
```

and

```
> time for f in $(find . -name "*.fut"); do futhark-
    opencl $f; done
```

which will run the futhark-vulkan and futhark-opencl compilers on all Futhark program files in all folders. The total compilation time recorded is around 8 minutes and 30 seconds for futhark-vulkan and around 7 minutes for futhark-opencl, which is about a 21% increase in compilation time for the compiler with the Vulkan backend, likely due some of the auxiliary passes used when generating SPIR-V shaders as well as the larger generated C programs due to the very explicit nature of the Vulkan API.

6.2 Execution

Using the futhark-bench tool we can run the Futhark benchmark programs from the folder as

```
> futhark-bench --compiler=futhark-vulkan --exclude=
    no_opencl .
```

and

```
> futhark-bench --compiler=futhark-opencl --exclude=
    no_opencl .
```

for running the benchmarks with the futhark-vulkan and futhark-opencl compilers respectively. Notice that, unlike with the Futhark testing suite, the benchmarks do not have the no_vulkan tag as it is a separate repository. The relative speedup and slowdown of the programs compiled with futhark-vulkan and futhark-opencl is illustrated in Figure 3. These relative speedups and slowdowns are based on the largest data-set generated for each file, as specified in their tests. Notice that three programs were excluded due to extreme slowdowns namely; lud-clean.fut and lud.fut from rodinia/lud, with a slowdown of about 624× and 448× respectively, and accelerate/fft/fft.fut with a slowdown of about 225×.

Investigating the culprit of this performance difference fell on short remaining time due to a focus on eliminating bugs in the compiler. However, the problem is likely to be located in some of the SPIR-V shaders due to the varying performance differences, as the launch overhead of all kernels should be roughly the same. The performance of the individual shaders can be further analyzed by running a futhark-vulkan compiled program with the debug flag set, which will output the average execution time of the individual shaders, possibly narrowing down the problem further.

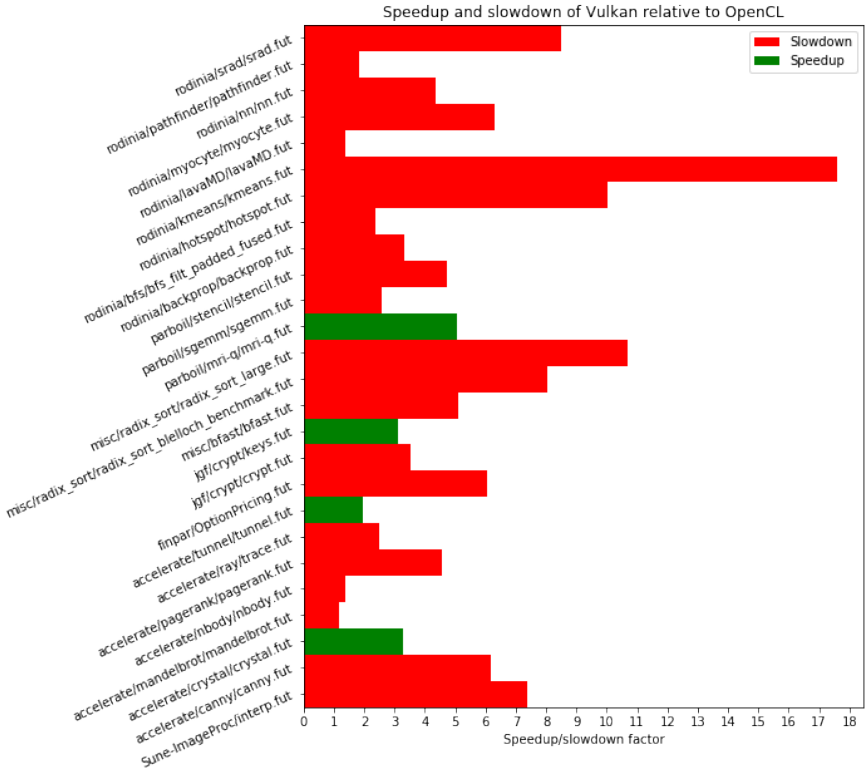

Figure 3: Speedup and slowdown of programs from the Futhark benchmark project compiled with the Futhark Vulkan compiler relative to those compiled with the Futhark OpenCL compiler.

7 Future work

Though the Futhark Vulkan backend seemingly covers most of the Futhark language with a few bugs and some lackluster performance, there are still some work missing that may prove useful or may enhance the quality of the code generated. In this section we will not exhaust the set of possible future work, but rather mention some that were considered during implementation of the current Futhark Vulkan backend, but were omitted due to time restrictions or current Vulkan limitations.

7.1 Push constant scalar-uses

In the current implementation of the Futhark Vulkan backend, any scalar-use parameter of a kernel is allocated as a memory block using the same allocation free-list that the imperative representation allocates from. This may incur more fragmentation in the free-list due to the small size of scalar-uses and may in general cause unnecessary load to the allocator as a whole.

Alternatively, given that scalar-use parameters remain constant throughout the execution of a kernel, the Futhark Vulkan backend could use push constants. These push constants are, as the name suggests, constant values that can be given when launching a compiled shader at runtime, declared accordingly in the corresponding SPIR-V shader [7, ch. 3.7]. These push constants represent a high-speed way to write constant memory [13, pg. 380], so migrating to use these may even increase performance slightly. Not only does migrating to push constants alleviate the free-list, but it would also potentially reduce the size of the descriptor pool as descriptor sets may become smaller.

Push constants are not exactly perfect however, as there is a physical limit to the number of bytes of push constants a device will allow, specifically given by the Vulkan device limit maxPushConstantsSize, which is required to be at least 128 bytes [13, ch. 30.2]. However, 128 bytes should normally be enough for scalar-uses, as they are non-composite and often few. Notice however, that push constant values must be byte-aligned equal to the number of bytes in their type, e.g. a 32-bit integer must be at a multiple of 4 byte-offset in the push constant structure given when push constants are submitted [13, ch. 14.5.4]. This will often force gaps in the structure since scalar-uses may be of different byte-sizes. To reduce this gap we can sort the scalar-uses by their size and pack them as tightly as possible in a structure, decreasing the wasted bytes in the limited push constant memory. This can be implemented in C by either allocating a block and distributing the data within it or by statically defining a structure and placing the elements in order, placing padding to ensure correct offsets. Notice that the latter will for some C compilers require the enabling of explicit byte packing as to prevent the compiler from adding its own padding. In GCC this can be done using the pack pragma [9].

The push constant scalar-use migration has almost been implemented, but due to a peculiar bug caused by the change it has been kept local and will likely appear in the Futhark Vulkan backend in the near future.

7.2 Variable memory references

The Futhark Vulkan backend does not exactly implement a complete mapping between Futhark's imperative kernel representation and SPIR-V instructions, as Vulkan does not currently allow for inline declaration of variable memory references and their assignments. When a variable memory reference declaration occurs it will, like with other memory blocks in the imperative representation, not carry any information about the size at which the elements may be accessed. Generally this would be discerned by the minimum access size analysis pass, but given that a variable memory reference may change at runtime, we would need the same minimum access size on all memory blocks that may be accessed. This could potentially be implemented by keeping track of all memory blocks that may be referenced by the same variable memory reference, letting their minimum access size be the minimum of the set after the analysis pass. Notice that sets of memory blocks referenced by variable memory references may intersect, so the minimum access size of one set may depend on another, therefore an evaluation order must be inferred.

Luckily, these variable memory references seems to currently only be in use if the experimental flattening environment variable is set on the machine compiling a Futhark program, so in the general case a Futhark programmer will not encounter this missing implementation. If somehow it is encountered, the compiler will inform the programmer at compile-time rather than fail at runtime. Notice that the memory copy operation has likewise not been implemented, as it seems to primarily appear in tandem with the use variable memory references, so it was omitted for now. If the memory copy operation is revealed to appear outside the use of variable memory references, it should be somewhat straight forward to implement using a loop and the SPIR-V memory copy instruction [7, ch. 3.32.8]. However, if the memory blocks being copied between have different minimum access sizes, it cannot use the SPIR-V memory copy instruction, but will instead have to read a suitable number of elements from memory before either splitting or combining them and writing the result to the target memory, akin to the current array write and read strategy discussed in Section 4.3.

Notice lastly that the additional restrictions on the minimum access sizes of memory blocks are more likely to cause the bad memory access patterns as described in Section 4.3, as they may change the minimum access sizes. If in the future we are to see the SPIR-V Addresses capability or other support for pointer conversion, it should eliminate the need for the minimum access size pass and make the implementation of variable memory references simpler.

8 Conclusion

With some challenges and extensive effort, the Futhark compiler project now includes a Futhark compiler variant utilizing the Vulkan API with locally generated SPIR-V shaders, expanding the choices available to the Futhark programmer as well as extending the set of hardware that the Futhark compiler project supports. Due to the modularity of the generated imperative representation and the corresponding kernels, the implementation of the Futhark Vulkan backend required no change to the structure of the existing compilers and was even able to reuse the existing C backend as the base for compiling the host code of Futhark programs. Therefore, the Futhark Vulkan backend came as a natural extension to the Futhark compiler project.

Though most programs seemingly compile and run as expected using the Futhark Vulkan backend, there are still some bugs and lackluster performance that is currently being investigated and is hopefully resolved as soon as possible. Additionally, there are also currently limitations to the implementation, such as limited support for 8-bit integers and missing support for pointer conversion. These limitations will however not generally influence the compilation of a program and will mostly be reported to the programmer if encountered. Due to these however, the Futhark Vulkan backend is likely to stay experimental in the foreseeable future. Hopefully we will see these limitations mitigated in a future version of Vulkan, potentially making the Futhark Vulkan backend a valid alternative to the Futhark OpenCL backend.

References

[1] Android: Vulkan support. https://source.android.com/devices/graphics/arch-vulkan.

[2] clspv. https://github.com/google/clspv.

[3] futhark-benchmarks. https://github.com/diku-dk/futhark-benchmarks.

[4] Khronos registry. https://www.khronos.org/registry/spir-v/api/spir-v.xml.

[5] LunarG Vulkan SDK. https://vulkan.lunarg.com/.

[6] MoltenVK. https://github.com/KhronosGroup/MoltenVK.

[7] SPIR-V Specification. https://www.khronos.org/registry/spir-v/specs/1.0/SPIRV.pdf. version 1.00, Revision 12.

[8] SPIR-V Tools. https://github.com/KhronosGroup/SPIRV-Tools.

[9] Structure-packing pragmas. https://gcc.gnu.org/onlinedocs/gcc-4.4.4/gcc/Structure_002dPacking-Pragmas.html.

[10] The Futhark Programming Language - vulkan-backend branch. https://github.com/diku-dk/futhark/tree/vulkan-backend.

[11] Vulkan Extensions and Devices Database. https://vulkan.gpuinfo.org/listextensions.php.

[12] Vulkan Validation and Debugging Layers. https://vulkan.lunarg.com/doc/view/1.0.13.0/windows/layers.html.

[13] Vulkan 1.1.96 - A Specification (with all registered Vulkan extensions). https://www.khronos.org/registry/vulkan/specs/1.1-extensions/pdf/vkspec.pdf, December 2018.

[14] James Batchelor. glNext revealed as Vulkan graphics API. *https://www.mcvuk.com/development/glnext-revealed-as-vulkan-graphics-api*, 2015.

[15] Neil Henning. A simple Vulkan Compute example. *http://www.duskborn.com/a-simple-vulkan-compute-example/*, 2016.

[16] John Kessenich. SPIR-V: A Khronos-Defined Intermediate Language for Native Representation of Graphical Shaders and Compute Kernels. https://www.khronos.org/registry/spir-v/papers/WhitePaper.html, 2015.

[17] John Kessenich. SPIR-V Extended Instructions for GLSL. https://www.khronos.org/registry/spir-v/specs/unified1/GLSL.std.450.pdf, August 2018.