A Vulkan Backend for Accelerate

Master Thesis

Xinliang Lu
xinliang.lu@outlook.com

12 August 2024

Supervisors: Prof. dr. G.K. (Gabriele) Keller
I.G. (Ivo Gabe) de Wolff MSc
Dr. M.I.L. (Matthijs) Vákár

Computing Science
Department of Information and Computing Sciences

Universiteit Utrecht

My light boat has already passed ranges
and ranges of mountains.

Li Bai

To my family, friends,
and my beloved important one.

Contents

Abstract

With the continuous advancement of parallel hardware, and the growing demand for cross-platform high-performance computing, Vulkan, as an open standard and cross-platform, low-overhead, low-level library for 3D graphics and computing, has emerged noticeably. The vast majority of scientific software, however, only supports executing programs on NVIDIA graphics processing units (GPUs) due to various reasons. Accelerate, a domain-specific language (DSL) for high-performance array computation in the functional programming language – Haskell, is one example that traditionally relies on CUDA for GPU acceleration.

This thesis presents the development of a Vulkan backend for Accelerate, enabling general-purpose GPU computing on a broader range of hardware. The main contribution includes the design and implementation of a code generator for the OpenGL shading language (GLSL), which is a language for building the Vulkan computing kernel. The code generator supports most primitive scalar operations in Accelerate. Additionally, a runtime for the Vulkan backend was developed.

Experimental results demonstrate that the Vulkan backend offers competitive performance compared to the CUDA backend, proving its viability as an alternative. However, Vulkan extension support across various platforms poses the biggest limitation of our work, restricting the ability to run programs on certain devices.

Beyond technical contributions, this thesis also provides comprehensive introductions to the components of Accelerate and the features of Vulkan. We hope to provide extensive learning material for those interested in the Vulkan compute pipeline and possibly developing a Vulkan backend for similar DSLs.

1 Introduction

In recent decades, computing technology has evolved quickly, especially in massive physical simulations, large-scale neural networks, and big data mining. They all have a similar need for efficient parallel computing, which is the key to boosting performance. For instance, the most effective approach to rapidly compute a fractal image involves avoiding single-thread algorithms and utilizing multiple cores on central processing units (CPUs) or other parallel hardware.

1.1 Array programming

Array programming is a way to access, manipulate, and operate on arrays [29]. Each array has a few important pieces of information: the shape and its element's type. Shapes contain both the dimension structure and the length for each dimension, i.e. the dimensional number. For example, a shape $(10, 5, 2)$ means a three-dimensional structure where the lowest dimensional number is 10, and the highest is 2. To be specific, a vector has one dimension, and a matrix has two dimensions. A matrix consists of rows and columns, which can be seen as the extension of a vector. So, if we index the matrix's rows first and then columns, the latter can be considered extended from the former. In this case, we called the dimension reflecting columns the higher dimension.

As for the array length, it is the linearization of shape, which can be computed by multiplying all dimensional numbers together. It is often used in storing arrays in the flat memory. At that point, the multi-dimensional array will be linearized into a vector according to the row-major or column-major orders [56], i.e. concatenating rows (fixing lower dimensional indices) or columns (fixing higher dimensional indices) into a long list of values. In this thesis, we use the row-major order to store arrays.

In terms of array manipulations, they can be classified into two types: array-level ones and element-level ones. The former focuses on transforming the array as a whole, e.g. array production and array slicing. However, inside these operations, more thorough and operational element-wise functions guide the transformation, e.g. updating elements one by one. In general, the former takes the latter as its argument or its subroutine, so it operates at a higher level (more abstract).

Apart from ordinary arrays, i.e. each position of the array only contains one value, array of structures (AoS) and structure of arrays (SoA) [52] are also common to represent high-dimensional data. The former is an array, but each position accommodates a record of data. For example, in C++, one can define an array of structures or an array of tuples in Haskell. SoA is the opposite of AoS, where a record contains multiple ordinary arrays. In most scientific libraries, including Accelerate, though users can interact with AoS data, inside the pipeline, AoS data is usually converted to SoA, which is easier to process.

1.2 Data parallelism

Data parallelism is one of the programming models that maximizes the potential of parallel hardware. It focuses on processing each data item of a data collection in independent task instances [30]. In other words, it maps a function to each value over sets of data.

Flat and nested data parallelism have been introduced to exploit parallelism for different data structures better. The major difference between these two models is the type of functions mapped to the data, where the former requires a sequential one while the latter takes any function, including parallel ones [11]. Nested data parallelism can thus be seen as an extension of flat data parallelism, and has been recognized as necessary for writing high-level parallel programs [12]. Compared to flat data parallelism, nested data parallelism can better utilize the power of parallel hardware. This is because the executed function can also be parallelized and run on multiple threads.

Nested data parallelism has many advantages. However, it comes with costs. The most obvious one is the overhead associated with the spawning, scheduling, and synchronizing of threads. To fully exploit the parallelism, nested data algorithms may decompose the mapped function into individual instances, and execute each part in parallel. In this process, the algorithm needs to balance the benefits of increased parallelism and the overhead incurred. The second cost comes from the execution efficiency of irregular parallelism on certain parallel hardware. Graphics processing units (GPUs) are a good example of such a problem. They provide good performance for integer and floating-point calculation, because of the massive amount of functional units (execution units for integer/floating-point data [43]) integrated in a single GPU. However, they can only peak the performance when all the units execute the same instruction within the minimum schedulable group, since these units share the same instruction unit (including instruction scheduler and some other resources) [43]. Consequently, when executing non-uniform (or irregular) parallelism, the parallel program becomes sequential, and only one unit is activated. As a result, nested data parallelism is not well-suited for execution on this type (known as Single-Instruction-Multiple-Data or SIMD, introduced in §2.1) of devices (such as GPUs) [9].

One solution is to vectorize the nested data parallelism program and flatten the nested array structures to execute operations more efficiently [9, 35]. This is also known as flattening transformation (or vectorization), introduced by Blelloch [10]. The approach includes two steps: lifting functions operating on a single value into array operations, and creating a flag vector to identify whether a value from the flattened vector corresponds with the beginning of the original array.

1.3 GPU computing and its eco-systems

Graphics processing units (GPUs) are among the most common and affordable pieces of parallel hardware. They have a long history starting from the 1980s or even earlier [21]. They were first used as devices to draw wire-frame shapes on the screen serving a specific purpose. Furthermore, they were built to process a large amount of data in parallel. Traditional GPUs, however, only serve to calculate graphical textures. People then proposed a special concept of using GPUs to enable general-purpose computing, and GPUs that fulfil this concept can be called general-purpose graphics processing units (GPGPUs) [57]. In other words, GPGPUs are a sub-class of GPUs that can be programmed to execute general-purpose code and are not limited to graphics commands. Compared to CPUs, GPUs or GPGPUs are designed to share each instruction unit (scheduler, dispatch unit, etc.) across multiple functional units (or cores). Hence, they have more space to accommodate more cores, which results in much better performance in processing integer and floating point data.

With the introduction of GPGPU hardware and developing kits by manufacturers, a considerable amount of third-party libraries have emerged, leveraging such computing power while offering user-friendly programming models and other features for users [14]. However, most of the scientific software is developed using the CUDA [23] library, which is NVIDIA's toolkit and a programming language for developing NVIDIA GPGPU programs. This is due to two main reasons: significant market share and the CUDA ecosystem.

As an important player in the GPU market, NVIDIA provided devices with innovative features and reasonable prices in the early years [48], and they were equipped with mature drivers. Thus, NVIDIA historically took a large share of the GPU market [48]. After that, NVIDIA first equipped even their gaming GPUs with the support of general-purpose parallel computing in 2006, namely the GeForce 8800 [25], and simultaneously implemented marketing campaigns of their CUDA library. For programmers and scientists, this met the need for cheap parallel hardware with mature development kits. Together, NVIDIA and their GPGPUs gradually became widespread in the programming community [51].

NVIDIA's CUDA ecosystem is mature and well-supported by most NVIDIA GPUs for general-purpose GPU programming and computing, which includes vast libraries, tools, documents, and large community supports. Because of its ease of use and relatively smooth learning curve, backends based on CUDA were developed and integrated into many scientific libraries and packages for utilizing the great power of GPGPU, including the PTX backend for Accelerate [13] (introduced in §1.4). Thus, many developers stuck with CUDA rather than investing resources to support alternative platforms.

In 2016, Vulkan [5] was initially released and many vendors support it nowadays. Vulkan is not only a low-level, low-overhead, cross-platform library for 3D graphics and GPU computing, but also an open standard in these areas [5]. Over time, Vulkan won significant support from both hardware and software industries. The main reason for this is that it was developed by the Khronos Group – an industry consortium that includes major hardware and software companies. Apart from that, its performance, scalability, cross-platform features, and advanced structure design, make it popular. It is gaining more and more support from consumer-grade device manufacturers, including phone makers.

1.4 Accelerate

To utilize the power of GPGPUs while maintaining a user-friendly programming model, various domain-specific languages (DSLs) were created to enable offloading computation to GPGPUs. Accelerate [13] is such a DSL, used for high-performance parallel array computations in Haskell with type safety as its biggest selling point. It currently has three backends:

1. CPU (Native) Backend: This backend employs multiple CPU cores for parallel computing by generating LLVM IR [36] first and then compiling to machine code.

2. NVIDIA GPU (PTX) Backend: Using the LLVM NVPTX backend [37] to generate NVIDIA GPUs executable machine code and enabling computations to be offloaded to such GPUs. PTX [44] is a low-level assembly-like language used as a target for CUDA code compilation. It represents NVIDIA GPU parallelism and memory operations in a low-level form, including comprehensive instructions for thread management, synchronization, and memory operations. In short, the relationship between PTX and CUDA is like that between assembly and C.

3. Sequential Interpreter: Unlike the other backends, this backend evaluates Accelerate's program directly without transforming it to a third-party intermediate representation (IR).

1.5 Dilemma and a way out

Currently, a large number of libraries selling a convenient way to access the power of GPGPU are emerging. However, many such parallel computing libraries or DSLs only have a CUDA backend for the GPGPU-offloading computing pipeline, not a cross-platform/cross-vendor backend, including Accelerate. This is because CUDA is mature and well-supported by both NVIDIA and the community. Additionally, Vulkan is not widely used for two reasons. First, the material about using Vulkan to perform general-purpose computations is not as extensive as that of CUDA. Vulkan is still new and needs time to mature and gain widespread adoption. Moreover, while it is supported by multiple GPU vendors, it may not receive the same level of focus and investment from individual vendors as CUDA does from NVIDIA. Second, using Vulkan is not as handy as using CUDA in general. Moreover, it is extremely verbose to use.

Although using Vulkan is not as easy as programming in CUDA, the cross-platform concept is tempting. The vision of running Accelerate applications on consumer devices, like smartphones, mobile VR/AR headsets, and wearables, is right in the corner powered by Vulkan. Therefore, this thesis studies the possibility of using the Vulkan compute pipeline to implement a backend of Accelerate. We evaluate such an implementation with a series of benchmarks, and compare it with the old CUDA backend and the CPU backend of the new pipeline. To be more specific, we create a pipeline to convert Accelerate IRs into Vulkan-compatible code and benchmark the performance of Accelerate programs.

1.6 Research questions

The main goal of the thesis is to examine whether it is possible or not to implement a Vulkan backend for Accelerate. Therefore, we investigate the following research questions:

RQ 1. Is it possible to generate GLSL kernel functions from Accelerate's intermediate representation (IR)?
The compute kernel (known as the compute shader) of Vulkan is coded using OpenGL shading language (GLSL) [40]. Thus, to perform computations, the intermediate representation of Accelerate taken from the surface language is necessary to convert into GLSL kernel code. This means that except for memory management (or array allocations) and data transfer, all operations, including mapping, generation, and permutations, need to be transformed into one kernel function in a way. Because these are represented as higher-

order array operations, while GLSL is a first-order programming language, a certain amount of work would be required to implement such a conversion layer.

RQ 2. Does the Vulkan provide suitable APIs to implement a backend for Accelerate?

After generating the GLSL code, a GLSL compiler compiles it into machine code, which the Vulkan framework uses to run the program on heterogeneous systems. Vulkan, however, is still new, and its community support needs time to mature, which means it may not include all the suitable APIs to implement the Acclerate backend.

Thus, for Vulkan, a certain list of functions supported on the host and the target (Vulkan-supported device) side must be checked. On the host side, APIs for data transfer from and to the target device and memory management are essential to inspect since they function as the cornerstone for all data manipulations.

On the target side, Accelerate currently supports a large range of mathematical operations and various data types, which should also be supported in Vulkan kernels. Although GLSL has rich support for such operations and data types, Vulkan only accepts a small subset of GLSL's features. Examining the range of supported features in GLSL of Vulkan version, checking whether they are sufficient and suitable or not to build the computation, are important for implementing such a backend.

This could be challenged if not all the functions, data types, and operations are supported by Vulkan or GLSL itself. On the Vulkan framework side, a certain amount of workarounds are needed to emulate missing functionalities. This requires me to be familiar with Vulkan's APIs. On the Vulkan-specific GLSL side, a decent amount of work will be allocated to simulate and bypass such invalid parts in GLSL to compensate for the missing data types and operations.

RQ 3. Compared to the old CUDA backend, how is the performance of the new backend?

If the Vulkan backend for Accelerate is implemented, a benchmark and comparisons between the new and old backend help spot out limitations of such a backend and Vulkan itself. Also, this gives insights for future work.

2 Preliminaries

2.1 Data parallelism models

Most GPGPUs have thousands of functional units (or cores), which make them suitable for executing massive parallel programs. To unleash and better utilize such computing power, two data-level parallelism programming models are commonly used: single instruction multiple data (SIMD) and single instruction multiple threads (SIMT).

2.1.1 SIMD

Just as its name describes, Single instruction multiple data (SIMD) enables hardware to perform the same operation on multiple data in parallel by a single instruction. This not only improves the overall throughput for tasks involving large datasets and repetitive calculations, but also increases performance for vectorizable computations by exposing native SIMD interfaces. In addition, the support for SIMD on the hardware depends on both architecture design and instruction set. For example, in the x86 architecture (a common CPU instruction set), `_mm256_add_pd` [32] is used to add two packed 64-bit floating-point numbers together (or add two vectors together) for a single instruction.

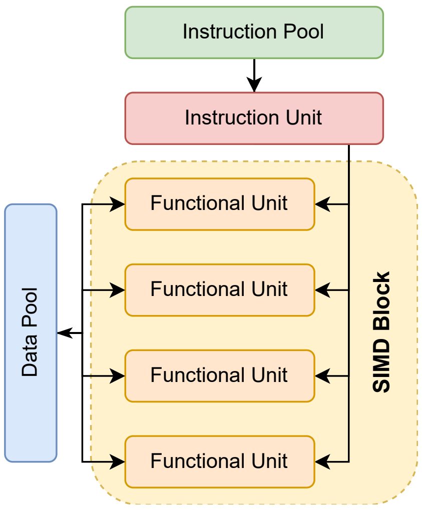

Figure 1. *4-lane SIMD block (inspired by Curtis et al. [19, Figure 1a]).*

In detail, Figure 1 depicts a simple yet common SIMD block featuring four execution units that share a scheduler. This block allows for applying a single instruction to four pieces of data. Since these units share the same instruction unit, when encountering any control-flow instruction, they should all enter the same branch to maximize performance. Otherwise, a penalty may happen if some units enter different branches [46] which is also known as non-uniform (or irregular) parallelism, since one instruction unit can only handle one situation at a time. However, thanks to the absence of a dedicated scheduler and other instruction units for each functional unit, manufacturers can have more area to place more execution cores on the chip, which is typically how mainstream GPUs are designed and what most GPU vendors are doing.

2.1.2 SIMT

Single instruction multiple threads (SIMT) is very similar to SIMD, but instead of executing one instruction with multiple data in one thread, SIMT enables the hardware to execute the same instruction with different data in different threads, which allows non-uniform parallelism. Recalling the vector-addition example given in the last paragraph, in SIMT, the way to achieve parallel addition for each element in two vectors is to create a bunch of threads and do single-element addition in each thread.

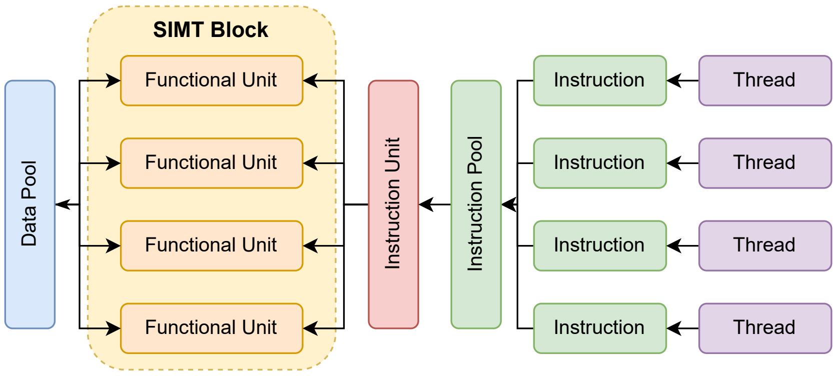

Figure 2. *4-cores SIMT block (inspired by Curtis et al. [19, Figure 1b]).*

Although SIMT treats data parallelism differently from SIMD, at the hardware level, not much modification is required. As shown in Figure 2, multiple instructions called by different threads enter the instruction pool first and then are scheduled and dispatched by the instruction unit. Take CUDA as an example, SIMT model is mostly a different software approach on the same hardware, because it can automatically map a thread to a functional unit in the SIMD block.

2.2 Accelerate

In Accelerate [13], CPU and NVIDIA GPU backends use SIMD and SIMT models to achieve data parallelism, but in the surface language, users program the computation using high-level interfaces without touching the parallelism. Furthermore, we call the stage, which transforms the surface language into low-level representations and executes them, as the pipeline.

2.2.1 Compiler pipelines

Accelerate has two pipelines to compile and execute programs: the old and new ones. Although in the thesis, the old pipeline is not relevant to our work on the Vulkan backend, we introduce it to readers who might be interested in learning about the involvement of Accelerate.

The new pipeline is currently in development and will eventually replace the old one. Each describes the construction of the computation flow, low-level code generation, and computation scheduling. For now, only the CPU and the Vulkan backend proposed in this thesis support the new pipeline. In both pipelines, to conduct the array computation, the first step is to build the compute steps according to the user-defined array computation using the surface language. No actual array or value is involved except their structure and type at this stage. Then, concrete values are passed into compute steps to produce the result.

Figure 3 demonstrates the old and new pipeline stages in Accelerate, with key components and essential transformations. SmartAcc is an intermediate representation (IR) for array-level computations, which is the input of all backend-specific compilers in Accelerate. Sharing Recovery and Fusion are optimization techniques to reduce redundant computation and overhead. The former recovers the sharing let-binding expressions to prevent recomputing them, and the latter eliminates intermediate values to avoid the overhead of dispatching additional compute kernels and data transfer [42]. "Named" IR is an abstract syntax tree (AST) representing collective array computations on program level parameterized over array variables. It also contains information to represent sharing and control evaluation orders.

In the old pipeline, there are two IRs: Delayed IR and Skeleton. The Delayed IR serves as a container capturing various array operations and simply composites them as a single kernel as long as they can be chained together without launching a new kernel. The Skeleton is a backend-specific (or hardware-specific) IR that encloses the actual kernel awaiting execution in devices. Due to the diverse implementations of backends across different devices, each supporting various kernel codes, the skeleton is tailored for both

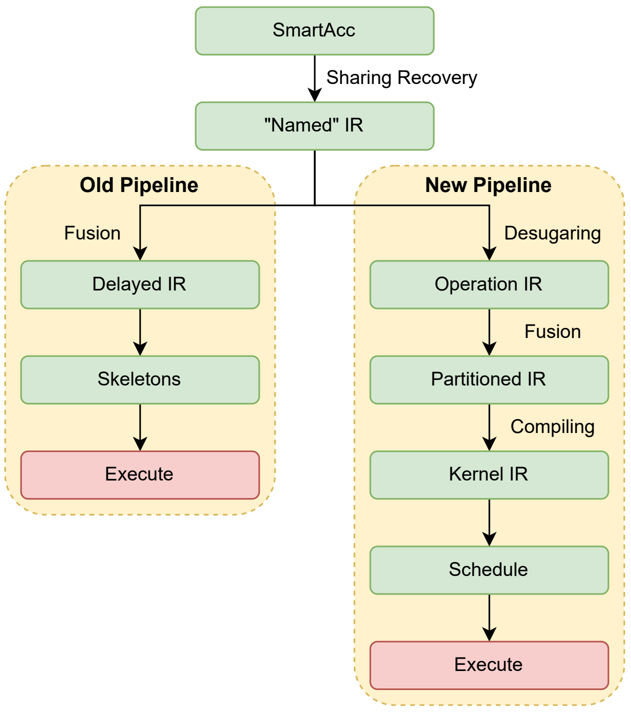

Figure 3. *Accelerate pipelines.*

hardware and the backend.

In the new pipeline, four IRs play key roles: Operation IR, Partitioned IR, Kernel IR, and Schedule. Beginning with the “Named” IR, desugaring takes place in the very first step. As its name describes, this transformation converts program-level array computations into more extensive computing operations (containing backend-specific operations), namely Operation IR. As such, Accelerate allows each backend more flexibility in deciding how to represent the array computation using dedicated hardware operations. Since different backends target different executing devices, each potentially providing unique features to accelerate data operations, Operation IR can improve computing efficiency. Then fusion shows up to convert operations into Partitioned IR, which is a collection of Operation IRs, as some of them can be chained together to reduce the overhead of launching additional kernels. In short, the purpose of Partitioned IR is the same as Delayed IR. After that, backend-specific kernel codes are generated and compiled according to the backend-specific operations within Partitioned IRs. Then, we can have Kernel IRs, which are not of the actual type in Accelerate. Instead, the type is the same as the Partitioned IR but parameterized over the Vulkan kernel type and contains device-specific kernel codes. Finally, these IRs would be scheduled and executed with the help of backend-specific runtimes. This step controls the order and place (either on devices or the host) of executing each Kernel IR.

The old pipeline is the very initial one for Accelerate, and has been used for a long time. It accommodates many transformations and optimizations in one module, plus some of them can be have made universal across different backends. For these reasons, the new pipeline has come out. Though it is still experimental, the concept of reusing common components and separating nested IR layers relieves the work of creating new backends for Accelerate. Now, not only is fusion more flexible, but also implementing a new backend is easier, and each can have its own optimization stages.

In general, all the internal stages and IRs can be classified into Accelerate's compile-time and runtime. Only the execution belongs to the runtime, while others take place in the compile-time. Sometimes, the" runtime" also refers to libraries or functions that are invoked at runtime.

2.2.2 Programming model

Accelerate is a higher-order DSL in Haskell. As such, array manipulations are done through high-level array operations instead of element-wise computations. To be more precise, unlike low-level array operations, such as using loops to compute or update each element in the array, Accelerate takes lambda calculus or functions and then maps them to the individual element.

Consider a dot product example in Accelerate:

```
 1 import Data.Array.Accelerate
 2 import Data.Array.Accelerate.LLVM.Native as CPU
 3 
 4 dotp :: Acc (Vector Float) -> Acc (Vector Float) -> Acc (Scalar Float)
 5 dotp xs ys = fold (+) 0 (zipWith (*) xs ys)
 6 
 7 main :: IO ()
 8 main = do
 9     let vec1 = fromList (Z:.10) (take 10 (repeat 1)) :: Vector Float
10     let vec2 = fromList (Z:.10) (take 10 (repeat 2)) :: Vector Float
11     let result = CPU.run $ dotp (use vec1) (use vec2)
12     putStrLn $ show result
```

Listing 1. *Accelerate code snippet for dot product which runs on CPU.*

In the above code snippet, the first and second lines import basic building blocks from Accelerate. The fifth line defines a function `dotp`, given two arrays as input. It performs as an element-wise product first, then folds all elements together by accumulating each one. The ninth and tenth lines create two floating-point vectors containing ten ones and twos, respectively. The eleventh line first calls function `use` to embed the input, making it available for the computation, then runs function (or kernel) `dotp` using the CPU backend and finally binds the result.

In Listing 1, only array-wise operations – `zipWith` [53, §Zipping] and `fold` [53, §Folding] are involved. Though deep in the backends of Accelerate, these high-level operations are desugared into low-level operations that hardware can recognize, the surface language provides concise higher-order interfaces to express array operations.

2.3 Vulkan

As described before, Vulkan [5] is a library and an open standard for 3D graphics and GPU computing with advanced designs and benefits. However, these advantages come with costs. Take cross-platform and hardware control as an example. To provide consistent APIs and configurations across various platforms while allowing developers to optimize their programs, Vulkan exposes many details of the underlying hardware architecture and chooses not to handle these. This means programmers have no choice but to manage all the resources manually. Though Vulkan has phenomenal features, such as cross-platform, good performance and efficiency, the design philosophy and goals of Vulkan make it extremely verbose to use, requiring developers to specify every detail explicitly. Moreover, programming models for the host side and device side are different. On the device side, developers need to use GLSL to code kernel functions that are awaiting execution on the Vulkan device, which further makes the overall learning curve steeper for beginners.

2.3.1 Programming model on the device side

The code on the host side depicts common steps for initialization, memory management, data transfer, command execution, etc. It builds a suitable Vulkan environment in which to execute shaders, where shaders are Vulkan kernels that are written in special shader languages (such as GLSL [40] or HLSL [45]) and compiled into SPIR-V (Standard Portable Intermediate Representation) [54] bytecode. SPIR-V is no more than an intermediate binary representation of defining shaders [54] and will be compiled by the GPU driver to the GPU native code when loading it into the shader [33]. Apart from the binary format, it has a human-readable text format called SPIR-V Assembly. In other words, the relationship between SPIR-V and GLSL is like that between PTX and CUDA languages.

There are various types of shaders (e.g. vertex shader, geometry shader, compute shader) in Vulkan, and each serves a specific purpose in either rendering or computing. Compute shader is one of those shaders and is used for general-purpose computing on GPUs. Here is an example of the compute shader that calculates the dot product coded in GLSL:

```
1  #version 450
2  #extension GL_EXT_shader_atomic_float : enable
3  #extension GL_ARB_shader_storage_buffer_object : enable
4
5  layout(local_size_x = 32) in;
6  layout(set = 0, binding = 0) buffer InBufferXs {
7      float xs[];
8  };
9  layout(set = 0, binding = 1) buffer InBufferYs {
10     float ys[];
11 };
12 layout(set = 0, binding = 2, std430) buffer OutBuffer {
13     float zs;
14 };
15
16 void main() {
17     uint globalIndex = gl_GlobalInvocationID.x;
18     if (globalIndex >= xs.length()) return;
19     float elementProduct = xs[globalIndex] * ys[globalIndex];
20     atomicAdd(zs, elementProduct);
21 }
```

Listing 2. *GLSL code snippet for dot product.*

In the above code snippet, from line 1 to line 3, the GLSL version and enabled extension for it are identified. Line 5 expresses the local thread number when executing the shader. From line 6 to line 14, input and output buffers are declared with variable names and binding numbers. Line 17 queries the global thread (called the invocation in Vulkan) index for the current thread, and the following line checks if it is indexing out of the buffer. Lines 19 and 20 compute the element-wise product and use atomic addition to accumulate it.

Although in the example Listing 2, we set the local thread number to 32 (on the fifth line) as a working group, during execution, we launch multiple working groups so that each thread only needs to handle one element from the buffer.

2.3.2 Programming model on the host side

To better introduce Vulkan, we show its programming model on the host side (i.e., the common steps to build a Vulkan application), where coding for the host side is more demanding than that of the device side. This also shows how verbose Vulkan is.

The starting point of making a Vulkan program is creating an instance, which is a fundamental object. It serves as the gateway for the application to interact with the Vulkan library. To create such an object, one must specify the creation information [55, §4.2. Instances], such as enabled extensions and layers. Before enabling them, users should call the Vulkan runtime to determine which device to use and to query what extensions and layers it supports.

Then, with the instance, we can create a Vulkan device object, which is a logical device that serves the purpose of bridging the Vulkan application with the physical device. It is used to perform operations like memory management, command management, and task submission. When creating a logical device [55, §5.2. Devices], we should specify the queue family [55, §5.3.1. Queue Family Properties] to identify the type of commands that will be executed on it. This tells the device whether to perform graphics, computing, or other operations. Each of them has different hardware behaviors and performance.

With the instance, logical device, and physical device, we can now create a Vulkan memory allocator [4] to simplify memory management. Specifically, it is a utility that exposes convenient interfaces to allocate and manage memory resources in Vulkan applications, including memory allocation, alignment, deallocation, and memory pooling. With the help of the allocator, developers can write efficient and robust Vulkan applications.

After creating an allocator, we can use it to allocate memory space to accommodate Vulkan buffers. Note that buffers are required to be declared before then. The byte size and usage flags are necessary to create a buffer, where the usage flag [55, §12.1. Buffers] points out whether the buffer is uniform or treated as a pipeline to transfer commands, etc. Then, we call the allocator to assign a memory space to the buffer, given the memory usage flag that tells the allocator which parties can visit the space and how often they visit it. For example, we can allocate a visible space for both the Vulkan device and the host, but the device performs read and write operations more frequently.

Then it comes to the step for creating the descriptor pool, the descriptor-set layout, and the descriptor set. We can consider the descriptor pool as a container to hold descriptor sets that collect descriptors [55, §14. Resource Descriptors]. In short, a descriptor functions as a pointer to resources stored in the memory. It directs the shader (or kernel) to the location of an object, either a buffer, an image, or even a function. Moving on to the descriptor-set layout, it defines the types and positions for descriptors in the descriptor set, with the position denoted by a binding number. When creating such a layout, users must set the type and binding for each descriptor. Once the descriptor pool and descriptor-set layout are established, a descriptor set can be allocated. However, buffers and other resources remain unbound to the set at this stage. To continue, we have to manually update it with all the buffers that will be accessed by the shader and corresponding binding numbers.

Now, it is time to create the key module, namely the shader module [55, §9. Shaders]. In Vulkan, shaders are kernels written in special shader languages (such as GLSL [40] or HLSL [45]) and compiled into SPIR-V (Standard Portable Intermediate Representation) bytecode. In this thesis, we choose GLSL as the shading language for making shaders. There are various types of shaders (vertex shader, geometry shader, compute shader, etc.) in Vulkan, and each serves a specific purpose in rendering or computing. Given the binary shader and the logical device, these shaders will be loaded into Vulkan as shader modules.

Then, we create a pipeline [55, §10. Pipelines] to wrap the shader module. Before building such a pipeline, supplementary information must be included. This includes the shader module and the entry point for the main function. We should also create a pipeline layout that contains a list of descriptor-set layouts. With all this information, we can now create a valid pipeline.

Similar to the descriptor pool and descriptor set, we need to define a pair of command pools and buffers [55, §6. Command Buffers] to record the commands that will be pushed to the device. Note that different commands should be classified to fit with the queue family to improve performance.

At this point, the command buffer is empty and in the pending state. For the next step, we need to wake the command buffer and make it in the recording state [55, §6.4. Command Buffer Recording]. Then we bind the pipeline and descriptor set to it and dispatch given a thread number for the kernel. Then, we end the command buffer to make it return to the pending mode. Before submitting the mission, we should map the Vulkan buffer to get a pointer for access and transfer the data from the ordinary memory space to the buffer with the pointer.

As mentioned, different pipelines should be submitted to different queues according to the queue family. We query the correct queue and submit the command buffer to it. After the submission, commands are executed instantly. Now, we wait for the fences that either mark the execution done or time out [55, §7.3. Fences]. In addition, we should also wait for the data transfer to be done and visible to the host.

For the very last step, data is waiting to be copied back from the Vulkan buffer. Apart from that, the essential action is to free all the resources that were created or allocated before. They cannot be deallocated too soon or too late.

In general, the actions described above are the common and required steps when creating a Vulkan application. Resource management and explicit control are the major difficulties in this process.

2.3.3 Vulkan-Haskell bindings

Because Accelerate is coded using Haskell while Vulkan only provides interfaces in C++, to make a Vulkan backend for it, we have to find a way to bridge native Vulkan APIs with Haskell, which is to find a Vulkan-Haskell binding. In other words, Vulkan-Haskell bindings are supposed to allow developers to utilize Vulkan APIs within Haskell programs. These bindings empower Haskell developers to create high-performance graphics and GPU computing applications with Vulkan's flexibility and low-level control. Currently, there are two such bindings available in Hackage, which are `vulkan` [22] and `vulkan-api` [3]. The former provides more Vulkan APIs than the latter and keeps updating. Thus, in this thesis, we use it to call Vulkan interfaces and implement a Vulkan backend for Accelerate.

2.4 Haskell

In Haskell, both algebraic data types (ADTs) and generalized algebraic data types (GADTs) are data constructors. They work as containers to group multiple data together and expose specific types for such data groups to guarantee more type safety. Though ADTs and GADTs are similar, the major difference is that GADTs provide a finer and more flexible way to define data types with varied and explicit type constraints. Because of these features, ADTs and GADTs are commonly used in Accelerate to construct complex data with metadata, such as storage locations, data representations, and lifetimes. In other words, Accelerate is built based on ADTs and GADTs, so we introduce them in the thesis.

Apart from them, record notation is a useful Haskell syntax, which is also widely used in Accelerate.

2.4.1 ADTs

Though the name algebraic data types (ADTs) sound too mathematical, the concept is quite straightforward. In ADTs, several common types or usage patterns are used to form the constructor: sum types, product types, single constructors, recursive types, and enumerated types. To clarify, type or type-level information means the part on the left-hand side of the equality, and data or data-level information means that on the right-hand side of the equal symbol. Type constructors and data constructors can share the same names.

Sum types These represent a value that can only be one of several different yet fixed types. In Accelerate, `Tree` is such an example:

```haskell
data Tree a = Leaf a | Forest [Tree a]
```

Listing 3. *Sum type example.*

The `Tree` ADT contains two different constructors, namely a leaf and a list of trees, which are parameterized over the type `a` of data stored in the leaf. The `Tree` data type can only represent one of two constructors at one time, e.g. `Tree Int` can be either `Leaf Int` or `Forest [Tree Int]` but not both. Sum types types are similar to union types in C.

Product types Unlike the sum types, these ones represent a value that contains multiple data, each of which can have different types. Consider a desugared array shape in Accelerate, which is defined as

```haskell
data Sh sh e = Shape (ShapeR sh) sh
```

Listing 4. *Product type example.*

It depicts that each desugared shape must have two fields in its constructor, namely the representation of the shape (`ShapeR sh`) and a set of pointers (`sh`) that can query the length of each dimension. Product types are similar to structure types in C.

Single constructors As the name implies, single-constructor ADTs only contain one constructor, which is often used to add semantic meaning. In Accelerate, a good example is implementing the zero-dimension shape:

```haskell
data Z = Z
```

Listing 5. *Single constructor example.*

The above code makes an ADT represent the zero-dimension shape for data, e.g., a single scalar. Though single scalars are usually stored as one-dimension arrays with length one, using `Z` as the shape is also fine.

Recursive types These are critical to enriching the representation of ADTs. They allow for the creation of complex, self-referential structures, such as the Haskell snoc-list [34], and higher ranks of arrays in Accelerate:

```haskell
data tail :. head = tail :. head
```

Listing 6. *Recursive type example.*

The above ADT recursively nests multiple singleton or complex data constructed by `:.` together, with left associativity to build higher shape representations, which is also a special case of the product type. For example, the shape of a 10 by 5 by 3 matrix can be represented as `(Z:.10:.5:.3)` or with parenthesis `(((Z:.10):.5):.3)` that both have the same type `(Z:.Int:.Int :.Int)`.

Enumerated types These types are special cases of the sum type, where each constructor takes no arguments, which is often used to define a fixed set of possible values. Similar to single constructors, they are used to add semantic meaning or labels in Accelerate, for instance:

```haskell
data Check = Bounds | Unsafe | Internal
```

Listing 7. *Enumerated type example.*

`Check` provides a series of tags to mark the type of runtime errors, which enables better output information for such errors.

2.4.2 GADTs

Generalized algebraic data types (GADTs) extend the concept of ADTs by allowing more explicit control over constructor types. By using them, we can specify more complex type relationships and constraints. In Accelerate, this feature is particularly beneficial since its biggest selling point is type safety. McDonell et al. [41] were able to lower Accelerate's surface language down to low-level register languages while preserving the type safety by using GADTs and type-safe interfaces for code generation. In their work, this approach helped to minimize the application compile-time errors. However, type-safe interfaces for generating GLSL code are beyond the scope of this thesis; we include only the type-safety for Accelerate expression terms. Specifically, we simply append generated strings together, which results in the loss of type information.

Here is an example of using the GADT in Accelerate to guarantee more precise types for constructors:

```haskell
data PrimFun sig where
    -- operators from Num
    PrimAdd  :: NumType a -> PrimFun ((a, a) -> a)
    PrimSub  :: NumType a -> PrimFun ((a, a) -> a)
    PrimMul  :: NumType a -> PrimFun ((a, a) -> a)
    PrimNeg  :: NumType a -> PrimFun (a      -> a)
    PrimAbs  :: NumType a -> PrimFun (a        -> a)
    PrimSig  :: NumType a -> PrimFun (a        -> a)
    -- operators from Integral
    PrimQuot     :: IntegralType a -> PrimFun ((a, a)    -> a)
    PrimRem      :: IntegralType a -> PrimFun ((a, a)    -> a)
    PrimQuotRem  :: IntegralType a -> PrimFun ((a, a)    -> (a, a))
    PrimIDiv     :: IntegralType a -> PrimFun ((a, a)    -> a)
    PrimMod      :: IntegralType a -> PrimFun ((a, a)    -> a)
    PrimDivMod   :: IntegralType a -> PrimFun ((a, a)    -> (a, a))
    ...
```

Listing 8. *`PrimFun` GADT.*

`PrimFun` is a GADT that represents primitive scalar operations. Each operation takes different argument types, parameterizing `PrimFun` over different `sig` types. For instance, primitive addition of two integer numbers is represented as `PrimAdd TypeInt` (simplified), with the type `PrimFun ((Int,Int) →Int)`. Though this does not directly guarantee type safeties, it exposes more extensive types for such constructors, enabling any function that operates on this data type to perform more rigorous type checks.

The same situation happens in environment GADTs in Accelerate:

```haskell
data Env f env where
    Empty :: Env f ()
    Push  :: Env f env -> f t -> Env f (env, t)
```

Listing 9. *`Env` GADT.*

`Env` constructs a series of data to form a parameterized data type, where `f` is a partially applied data type or a data type that takes type arguments (known as higher-rank types), and `t` is a concrete data type. Take `Identity` [24, §Data.Functor.Identity] as an example. It is a data type that takes a singleton type as the argument and returns exactly the same type, as its name suggests, which is a higher-rank type. On the data level, it simply stores the original values. So, if we have `Push (Push Empty Identity 2.5) Identity 3`, its type would be `Env Identity (((),Float),Int)`. Similar to the previous GADT, although this one only exposes rich type information, any function that takes it should leverage this information to ensure type safety.

In general, ADTs cannot achieve such delicate type controls.

2.4.3 Record notation

Record notation is a Haskell syntax for defining data types with named fields. It makes constructing, accessing, and updating these data types easier. Due to these features, record notation is frequently used in Accelerate. The desugared variable data type in Accelerate is defined as:

```haskell
data Var s env t = Var { varType :: s t, varIdx :: Idx env t }
```

Listing 10. *Record notation example.*

It is parameterized over `env`, a partially applied type or a higher-rank type – `s`, and a type `t`. Assume `var` is the data. To get the `varType` from it, we can write `varType var`. Or, to update a field, we can do `var { varType=... }`.

3 Inside Accelerate

The old pipeline of Accelerate can be found in the GitHub repository `AccelerateHS/accelerate` [1], and `AccelerateHS/accelerate-llvm` [2] provides implementations for all three backends on the old pipeline. For the new pipeline, the relevant repositories are `dpvanbalen/accelerate` [6] and `dpvanbalen/accelerate-llvm` [7] under the branch *new-pipeline*. Note that only the CPU backend is implemented for the new pipeline. The Vulkan backend has been implemented exclusively for the new pipeline to narrow the scope of this thesis.

As shown in Figure 3, to have a functional backend for Accelerate on the new pipeline, it is necessary to implement the desugaring, compiling, and executing stages. These stages are centred around backend-specific operations, hardware-specific code generation, and the runtime of the Vulkan backend. As their names suggest, each stage serves a distinct purpose. Since fusion is primarily an optimization, it is not required for a functional implementation and is outside the scope of this thesis.

In Accelerate, several basic data types provide interfaces for storing common data across multiple modules or serve as components of other data types in Accelerate. In addition, a large number of data types construct the Accelerate program in different levels or tiers. In this section, we will introduce the foundational Accelerate elements involved in the Vulkan backend.

3.1 Tuples

Tuples are commonly used in Accelerate among other data types, and most of them parameterize over tuple types. It is defined as nested pairs:

```haskell
data TupR s a where
    TupRunit    ::                           TupR s ()
    TupRsingle  :: s a                    -> TupR s a
    TupRpair    :: TupR s a -> TupR s b -> TupR s (a, b)
```

Listing 11. *`TupR` data type, which constructs elements in nested pairs.*

`TupR` is constructed using the `TupRsingle` constructor, where `s` is a partial-applied type or higher-rank type, and `a` is a data type. This data type parameterizes over the type of nested elements to form a tuple of types.

In Addition, only binary tuples are used within internal Accelerate. Therefore, built from `TupRunit`, each element in the tuple is appended to it using `TupRpair` as the second argument starting from the first one of the tuple. Recalling the higher-rank type `Identity` from §2.4.2, if we construct data `TupRpair (TupRpair TupRunit (TupRsingle (Identity 2.5))) (TupRsingle (Identity 5))` for example, its type is `TupR Identity (((), Float),Int)`.

Though in the tuple, type `a` can vary, the higher-rank type `s` must remain consistent across all elements. Note that constructing a single value (assuming an `Int` value) by using a standalone `TupRsingle` (of type `TupR s Int`) and a `TupRpair` (of type `TupR s ((),Int)`) with a `TupRunit` and a `TupRsingle` differs from each other, since the former constructs a unary while the latter builds a binary tuple. Only the latter is accepted inside Accelerate.

3.2 Environments

As illustrated in Listing 9, environments are parameterized over a higher-rank type and a tuple of types. Along with *de Bruijn* indices [8], elements in the environment can be queried.

A de Bruijn index is a mathematical notation used to represent lambda calculus expressions without using variable names. It specifies the position of a variable's binding lambda in terms of the number of intervening lambdas. For example, with de Bruijn indices, lambda calculus $\lambda x.\lambda y.\lambda z.\ x\ y\ z$ can be represented as $\lambda\lambda\lambda\ 2\ 1\ 0$.

Pushing a new element into the environment is similar to using lambda calculus to bind a local variable, so to query it we can use de Bruijn indices. Instead of denoting each element with a numeric number, Accelerate uses Peano numbers [20]. The `Idx` data type provides this functionality:

```haskell
data Idx env t where
    ZeroIdx ::                Idx (env, t) t
    SuccIdx :: Idx env t -> Idx (env, s) t
```

Listing 12. *`Idx` data type, which constructs a de Bruijn index.*

In this data type, data constructors are recursively stacked together where `ZeroIdx` points to the correct location of the desired element with type `t` in the `env` type parameter. Recalling the environment example we introduced in §2.4.2, which is `Push (Push Empty Identity 2.5) Identity 3` with type

`Env Identity ((((),Float),Int)` where `env` is `(((),Float),Int)`, we make a index to the floating number 2.5 as `SuccIdx ZeroIdx`, and its type is `Idx (((),Float),Int) Float`. They are both perfectly parameterized over the same environment type `env`, which can guarantee the type safety and eliminate the out-of-bound indexing when querying a member.

By recurring through the environment with the typed index, the expected value can be fetched. This process is called the projection:

```haskell
prj' :: Idx env t -> Env f env -> f t
prj' ZeroIdx       (Push _   v) = v
prj' (SuccIdx idx) (Push env _) = prj' idx env
```

Listing 13. *Environment projections.*

Apart from the regular environment, Accelerate also has a partial environment type – `PartialEnv`. It allows missing elements but can still have a consistent type parameter with a regular environment.

```haskell
data PartialEnv f env where
    PEnd  :: PartialEnv f env
    PPush :: PartialEnv f env -> f t -> PartialEnv f (env, t)
    PNone :: PartialEnv f env ->        PartialEnv f (env, t)
```

Listing 14. *Partial environments.*

Listing 14 presents that `PartialEnv` has three data constructors. Compared to the regular environment (as shown in Listing 9), `PEnd` functions similarly with `Empty` but can type any environment parameter in its type argument `env`, which means that building an environment does not have to start from an empty tuple. `PPush` works as same as `Push`. `PNone` represents a missing value, and there is no restriction for its type `t`, resulting in it being able to push any type in the tuple. For example, `Push (Push (Push Empty Identity 2.5) Identity 3) Identity 3.5` has a regular environment type `Env Identity ((((),Float),Int),Float)`, but using `PNone (PPush PEnd Identity 3)` can give us the same type parameter `env`.

3.3 Scalar types

Scalar types in Accelerate are classified into two categories: `VectorType` and `SingleType`.

```haskell
data ScalarType a where
    SingleScalarType :: SingleType a         -> ScalarType a
    VectorScalarType :: VectorType (Vec n a) -> ScalarType (Vec n a)
```

Listing 15. *`ScalarType` data type, which constructs types for basic elements.*

The former constructs the type of vectors (constructed by `Vec` data type) which are only used in SIMD expressions, but this is not commonly used within Accelerate and can be replaced by using regular expressions with normal arrays without significant performance impact. Therefore, both `VectorType` and `Vec` won't be implemented and supported in the thesis, while the latter – `SingleType` is more interesting and valuable to be researched here.

```haskell
type TypeR = TupR ScalarType

data SingleType a where
    NumSingleType :: NumType a -> SingleType a
data NumType a where
    IntegralNumType :: IntegralType a -> NumType a
    FloatingNumType :: FloatingType a -> NumType a

-- Integral types supported in array computations.
data IntegralType a where
    TypeInt      :: IntegralType Int
    TypeInt8     :: IntegralType Int8
    TypeInt16    :: IntegralType Int16
    TypeInt32    :: IntegralType Int32
    TypeInt64    :: IntegralType Int64
    TypeWord     :: IntegralType Word
    TypeWord8    :: IntegralType Word8
    TypeWord16   :: IntegralType Word16
    TypeWord32   :: IntegralType Word32
    TypeWord64   :: IntegralType Word64
-- Floating-point types supported in array computations.
data FloatingType a where
    TypeHalf     :: FloatingType Half
    TypeFloat    :: FloatingType Float
    TypeDouble   :: FloatingType Double
```

Listing 16. *Type constructors for single scalar numbers.*

As depicted by Listing 16, integers and floating-point numbers are constructed separately. They are used to identify data structures to represent which type of numbers. For example, `(SinlgeScalarType NumSingletype IntegralNumType TypeWord)` has type `SingleType Word`, where `Word` is type unsigned integer. Because acquiring the type parameter is sometimes difficult, this also provides an explicit way to get the type of the representation. Together with `TupR`, we have nested tuples to denote any data type that contains multiple `SingleType` data.

3.4 Primitive types

Apart from scalar types, primitive types are also used in primitive expressions in Accelerate. They are defined as:

```haskell
type TAG = Word8
type PrimBool     = TAG
type PrimMaybe a = (TAG, ((), a))
```

Listing 17. *Primitive types and `TAG`.*

In the above code snippet, `TAG` is a type synonym for `Word8` (unsigned 8-bit integer), i.e. a new name for the existing type. It is used as the condition for flow-controls and loops, e.g. if-else, switch-case, and for-while expressions. Thus, `PrimBool` is also a type synonym for `TAG`.

Additionally, type `PrimMaybe` is special since it constructs an irregular flow-control for data, which is unlike the if-else expression. If `TAG` is zero in the data field, `PrimMaybe` encloses a `Nothing` [31] -like structure, which is the left element in the pair; otherwise, it contains a type `a` data. To support it, the backend should either discard the value constructed by `PrimMaybe` or return an undefined value if it represents a `Nothing`, which can be done by using regular flow-controls and special handling for the pointless return.

3.5 Buffers

Every piece of numeric value is stored in a buffer during execution, making it one of Accelerate's most critical components. A buffer represents an array by storing each field of its SoA [49] (structure of arrays) separately in flat memory, where the shape and dimensional information are stored in independent buffers as scalars. Not just arrays but even scalar values are placed in buffers, except their length is one.

In Accelerate, we have two types of buffer, namely `Buffer` and `MutableBuffer`. As their names suggest, the former is the immutable buffer. Having these separate data types can avoid unexpected buffer modifications. They are defined as:

```haskell
-- Distributes a type constructor over the elements of a tuple.
type family Distribute f a = b where
    Distribute f () = ()
    Distribute f (a, b) = (Distribute f a, Distribute f b)
    Distribute f a = f a

-- Mapping from scalar type to the type as represented in memory.
type family ScalarArrayDataR t where
    ScalarArrayDataR (Vec n t) = t
    ScalarArrayDataR t         = t

newtype Buffer e = Buffer (UniqueArray (ScalarArrayDataR e))
-- A structure of buffers represents the SoA of an array.
type Buffers e = Distribute Buffer e
newtype MutableBuffer e = MutableBuffer (UniqueArray (ScalarArrayDataR e))
type MutableBuffers e = Distribute MutableBuffer e
```

Listing 18. *Accelerate buffer types.*

As depicted in Listing 18, `Buffer` data type works as a wrapper for `UniqueArray` which is an Accelerate-specific pointer to the actual space that contains the data. On the other hand, to construct the whole SoA of an array, we have `Buffers`. It distributes the `Buffer` type to a regular Haskell tuple containing multiple buffers with each representing one field of the SoA. Assuming we have a multi-dimensional array storing a ternary tuple of type `(Float,Half,Int)` as each element, then it will be represented by a `Buffers` data of type `Buffers ((((),Float),Half),Int)`. Within Accelerate IR, `(Float,Half,Int)` will be transformed to `((((),Float),Half),Int)`, and `Distribute Buffer ((((),Float),Half),Int)` ∼ `((((),Buffer Float), Buffer Half),Buffer Int)`. In general, an array of tuples will be transformed into a tuple of arrays throughout the internal Accelerate.

Not only buffer data types, module `Data.Array.Accelerate.Array.Buffer` also includes a series of functions to manage and access buffers. As for the special pointer – `UniqueArray`, module `Data.Array.Accelerate.Array.Unique` exposes several functions to allocate and deallocate such an object, and convert the pointer into the ordinary Haskell `Foreign.Ptr` [39] so that users can manipulate the corresponding memory space.

3.6 Variables

Variables in Accelerate are represented based on `Var` and `Vars` type, which are record notations. They are defined as:

```haskell
data Var  s env t = Var { varType :: s t, varIdx :: Idx env t }
type Vars s env   = TupR (Var s env)
```

Listing 19. *Basic variable types.*

Each variable has a field to mark the type with a higher-rank type `s` and a field to identify the referenced position in the environment. It is also parameterized over the corresponding environment that it points to. For `Vars`, it gives the partial applied `Var` to `TupR`, forming a tuple structure for `Var` data.

To introduce new local variables, which usually happens when passing arguments to a function, we have data type `LeftHandSide` in Accelerate. As its name suggests, `LeftHandSide` is used as a binding point which is the part on the left-hand side of the equality when calling a let-binding. Unlike binding a value to a named local variable, `LeftHandSide` indicates how to push new elements to the existing environment.

```haskell
data LeftHandSide s v env env' where
    LeftHandSideSingle
        :: s v
        -> LeftHandSide s v env (env, v)
    LeftHandSideWildcard
        :: TupR s v
        -> LeftHandSide s v env env
    LeftHandSidePair
        :: LeftHandSide s v1       env   env'
        -> LeftHandSide s v2       env'  env''
        -> LeftHandSide s (v1, v2) env  env''
```

Listing 20. *Basic variable types.*

As presented in Listing 20, we have three constructors which are similar to `TupR` (depicted in Listing 11), to denote the similar tuple type structure. `LeftHandSide` is parameterized over the order and structure of the local binding. which will be reflected in the type of new environment (`env'`). Among these constructors, `LeftHandSideSingle` contains a single element of type `s v`, and exposes its type `v` by nesting `v` with the original environment using a tuple. Similar to the environment type, to build from the ground, `LeftHandSideWildcard` is used to contain wildcard variables (e.g. `()` – `TupRunit`) such that the environment type remains intact. Finally, `LeftHandSidePair` just recursively combines all single elements to form a new environment type. For example, to make a local binding for one variable in the environment, we can write `LeftHandSidePair (LeftHandSideWildcard TupRunit) (LeftHandSideSingle Identity 5)`, so that its type becomes `LeftHandSide Identity ((),Int) env (env,Int)`. Note that this data type does not build a new environment; it only exposes types for other functions to ensure type safety. Users should implement a function to update the environment themselves.

Listing 21 shows `ExpVar` and `ExpVars`. They represent the expression type within internal pipelines as their results are either in `ScalarType` type or a nested tuple of `ScalarType` type corresponding to the expression of tuple arrays. To bind values from expressions, and introduce these local variables to other expressions, `ELeftHandSide` is implemented.

```
type ELeftHandSide = LeftHandSide ScalarType
type ExpVar        = Var ScalarType
type ExpVars env   = Vars ScalarType env
```

Listing 21. *Expression types.*

Another special internal variable type is the ground type. Ground variables are those values that are stored in the remote memory. In general, the keyword "Ground" means something placed in the device's memory. Similar to expression variable types, Listing 22 illustrates ground variable types with similar type definitions. Additionally, `GroundR` identifies whether a ground variable is a scalar value or an array in the buffer.

```
-- Ground values are buffers or scalars.
data GroundR a where
    GroundRbuffer :: ScalarType e -> GroundR (Buffer e)
    GroundRscalar :: ScalarType e -> GroundR e

type GLeftHandSide  = LeftHandSide GroundR
type GroundVar      = Var  GroundR
```

```haskell
type GroundVars env = Vars GroundR env
```

Listing 22. *Gound variable types.*

In general, `Var` and `Vars` are parameterized over different variable representations to form different Accelerate variables, while representations are no more than identifiers to mark types and contain some metadata. The commonality across all variables is that they all have a field for de Bruijn indices of the actual location in the environment storing the value.

3.7 Shapes

Shapes in Accelerate are represented by `ShapeR` which only preserved the structure instead of actual numbers for each dimension. `ShapeR` and common shapes are defined as follows:

```haskell
data ShapeR sh where
    ShapeRz     :: ShapeR ()
    ShapeRsnoc :: ShapeR sh -> ShapeR (sh, Int)

-- Synonyms for common shape types
type DIM0 = ()
type DIM1 = ((), Int)
type DIM2 = (((), Int), Int)
type DIM3 = ((((), Int), Int), Int)
```

Listing 23. *Shape representations.*

The shape data type takes nested pairs of type `Int` to construct the shape representation. As mentioned in §2.4.2, the surface language of Accelerate provides a set of data types to assign the shape for an array. Take the same example `(Z:.10:.5:.3)` from that section; it will be represented as `ShapeR ((((),Int),Int),Int)` inside Accelerate's pipeline, while the actual value of each dimension is stored in the environment and can be accessed by given de Bruijn indices.

Though shapes are stored as nested tuples to reflect the structure, they can be converted to and from ordinary linearized numbers. Meaning that it is possible to access a high-dimensional array given its linearized indices. Also, it is one of the critical functionalities that any backend should support. These conversions are written in the file `Representation/Shape.hs` of Accelerate with the name `toIndex` and `fromIndex`.

3.8 Arrays

Similar to the shape, arrays are represented by `ArrayR` data type where actual elements are obtained in the environment.

```haskell
data Array sh e where
    Array :: sh           -- extent of dimensions = shape
          -> Buffers e    -- array payload
          -> Array sh e

-- Type witnesses shape and data layout of an array
data ArrayR a where
    ArrayR :: { arrayRshape :: ShapeR sh
              , arrayRtype  :: TypeR e
              }
           -> ArrayR (Array sh e)

type ArraysR = TupR ArrayR
```

Listing 24. *Array representations.*

The above code snippet depicts the structure of the array representation. `Array` is parameterized over shape and element type of `Buffers` containing the actual shape numbers in nested tuples, and `Buffers` data. Though it hosts the actual shape rather than shape representations, this is only available in surface language to interact with the array. When diving deep into internal pipelines, we only use `Array`'s type to guarantee the type-safety for other data types, instead of its data. `ArrayR` data type is defined using the record notation, which captures `ShapeR` and `TypeR`. And it is parameterized over `Array` type.

Array instruction defined in module `AST.Operation` of Accelerate types the operation of fetching a single element from the external environment directly or indirectly. As shown in Listing 25, it has two constructors: `Index` and `Parameter`. The former types an array-indexing, where its type parameter `Int` $\rightarrow$ `e` means returning an element `e` from the array when given a linearized index. The argument of `Index` is the de Bruijn index of a buffer, which is parameterized over a buffer environment. `Parameter` constructs the operation of fetching a scalar element directly from the external environment with type `()` $\rightarrow$ `e` meaning that returns element `e` given nothing.

```haskell
data ArrayInstr benv t where
    Index     :: GroundVar benv (Buffer e) -> ArrayInstr benv (Int -> e)
    Parameter :: ExpVar benv e              -> ArrayInstr benv (()  -> e)
```

Listing 25. *Array instructions.*

3.9 Arguments

When desuaring the surface Accelerate program into IRs, part of the program will be represented as arguments for some operations. This involves not only arrays but also expressions and functions from the user interfaces.

```haskell
data Arg env t where
```

```
    ArgVar      :: ExpVars env e           -> Arg env (Var' e)
    ArgExp      :: Exp env e               -> Arg env (Exp' e)
    ArgFun      :: Fun env e               -> Arg env (Fun' e)
    ArgArray    :: Modifier m
                -> ArrayR (Array sh e)
                -> GroundVars env sh
                -> GroundVars env (Buffers e)
                -> Arg env (m sh e)
```

Listing 26. *Argument types.*

As listed above, we have four different constructors to contain arguments. Each of them parameterizes over a corresponding type. The `ArgArray` is the most common one, as it builds metadata for arrays. Its first field marks the access permission (depicted in Listing 27) for corresponding buffers. As mentioned in §3.7, the number of each dimension is stored in the environment as a scalar value. And the Third field of `ArgArray` is the place that provides an interface to get the number, which contains scalar ground variables which have de Bruijn indices to query concrete values from the environment. As for the fifth one, it consists of de Bruijn indices for ground buffers.

```
data Modifier m where
    -- Read-only.
    In  :: Modifier In
    -- Write-only.
    Out :: Modifier Out
    -- Read and write.
    Mut :: Modifier Mut
```

Listing 27. *Array access controls.*

To chain individual arguments together, we have operator `(:>:)`. It is parameterized over a special function type, namely `s→t`.

```
data PreArgs a t where
    ArgsNil :: PreArgs a ()
    (:>:)   :: a s -> PreArgs a t -> PreArgs a (s -> t)
infixr 7 :>:

type Args env = PreArgs (Arg env)
```

Listing 28. *Arguments.*

3.10 Program terms

The Accelerate program that will be desugared is represented by a series of program-level terms. This means that though users create various Accelerate programs, deep in the IR level, they are replaced with certain combinations of corresponding program-level dialects. These program-level terms or dialects are wrapped in `PreOpenAcc` data type. Although it has the same name with array-level terms `PreOpenAcc`, they share little similarity with each other. Reflecting on Figure 3, this PreOpenAcc is the “Named” IR. Unlike Operation IR, it only contains general information about the program computation without details, e.g. memory management and kernel executions.

```haskell
data PreOpenAcc (acc :: Type -> Type -> Type) aenv a where
    -- Local non-recursive binding.
    Alet        :: ALeftHandSide bndArrs aenv aenv'
                -> acc             aenv  bndArrs
                -> acc             aenv' bodyArrs
                -> PreOpenAcc acc aenv  bodyArrs

    -- Variable bound by a 'Let', represented by a de Bruijn index.
    Avar        :: ArrayVar        aenv (Array sh e)
                -> PreOpenAcc acc aenv (Array sh e)

    -- Tuples of arrays.
    Apair       :: acc             aenv as
                -> acc             aenv bs
                -> PreOpenAcc acc aenv (as, bs)
    Anil        :: PreOpenAcc acc aenv ()

    -- Array-function application.
    Apply       :: ArraysR arrs2
                -> PreOpenAfun acc aenv (arrs1 -> arrs2)
                -> acc              aenv arrs1
                -> PreOpenAcc  acc aenv arrs2

    -- If-then-else for array-level computations.
    Acond       :: Exp             aenv PrimBool
                -> acc             aenv arrs
                -> acc             aenv arrs
                -> PreOpenAcc acc aenv arrs

    -- Value-recursion for array-level computations.
    Awhile      :: PreOpenAfun acc aenv (arrs -> Scalar PrimBool)
                -> PreOpenAfun acc aenv (arrs -> arrs)
                -> acc              aenv arrs
                -> PreOpenAcc  acc aenv arrs

    -- Array inlet.
    Use         :: ArrayR (Array sh e)
                -> Array sh e
                -> PreOpenAcc acc aenv (Array sh e)

    -- Capture a scalar (or a tuple of scalars) in a singleton array.
    Unit        :: TypeR e
                -> Exp             aenv e
                -> PreOpenAcc acc aenv (Scalar e)

    -- Change the shape of an array without altering its contents.
    Reshape     :: ShapeR sh
                -> Exp             aenv sh
                -> acc             aenv (Array sh' e)
                -> PreOpenAcc acc aenv (Array sh e)

    -- Construct a new array by applying a function to each index.
    Generate    :: ArrayR (Array sh e)
                -> Exp             aenv sh
                -> Fun             aenv (sh -> e)
```

```
                 -> PreOpenAcc acc aenv (Array sh e)

    -- Generalised forward permutation.
    Permute      :: Fun            aenv (e -> e -> e)
                 -> acc            aenv (Array sh' e)
                 -> Fun            aenv (sh -> PrimMaybe sh')
                 -> acc            aenv (Array sh e)
                 -> PreOpenAcc acc aenv (Array sh' e)
    ...
```

Listing 29. *Program terms data type.*

In Listing 29, we only show those constructors that are currently used. The full list can be found in module `Data.Array.Accelerate.AST`. Computation dialects that can be replaced by `Generate` and `Permute`, and those that are not supported yet in the general Accelerate pipeline, are not included. These operations interact with surface languages and are parameterized over the returned element type `a`:

- **Tuples**: Unlike `Buffers`, `Apair` and `Anil` have nothing to do with the SoA of arrays. They are used to wrap multiple arrays as a binary nested tuple similar to `TupR`, which might be used in array loops and array control-flows.

- **Local Variables**: To bind and query local arrays, `Alet` and `Avar` are introduced. They both interact with the environment. The first and second arguments of `Alet` are the bound expression and the bound scope.

- **Shapes**: `Reshape` constructor marks the array computation of reshaping. The second argument is the expression to compute the new dimensional number for the new shape. The third one is the original array.

- **Array Inlet**: Though `Use` does not return a new data type for an embedded array since its returned type `a` is in the same type with its second argument, it marks a possible data transfer.

- **Singleton Array**: As the name of `Unit` implies, it is used to mark a singleton array with only one scalar element.

- **Application**: `Apply` marks the application for an Accelerate function given the function arguments at its third position.

- **Flow-Control**: To make the program switch over different branches, we have `Acond`. Its first argument is an expression giving the condition. Its second and third arguments contain other Accelerate programs, which fall into the true and false branches separately.

- **Loop**: Similar to `Acond`, `Awhile` also branches over different conditions, and keeps updating the array `arrs` until it does not satisfy the condition.
- **Generate** [53, §Initialisation]: This constructor represents the program generating each element for the array given a shape expression and a scalar expression to compute the value according to each element index.
- **Permute** [53, §Forward permutation (scatter)]: `Permute` (forward permutation) contains the combination expression as the first argument of updating the default array given its original value and a value from the source array according to the permutation expression. The second argument of `Permute` is the default array. The third one is the index permutation expression, which maps a position in the source array to a new position of the default array, or just drops the current index and doesn't update the default array. The source array is given as the fourth argument.

Additionally, program-level `PreOpenAcc` is also the entry point for all Accelerate backends, where transformations and analysis of backends start from here.

3.11 Array terms

Though backend-specific operations are the core for building computations on the device, the whole Accelerate program still needs supplementary information to run, such as the memory allocation, data transfer, executing the built computation. `PreOpenAcc` is such the data type to contains all these operations, namely array terms, including the execution for backend-specific operations. Though it shares the same name with the program-level term data type, they are totally different. Note that this data type does not execute any operation. Instead, it just represents the internal process to run the computation and provides certain type parameters. As shown in Figure 3, this is actually the Operation IR, which contains details about how the operation should go, and it will be executed in the execution module.

```haskell
data PreOpenAcc (op :: Type -> Type) env a where
    -- Executes an executable operation.
    Exec    :: op args
            -> Args env args
            -> PreOpenAcc op env ()

    -- Returns the values of the given variables.
    Return  :: GroundVars env a
```

```
            -> PreOpenAcc op env a

    -- Evaluates the expression and returns its value.
    Compute :: Exp env t
            -> PreOpenAcc op env t

    -- Local binding of ground values.
    Alet    :: GLeftHandSide bnd env env'
            -> Uniquenesses bnd
            -> PreOpenAcc op env  bnd
            -> PreOpenAcc op env' t
            -> PreOpenAcc op env  t

    -- Allocates a new buffer of the given size.
    Alloc   :: ShapeR sh
            -> ScalarType e
            -> ExpVars env sh
            -> PreOpenAcc op env (Buffer e)

    -- Buffer inlet.
    Use     :: ScalarType e
            -> Int -- Number of elements
            -> Buffer e
            -> PreOpenAcc op env (Buffer e)

    -- Capture a scalar in a singleton buffer
    Unit    :: ExpVar env e
            -> PreOpenAcc op env (Buffer e)

    -- If-then-else for array-level computations
    Acond   :: ExpVar env PrimBool
            -> PreOpenAcc op env a
            -> PreOpenAcc op env a
            -> PreOpenAcc op env a

    -- Value-recursion for array-level computations.
    Awhile  :: Uniquenesses a
            -> PreOpenAfun op env (a -> PrimBool)
            -> PreOpenAfun op env (a -> a)
            -> GroundVars     env a
            -> PreOpenAcc  op env a
```

Listing 30. *Array terms data type.*

As shown in Listing 30, `PreOpenAcc` consists of constructors for converting ordinary arrays to and from Accelerate arrays, local bindings, backend operation executions, array-level flow-controls and loops. This type is parameterized over the backend-specific operation type `op`. Unless stated explicitly, the following operations interact with memory space on the device side.

- **Backend Operation Execution**: `Exec` consists of a backend-specific operation that is awaiting to be executed on the device with arguments. Since its return type (which is `a`) is an empty tuple, instead of returning a value, it should directly write results to buffers after execution.

- **Expression Evaluation**: `Compute` constructor represents an expression that is awaiting execution on the host rather than the device and returns the evaluation result with type `t`.

- **Array Lift**: `Use` represents lifting a buffer from the host side to the device side. It explicitly tells when to transfer the data.
- **Array Return**: Unlike `Use`, `Return` does not trigger data transfer. It contains querying buffers from the environment given de Bruijn indices and returns them.
- **Array Allocation**: Given the shape and element type, `Alloc` marks the operation of allocating a new empty buffer on the device.
- **Singleton Array**: As mentioned before, scalars are actually captured in buffers with size one. To mark the action of allocating and writing such the buffer, `Unit` is created given the index for such a scalar.
- **Array-Level Local Binding**: `Alet` identifies the array-level binding operations. Its third argument introduces new arrays to the environment, whereas the fourth argument is the actual computation. As the binding value `bnd` might be changed during execution, `Uniquenesses` marks the buffer usage. After executing this constructor, the environment should stay intact.
- **Array-Level Flow-Control**: `Acond` is the array-level flow-control, where its first argument marks the index for the condition. The second and third array operations are the true and false branches, respectively.
- **Array-Level Loop**: `Awhile` constructs the loop-terminated function, the loop body, and the index for the initial value. Then it returns the final result when breaking the loop.

None of the above operations triggers the data transfer from the device to the host. This is because after executing all such operations, the very outer layer of the recursive `PreOpenAcc` data type should return the final value `a` which is either a single buffer or nested buffers stored on the device, while this value should be caught and fetched from the remote memory by the function who calls to execute `PreOpenAcc` data.

The Schedule (of type `SeqSchedule`) of Figure 3 is essentially the same as `PreOpenAcc` but parameterized over the backend-specific kernel type `kernel`. It will contain scheduled operations and call the runtime to execute them during the execution.

3.12 Modifications

Compared to the original Accelerate and Accelerate-LLVM, we have made a few modifications to those libraries. We added element size (or length) for buffers (shown in Listing 18), so that new buffer types looks like:

```haskell
data Buffer e = Buffer Int (UniqueArray (ScalarArrayDataR e))
type Buffers e = Distribute Buffer e
data MutableBuffer e = MutableBuffer Int (UniqueArray (ScalarArrayDataR e))
type MutableBuffers e = Distribute MutableBuffer e
```

Listing 31. *Modified buffer types.*

To make them compatible and work with other components, we also modified the functions that use them. Their length fields are filled when allocating new buffers, since only at this point can we know how many elements will be put inside each buffer. These modifications are important for the Vulkan backend because when the array indexing takes place in `PreOpenExp`, there is neither information about the length of the array (only buffer indices) nor is it possible to get such one in runtime. Indexed arrays are transferred to the remote (device) memory using their lengths.

4 The Vulkan backend

The main research goal of this thesis is to discover the possibility of implementing a Vulkan backend for Accelerate. As elaborated in Figure 3, a backend of the new pipeline should have support for the process within the right yellow, except that scheduling is automatically done by Accelerate's scheduler. This means a functional Vulkan backend is necessary to conduct desugaring, compiling, and executing internal IRs. The most important part of this process is to generate and compile GLSL code for each array computation since they are coded in higher order, whereas GLSL is a first-order programming language.

To run an Accelerate program, there are a few steps for the Vulkan backend to do. For starters, program-level `PreOpenAcc` (which is the "Named" IR) is desuagred into array-level `PreOpenAcc` (namely the Operation IR) using backend-specific operations. Then, Operation IRs are fused into clustered IRs (or Partitioned IRs), in which the fusion's behavior is controlled by implemented instances of the fusion class. Specifically, fusion not only combines multiple backend-specific operations as a whole but also tries to eliminate unnecessary array allocations. However, it is beyond the scope of this thesis and won't be enabled nor included in this thesis; thus, each Partitioned IR only contains individual operations. After that, a code generator takes these IRs to generate GLSL codes individually. These codes then are compiled into SPIR-V codes. Furthermore, global and local Vulkan contexts (also known as objects or resources) are created with the SPIR-V code. At this point, backend operations are replaced with Vulkan kernels (shown in Listing 34, a customized data type) containing local contexts. Finally, these IRs are automatically scheduled and executed by a runtime of the Vulkan backend. When executing them, both the local context and shared global context are called.

The full implementation of the Vulkan backend can be found in GitHub repository `largeword/accelerate-vulkan` [38]. The Vulkan-Haskell binding we use is `expipiplus1/vulkan` [22], which is maintained by a few Vulkan developers. It has good support for all the core functionalities of Vulkan from version 1.0 to 1.3, as well as almost all the extensions. We thus choose this one among all Vulkan-Haskell bindings.

4.1 Backend-specific operations

The first step that happens on the backend level is to desugar or transform the Accelerate program (program-level `PreOpenAcc`) into a series of backend-specific operations. These operations are dialects used to build the array computation on the hardware, which are extremely hardware- and platform-dependent. Furthermore, the operation set is different on different devices since various hardware has diverse implementations of the same array computation. For instance, a permutation of the program level can be desugared into a single backend-specific permutation, or decomposed into the combination of multiple array additions, productions, and other basic operations.

When creating the backend-specific operation set, we are actually designing array computation dialects on the hardware. Thus, we need to consider the overhead of conducting each computation on the hardware, as well as the difficulty of the device code generation for each dialect.

4.1.1 Vulkan operations

The `PreOpenAcc` (shown in Listing 30) takes a backend-specific operation type as its first type parameter – `op`. Then constructor `Exec` wraps such an operation set as its first argument. As such, we have made Vulkan-specific operations like:

```haskell
data VulkanOp t where
    VkGenerate :: VulkanOp (Fun' (sh -> t)
                          -> Out sh t
                          -> ())
    VkPermute  :: VulkanOp (Fun' (e -> e -> e)
                          -> Mut sh' e
                          -> Fun' (sh -> PrimMaybe sh')
                          -> In sh e
                          -> ())
```

Listing 32. *Vulkan-specific operations.*

`VulkanOp` does not contain any value in constructors; instead, it exposes type parameters to guarantee such safeties for argument types (mentioned in Listing 26). Each constructor is parameterized over different types:

- `VkGenerate` marks the operation of `Generate` (presented in Listing 29). `Fun' (sh -> t)` reflects the generating function in `Generate`, which takes an index in the type of the shape and gives a scalar value in the type of `t`. Parameterized over the same modifying mode `Out` (shown in Listing 27), `Out sh t` marks a write-only buffer containing elements of scalar type `t`, which is of shape `sh`.

- `VkPermute` is parameterized over a more complex type; it is corresponding to `Permute` (presented in Listing 29). `Fun' (e -> e -> e)` types a combination function that takes two arguments in the same type and returns a value. It elaborates on how the original and given values are combined to update the permuted array. Then `Mut sh' e` explicitly tells that the permuted array argument is of type `Arg env (Mut sh' e)` from Listing 26, which is the mutable buffer. `In sh e` works similarly to the mutable buffer type, which constrains the source input type. `Fun' (sh -> PrimMaybe sh')` guarantee the permutation function type, which maps the shape of the source (input) array to maybe the shape of the default (permuted) array that is taken as a mutable buffer. This mapping process reflects how each element from the input array is related to the destination array or is just simply dropped.

4.1.2 Desugaring

After having the backend operation set, according to the compiler pipeline (as shown in Figure 3), we need to desugar the program-level `PreOpenAcc` ("Named" IR, shown in Listing 29) into the array-level one (Operation IR, shown in Listing 30), and use `VulkanOp` to represent all the array computations. Thankfully, in module `Data.Array.Accelerate.Trafo.Desugar`, functions are implemented to automatically desugar all program-level operations into Operation IRs, except for array computations. Implementation of desugaring `Generate` and `Permute` among all array computations is the minimal requirement.

Though apart from `Generate` and `Permute`, other array computations (`Map`, `Scan`, `Fold`, etc.) are accommodated in the program-level IR (i.e. `PreOpenAcc`), they can be all desugared and represented by those two operations. That's why we only defined those two operations in the backend operation set.

To make the desugaring function work, implementing the instance `DesugarAcc VulkanOp` is necessary, but only the function `mkGenerate` and `mkPermute` are required as mentioned before. Though implementing more such desugaring functions for corresponding array computations can improve the parallelism and utilize the Vulkan device better, they are optional and won't be included in the thesis.

```haskell
instance DesugarAcc VulkanOp where
    mkGenerate :: Arg env (Fun' (sh -> t))
               -> Arg env (Out sh t)
               -> OperationAcc VulkanOp env ()
    mkGenerate f aOut = Exec VkGenerate (f :>: aOut :>: ArgsNil)

    mkPermute :: Arg env (Fun' (e -> e -> e))
              -> Arg env (Mut sh' e)
              -> Arg env (Fun' (sh -> PrimMaybe sh'))
              -> Arg env (In sh e)
              -> OperationAcc VulkanOp env ()
    mkPermute combF aMut idxF aIn = Exec VkPermute ( combF
                                                   :>: aMut
                                                   :>: idxF
                                                   :>: aIn
                                                   :>: ArgsNil)
```

Listing 33. *Desugaring surface programs.*

As elaborated in Listing 33, the function `mkGenerate` and `mkPermute` are defined. Since constructor `Exec` of Listing 30 accommodates backend operations and typed arguments, we can simply wrap the matching `VulkanOp` constructor and chained input arguments with `Exec` to complete the desugaring. This will tell the runtime to execute such operations on the Vulkan device during the execution.

Unlike Soest's thesis [50] about implementing a TensorFlow backend for Accelerate, where he desugared `Generate` and `Permute` into more basic operations, we halted this journey at a different point. This is because TensorFlow combines all array computations as a whole compute graph, resulting in no data transfer and execution overhead in between, and separating higher-order operations at the desugaring can save much work for later stages, while Vulkan executes each operation individually. Thus, we hope to wrap as many operations as possible within one kernel (or backend-specific operation). The downside, however, is the increasing work for code generation.

In general, during the process of lowering "Named" IRs into Operation IRs, we only need to interact with instance `DesugarAcc VulkanOp` implementing the function `mkGenerate` and `mkPermute` at minimal.

4.2 Vulkan kernels

After desugaring into Operation IR, we now obtain array computations represented in Vulkan-specific operations. Compared to executable Vulkan kernels, these operations are merely high-level representations of low-level device codes. Thus, we need to compile each representation into separate Vulkan executable objects, namely Vulkan kernels, to form Kernel IR reflected in Figure 3.

```haskell
data VulkanKernel env where
    VkKernel ::
             { kernelId         :: {-# UNPACK #-} !UID
               -- Contains the shapes of all buffers.
             , kernelShapes     :: [Idx env Int]
               -- Contains input buffer args.
             , kernelInArgs     :: [Exists (VulkanArg env)]
               -- Contains mutable buffer args.
             , kernelMutArgs    :: [Exists (VulkanArg env)]
               -- Contains output buffer args.
             , kernelOutArgs    :: [Exists (VulkanArg env)]
               -- Store pre-built local Vulkan objects.
             , kernelResources :: Lifetime (LocalVulkanResources env)
               -- Store scalars to compute the number of threads.
             , kernelThreads    :: [Idx env Int]
             }
             -> VulkanKernel env
```

Listing 34. *Vulkan kernels.*

During the code generation and compilation stage, `VulkanKernel` data are built for each individual Vulkan operation (of type `VulkanOp`). Though the generation and permutation operations work differently, their kernel can be contained by the same data type. Presented in Listing 34, each field of the record notation has a specific purpose:

- `kernelId` uniquely identifies a kernel by containing a UID.
- `kernelShapes` contains a list of de Bruijn indices parameterized over the kernel environment `env`, which is related to integer scalars representing dimensional numbers of all array arguments.
- `kernelInArgs`, `kernelMutArgs`, and `kernelOutArgs` are Vulkan arguments (explained in Listing 35), which obtain both scalars and buffers in the same list eliminated the type parameter (either `a` or `Buffer a`) by `Exists`. They contain the items that are taken and used by the

computation. Each field records a list of such items classified by its access mode (as shown in Listing 27).

- Since each Vulkan kernel builds an executable Vulkan object with supplementary context, `kernelResources` is such the field to contain these data. In between, `Lifetime` is a finalizer (introduced in §4.2.2) type managing the local context `LocalVulkanResources` (introduced in §4.2.2), which uses the garbage collection to automatically recycle non-shared resources.

- The threads field `kernelThreads` marks how many threads will be launched to execute the current kernel. Since the thread number is usually set according to the shape of one of the array arguments, we here use a list to contain all dimensional numbers reflecting the shape. Thus, the actual number is the multiplication among all the scalars in the list.

```
newtype VulkanArg env a = VulkanArg (GroundVar env a)
```

Listing 35. *Vulkan arguments.*

To contain buffers and scalars that are called by the Vulkan kernel, we have `VulkanArg` (shown in Listing 35). It is parameterized over ground types, meaning that it can represent both buffer and scalar elements. `VulkanArg (Var (GroundRbuffer st) idx)` is a buffer argument for the kernel, where `st` is the scalar type and `idx` is the de Bruijn index for the current variable pointing to the position in the environment. Similarly, `VulkanArg (Var (GroundRscalar st) idx)` means a scalar argument.

The overall steps for making Vulkan kernels are generating GLSL code, creating a Vulkan global context, compiling GLSL code, and building a Vulkan local context. Spawning a kernel for the permutation is more complex than that of the generation.

4.2.1 Compiling to GLSL

During the compiling stage, our goal is to compile `PreOpenExp` (the definition can be found in module `Data.Array.Accelerate.AST.Exp`) type data into the concrete GLSL code. `PreOpenExp` is at a lower level compared to the array-level `PreOpenAcc`, which contains element-wise computations rather than array-wise ones. It does not show up alone; usually, it comes with a typed environment containing local bindings. `PreOpenFun` recursively nests multiple `ELeftHandSide` (introduced in Listing 21) and one

PreOpenExp as a function, which guides to build a suitable environment to compile the PreOpenExp.

A complete GLSL snippet consists of headers (including the standard version and enabled extensions), local thread declarations, buffer bindings (inlets), and a main body. Though expressions compiled from PreOpenExp are the most important part of the main body, to make it work, we need to query the global index for the current thread and bind results from expressions to buffers at the end.

We have made several customized data types to aid the compiling process.

```haskell
newtype VarName t = VarName String
type VarNameTup t = TupR VarName t
type VarNameEnv env = Env VarName env
```

Listing 36. *Variable names.*

VarName (in Listing 36) stores a unique name for a variable, and its type parameter t can be used to type variable references in PreOpenExp. VarNameTup and VarNameEnv build a tuple of variable names and an environment separately.

```haskell
newtype ExpString t = ExpString String
type ExpStringTup t = TupR ExpString t
type ExpStringEnv env = Env ExpString env
```

Listing 37. *Compiled expressions.*

Similarly, we have made ExpString (in Listing 37) to store a compiled expression of PreOpenExp, and AInstr (in Listing 38) is for the array instruction. The first field of AInstr is the unique name of a called buffer or scalar variable; the second one is a Vulkan argument that is parameterized over an external buffer environment.

```haskell
data AInstr benv t = AInstr String (VulkanArg benv t)
type AInstrEnv benv = PartialEnv (AInstr benv) benv
```

Listing 38. *Compiled array instructions.*

Then to contain self-defined and manually implemented functions (as listed in Table 1), we have made the following ordered map type. Its key is the function name and the value is the function definition.

```haskell
type FuncMap = OMap String String
```

Listing 39. *Function maps*

To keep counting variables and give a unique name for each variable, we have made a type synonym `type VarCount = Int`, which will be put in the state for functions that depend on it.

In the following paragraphs, we will introduce steps to compile `PreOpenExp` into GLSL code, but due to the page limitation, we won't include the full implementation here. Those who might be interested can refer to the GitHub repository `largeword/accelerate-vulkan` [38] under the folder `Compile` in our implementation.

Data types As elaborated in §3.3, we have many scalar types in Accelerate. Each of them has a corresponding value type in GLSL, and we have implemented the support for all scalar types except for the `VectorScalarType`.

```haskell
integralTypeToString :: IntegralType a -> String
integralTypeToString t = case t of
    TypeInt    -> case bytesElt (TupRsingle (SingleScalarType (NumSingleType
     (IntegralNumType TypeInt)))) of
        4 -> error "32-bit GHC not supported, default Int should be Int64"
        8 -> "int64_t"
        s -> error "Int size " ++ show s ++ "-bit not supported"
    TypeInt8   -> "int8_t"
    TypeInt16  -> "int16_t"
    TypeInt32  -> "int32_t"
    TypeInt64  -> "int64_t"
    TypeWord   -> case bytesElt (TupRsingle (SingleScalarType (NumSingleType
     (IntegralNumType TypeWord)))) of
        4 -> error "32-bit GHC not supported, default UInt should be UInt64"
        8 -> "uint64_t"
        s -> error "UInt size " ++ show s ++ "-bit not supported"
    TypeWord8  -> "uint8_t"
    TypeWord16 -> "uint16_t"
    TypeWord32 -> "uint32_t"
    TypeWord64 -> "uint64_t"

floatingTypeToString :: FloatingType a -> String
floatingTypeToString t = case t of
    TypeHalf   -> "float16_t"
    TypeFloat  -> "float32_t"
    TypeDouble -> "float64_t"
```

Listing 40. *Compiling scalar types.*

Listing 40 shows how to compile each scalar type to a corresponding type in GLSL. The two exceptions here are `TypeInt` and `TypeWord`. In most environments, default integer and unsigned integer types are set to 64-bit lengths, but in GPUs, they are 32-bit. Thus, to make the length of data types on the host side consistent with that of the device, we enforce the default integer and unsigned integer on the host side explicitly converted to their 64-bit version on the device side and throw an error if they are not of 64-bit on the host.

Enabled extensions Native GLSL only supports a small set of functionalities and types. Adding extensions on the top level of the code enables the compiler to provide more flexibility and customized features for Vulkan shaders. In the Vulkan backend, we enable all these extensions for each shader:

- `ARB_separate_shader_objects`, which provides more flexibility and modularity in shader management of Vulkan;
- `EXT_nonuniform_qualifier`, which introduces dynamic buffer indexing, i.e. accepting ordinary integers as indexing numbers rather than constants;
- `EXT_shader_explicit_arithmetic_types`, which extend the range of supported data types for the shader;
- `EXT_shader_atomic_int64`, which allows atomic operations to take 64-bit integers;
- `EXT_shader_atomic_float`, which allows atomic operations to support single-precision (32-bit) and double-precision (64-bit) floating-point numbers. However, it only works for `atomicAdd`, `atomicExchange`, `atomicLoad`, and `atomicStore`.

Although they are the minimal requirement for the backend, not all of them are supported on any device. After enabling extensions in the GLSL code, we are required to also enable corresponding features in the logical device creation information of Vulkan to support such extensions. Before then, these features should be checked whether they are supported on the target device or not.

Global indices Within the GLSL code, we are required to set the local thread number for a working group, which can be fixed or dynamically given similar to buffer inlets. We can launch multiple working groups for the same shader to have more threads. Both generation and permutation take an index as the source for the computation, which makes it important to query a unique global index for each thread and then bind it to the compilation environment for later use. To achieve this, we first query the index by:

```
const int64_t globalID = int64_t(dot(vec3(gl_GlobalInvocationID),
                                vec3(1,
                                     gl_NumWorkGroups.x,
                                     gl_NumWorkGroups.y *
                                     gl_NumWorkGroups.x)));
```

Listing 41. *Computing global index.*

Since the index type in Accelerate is denoted by a 64-bit integer, we follow the same principle here.

Under some cases, the linearized `globalID` would be more useful if it is unfolded into multi-dimensional indices. Thus, we have made such a function:

```
compileFromIdx :: ShapeR t
               -> String
               -> VarCount
               -> (String, String, VarNameTup t, VarNameTup t, VarCount)
```

Listing 42. *Unfolding the global index.*

Given a shape representation, a prefix string, and a variable number, `compileFromIdx` returns a GLSL buffer inlet string, a statement to disassemble the global index, a tuple for variable names of multi-dimensional indices, a tuple for variables of dimensional numbers matching the shape representation, and an updated variable counting number. It executes two steps inside: First, creating buffer inlets to get each dimensional number according to `ShapeR`; Second, using modulos and integer divisions to recursively unfold the global index.

```
layout(set = 0, binding = 0, std430) buffer Dim0 { int64_t outDim0; };
layout(set = 0, binding = 1, std430) buffer Dim1 { int64_t outDim1; };
layout(set = 0, binding = 2, std430) buffer Dim2 { int64_t outDim2; };
```

Listing 43. *Buffer inlets from `compileFromIdx`.*

Supposing calling `compileFromIdx` to generate three indices for a three-dimensional array, where each of them matches a dimensional index, Listing 43 and Listing 44 show the buffer inlets and expressions statements separately. Each scalar buffer in Listing 44 represent a dimensional number for the array.

```
const int64_t outIdx2 = (globalID) % outDim2;
const int64_t outIdx1 = ((globalID) / outDim2) % outDim1;
const int64_t outIdx0 = (((globalID) / outDim2) / outDim1) % outDim0;
```

Listing 44. *Unfolding indices from `compileFromIdx`.*

After having these indices from the `VarNameTup`, we can then either push them to the environment or pass them to another function for further processing. For the former, the function `makeEnv` (as shown in Listing 45) takes a set of left-hand-sides and a typed tuple of variable names to build a new environment from an empty one recursively.

```
makeEnv :: LeftHandSide ScalarType t () env
        -> VarNameTup t
        -> VarNameEnv env
makeEnv lhs idx = go Empty lhs idx
  where go :: VarNameEnv env
```

```
            -> LeftHandSide s t env env'
            -> VarNameTup t
            -> VarNameEnv env'
         go env (LeftHandSideWildcard _) _
               = env
         go env (LeftHandSideSingle _) (TupRsingle varName)
               = Push env varName
         go env (LeftHandSidePair lhs1 lhs2) (TupRpair v1 v2)
               = go (go env lhs1 v1) lhs2 v2
         go _ _ _ = error $ "Tuple types do not match"
```

Listing 45. *Making environments.*

Compiling called buffers Buffers are called in two ways: direct calls and indirect calls. The former involves the data constructor `ArgArray` of `Arg` (as shown in Listing 26), where buffers are introduced with `VkGenerate` or `VkPermute` in the Partitioned IR. To generate GLSL code declaring buffer inlets, we have made the following function:

```
compileAddBufferArgs :: TypeR t -> String -> VarCount -> (String,
                                                          VarNameTup t,
                                                          VarCount)
```

Listing 46. *Declaring buffer inlets for GLSL code.*

Given a type representation of SoA, a prefix, and a variable count, `compileAddBufferArgs` returns a buffer inlets statement, a tuple of bound buffers' names, and a new variable count, which does a similar job to `compileFromIdx`. To fill one of the buffer arguments fields for the Vulkan kernel, we have built:

```
argBuffers :: Arg env (m sh e) -> [Exists (VulkanArg env)]
```

Listing 47. *Compiling buffer arguments.*

`argBuffers` takes an array argument and flattens its tuple of ground variables into a list of `VulkanArg` constructor. As for indirect calls of buffers, they come from array instructions (or array indexing). They are compiled to both the GLSL code and buffer arguments at the same time by recursing through `AInstrEnv` and only taking `PPush`-constructed values. Thus, we have made function `compileAInstrEnv` as follow:

```
compileAInstrEnv :: AInstrEnv benv
                 -> Int
                 -> (String, [Exists (VulkanArg benv)], Int)
```

Listing 48. *Compiling array instructions.*

Compiling regular expressions In PreOpenExp (the definition can be found in module Data.Array.Accelerate.AST.Exp), there are different constructors handling local bindings, variable references, indices slicing, indices conversions, flow-controls, primitive scalar operations, etc. To compile them into statements and expressions coded in GlSL, we have made a function compileStatement (as presented in Listing 49). Given a data of type PreOpenExp parameterizing over ArrayInstr benv (as shown in Listing 25), it would return a tuple of GLSL statements and compiled expressions.

```
compileStatement :: VarNameEnv env
                 -> PreOpenExp (ArrayInstr benv) env t
                 -> State (VarCount, FuncMap, AInstrEnv benv)
                      -- return (Statement, TupR Expression)
                      (String, ExpStringTup t)
```

Listing 49. *Compiling PreOpenExp*

Inside compileStatement, various pattern-matching cases handle constructors of the element-wise operation:

- The Let constructor introduces a local binding to the environment for the body expression given a ELeftHandSide (mentioned in Listing 21) and a bound expression. The bound expression should be compiled first, giving a statement and a tuple of compiled expressions. Then, the tuple and the left-hand-side are taken to update the original variable environment. This process is done by the function compileLhs, which declares and binds expressions to new variables, and then recursively pushes new variables to the environment according to the left-hand-side. After that, it will return a statement for variable declarations and a new environment. Finally, the body expression can be compiled by recursively calling compileStatement with the new environment. For instance, the expression let x = i+1 in x*x is compiled to int64_t e0_let = (1L) + (outIdx0); returning (e0_let) * (e0_let) as the expression.

- Evar stands for a variable reference containing a ExpVar. Thus, we can simply get a de Bruijn index from it and project the index to the environment fetching the name of the referenced variable.

- Constructor Foreign builds a backend-specific function, which is not used throughout Accelerate pipelines. The easy way to compile it is to call the function rebuildNoArrayInstr to recover a PreOpenFun from its fallback function. In the first step, its input expression is compiled and bound to newly declared variables. Then given new variables and the left-hand-side from PreOpenFun, function compileFun would push them to an empty environment and use it to recursively compile the `PreOpenExp` from `PreOpenFun`.

- For tuples, `Pair` constructs a pair of scalar expressions. The same environment should be used to compile them individually and then wrap their returned results as a pair. `Nil` is also a tuple constructor. It represents an empty tuple without any expression or value.

- SIMD operators are provided specifically for `VectorType`, so they are not included in our implementation for the reason mentioned in §3.3.

- For array indices manipulations, we have `IndexSlice` and `IndexFull`. The former slices an array into a sub-array by returning the sliced indices. The latter rebuilds the full shape from sliced indices. To implement such operations, we only need to type with constructors in `SliceIndex` data type of module `Data.Array.Accelerate.Array.Representation` and use the type construction there.

- `ToIndex` and `FromIndex` mark indices linearization and reverse linearization. Given a shape representation, a shape of the array, and multi-dimensional indices, `ToIndex` is supposed to calculate a linearized index. For example, we have an array of shape $(d_0, d_1, \ldots, d_n)$ and multi-dimensional indices $(i_0, i_1, \ldots, i_n)$, they should be compiled to a new index in compliance with the equation $((((0) \cdot d_0 + i_0) \cdot d_1 + i_1) \ldots) \cdot d_n + i_n$. As for `FromIndex`, it is compiled in the same way as how `compileFromIdx` converts the global index shown in Listing 44.

- The `Case` constructor identifies case-switches with its first argument as the condition. To compile it, we declare new variables to catch returns from each branch. Then, we regularly compile each case and the default case. Finally, the `switch` statement resembles all the branches where each branch has a returned variable binding statement. The return for the whole case expression is a tuple of returned variables.

- Condition expression – `Cond` is a simplified version of `Case`, so it can be compiled in the same way. For example, we have an expression `ifThenElse (i == 5) (i*i) (mod i 5)`; it is compiled to the following GLSL code.

```
int64_t e0_condRtn;
if (bool(uint8_t((5L) == (outIdx0)))) {
    e0_condRtn = (outIdx0) * (outIdx0);
} else { e0_condRtn = (outIdx0) % (5L); }
```

In the condition part of the if-statement, multiple type conversions nest with each other. This is because in Accelerate, `PrimBool` is declared as `Word8`, and logical functions only accept boolean values, so we explicitly convert it to avoid type-mismatch errors for such variables. Moreover, in GLSL, a `Word8` variable cannot take a boolean value or vice versa. To not introduce extra complex analysis while preserving the consistent type of Accelerate expressions, we convert the result back to `Word8` for every logical operation and convert the argument to a boolean for every condition function.

- The value recursion – `While` describes a while-loop, which has a condition function, an updating function, and an initialization expression. The first two arguments are of type `PreOpenFun` and will be compiled in the same way as `Foreign` given the initialization expression as the input.
- Constant values can be determined during compile-time. `Const` is directly integrated into the expression, while `PrimConst` needs to be calculated manually per scalar type.

Table 1. *Mapping all Accelerate primitive operations to GLSL functions (based on Soest [50, Table 1]). Unless stated explicitly, GLSL functions support all integer types (`Int8`, `Int16`, `Int32`, `Int64`, and their unsigned versions), or all floating-point types (`Float16`, `Float32`, `Float64`), or both, depending on original operations.*

Operation	**Accelerate Constructor**	**GLSL Function**

addition, multiplication;
subtraction, negation;
absolute, sign
	`PrimAdd`, `PrimMul`;
	`PrimSub`, `PrimNeg`;
	`PrimAbs`, `PrimSig`
		`+`, `*`;
		`-`, `-`;
		`abs`, `sign`

integer division truncated to 0; reminder; simultaneous `quot` and `rem`	`PrimQuot`; `PrimRem`; `PrimQuotRem`	implemented manually supporting all integers

integer division;
modulo;
simultaneous `div` and `mod`
	`PrimIDiv`;
	`PrimMod`;
	`PrimDivMod`
		`/`;
		`%`;
		`\x,y→(x/y, x%y)`

bitwise and, or; xor, not	`PrimBAnd`, `PrimBOr`; `PrimBXor`, `PrimBNot`	`&`, `|`; `^`, `~`
bitwise shift left, shift right; rotate left, rotate right	`PrimBShiftL`, `PrimBShiftR`; `PrimBRotateL`, `PrimBRotateR`	`<<`, `>>`; implemented manually supporting all integers

Table 1 Continued. *Mapping all primitive operations to GLSL functions.*

Operation	**Accelerate Constructor**	**GLSL Function**

bitwise count ones;
count leading zeros;
count trailing zeros
	`PrimPopCount`;
	`PrimCountLeadingZeros`;
	`PrimCountTrailingZeros`
		`binCount`;
		bit-length − `findMSB`;
		`findLSB`;
		their 64-bit versions are implemented manually

floating-point division; reciprocal fraction	`PrimFDiv`; `PrimRecip`	`/`; `\x→1/x`; double-precision is not supported
sine, cosine, tangent	`PrimSin`, `PrimCos`, `PrimTan`	`sin`, `cos`, `tan`; double-precision is not supported
arc sine, cosine, tangent, 2-argument tangent	`PrimAsin`, `PrimAcos`, `PrimAtan`, `PrimAtan2`	`asin`, `acos`, `atan`, `atan`; double-precision is not supported
hyperbolic sine, cosine, tangent	`PrimSinh`, `PrimCosh`, `PrimTanh`	`sinh`, `cosh`, `tanh`; double-precision is not supported
arc-hyperbolic sine, cosine, tangent	`PrimASinh`, `PrimACosh`, `PrimAtanh`	`asinh`, `acosh`, `atanh`; double-precision is not supported

exponent, logarithm;
square-root, power;
logarithm-base
	`PrimExpFloating`, `PrimLog`;
	`PrimSqrt`, `PrimFPow`;
	`PrimLogBase`
		`exp`, `log`;
		`sqrt`, `pow`;
		`\x,y→log(y)/log(x)`;
		double-precision is not supported

Table 1 Continued. *Mapping all primitive operations to GLSL functions.*

Operation	**Accelerate Constructor**	**GLSL Function**

round;
floor;
ceil;
truncate to 0
	`PrimRound`;
	`PrimFloor`;
	`PrimCeil`;
	`PrimTruncate`
		`round`;
		`floor`;
		`ceil`;
		`trunc`

isNaN, isInfinite	`PrimIsNan`, `PrimIsInfinite`	`isnan`, `isinf`
$<$, $>$; $\leq$, $\geq$; $=$, $\neq$	`PrimLt`, `PrimGt`; `PrimLtEq`, `PrimGtEq`; `PrimEq`, `PrimNEq`	`<`, `>`; `<=`, `>=`; `==`, `!=`
maximum, minimum	`PrimMax`, `PrimMin`	`max`, `min`
logical and, or, not	`PrimLAnd`, `PrimLOr`, `PrimLNot`	`&&`, `||`, `!`
Haskell's fromIntegral, and toFloat	`PrimFromIntegral`, `PrimToFloating`	type conversions

- Primitive scalar operations (of type `PrimFun`, partially shown in Listing 8) are represented in the first order so that they can be directly transformed into GLSL code. As shown in Table 1, though most operators have a direct matching function, and the rest can be implemented manually, some mathematical functions, such as exponent, power, square root, and trigonometric operators, don't have any support for double-precision floating-point numbers, nor do there exist magic extensions allowing to extend such functions. Those functions that are called and implemented manually are stored in `FuncMap`. In general, we map almost all primitive operations to GLSL functions except for those that don't support double-precision floating-points.

- Array instruction – `ArrayInstr` contains two types of instructions (introduced in Listing 25), where they all represent the operation of fetching elements from an external independent environment, namely the buffer environment (`benv`) that stores all buffers and scalars and shows up during the execution. The first one is the indexing instruction (`Index`), which means that given a linearized index, it should return an element from the desired buffer, i.e. array indexing. To achieve this, we first look into the `AInstrEnv`, checking whether the called buffer is there or not. If it is missing, we can simply add it there with a unique name. The next step is to append the indexing operator to the end of the buffer name. The second one is the parameter instruction (`Parameter`), which directly returns the desired scalar value from the external buffer environment.

- `ShapeSize` marks the calculation of the length for a given shape representation and expressions to get dimensional numbers. The way to implement it is quite straightforward. We just need to multiply all returned values from expressions together.

- Unlike other operations, `Undef` marks an undefined value (thrown by an undefined behavior) rather than a kind of operation. The purpose of its existence is to enable the program to continue when encountering undefined behaviors, and it is supposed to be overwritten by other values. Thus, it can be any value as long as it won't cause a runtime error.

- For bits reinterpretations, GLSL has direct support on all the type casting for such operations. In our implementation, there are no more than a large number of pattern-matching cases. Note that the conversion can only be operated on two types with the same bit length.

Compiling combination expressions The combination expression is only used in permutations. It tells the backend how to update each position of a buffer given an old value and a new one. However, this kind of update is not thread-safe, meaning that multiple threads may update the same piece of memory at the same time, causing data races. The general way to solve this, is either using atomic operations or using a spinlock to do the thread synchronization.

In the Vulkan backend, we implemented three ways to compile combination expressions, and they are presented below. In theory, the first solution provides the best performance, while the last one has the worst efficiency.

1. We try to compile expressions using matching atomic operations. GLSL natively supports `atomicAdd`, `atomicAnd`, `atomicOr`, `atomicXor`, `atomicMin`, and `atomicMax` for 32-bit integers and unsigned integers. To allow 64-bit integers, single- and double-precision floating-point numbers, extra extensions are required (mentioned in §4.2.1). That limits us to only compile addition, logical operations, and min-max expressions.

2. Combination expressions are compiled to lock-free loops using `atomicCompSwap` if regular atomic operations do not support a certain

type or a complex expression. `atomicCompSwap` compares a specific element in the buffer to a compared value, and swaps the element with a candidate value, then returns the original element regardless of the outcome of the comparison. It avoids data races. The compiling steps are: first, we declare new variables storing original values from both the default array and source array, and then we declare a compared variable containing the original value from the default array. The next step is to compile the expression in the same way as that for `PreOpenExp`. Finally, we call `atomicCompSwap` to update the array and catch its returned value. If the return matches the compared value, the update succeeds; otherwise, we should keep looping from the first step. Considering the following example:

```
let xs = fromList (Z:.100) [100..] :: Array (Z :. Int) Int32
permute (*) (fill (I1 100) 1) (\(I1 i) -> Just_ (I1 i)) (use xs)
```

The combination expression in the above example is multiplication, where there is no direct matching atomic operation. It is compiled into a lock-free loop following the aforementioned procedure:

```
 1  int32_t e2_CASin;
 2  int32_t e3_CASin;
 3  int32_t e4_CAScomp;
 4  do {
 5      e2_CASin = permBuff2[uint32_t(globalID)];
 6      e3_CASin = mutBuff3[uint32_t(e1_mutIdx)];
 7      e4_CAScomp = e3_CASin;
 8      e3_CASin = atomicCompSwap(mutBuff3[uint32_t(e1_mutIdx)],
 9                                e4_CAScomp,
10                                (e2_CASin) * (e3_CASin));
11  } while (e3_CASin != e4_CAScomp);
```

In the above code snippet, `mutBuff3` is the default array, and `permBuff2` is the `xs`. From line 1 to line 7, temporary variables are declared and bound. From line 8 to line 10, `atomicCompSwap` are called to update `mutBuff3` with the combination expression as its third argument. The result of it is bound to `e3_CASin` for later checking the update.

Since single- and double-precision floating-point numbers have the same bit lengths as `Int32` and `Int64`, inspired by Hamuraru [28], they can be cast to those two types while preserving the bit pattern to utilize the same lock-free loop. Buffer inlets for single- and double-precision arrays are declared in `Int32` and `Int64`, and then we involve bits reinterpretations in the compilation steps of ordinary lock-free loops. Here is an example:

```
let xs = fromList (Z:.100) [100..] :: Array (Z :. Int) Float
permute (*) (fill (I1 100) 1) (\(I1 i) -> Just_ (I1 i)) (use xs)
```

Both the source and default arrays are of type `Float`, which is not supported by `atomicCompSwap`, but with the type casting, we can compile it into:

```
 1 float32_t e2_CASin;
 2 float32_t e3_CASin;
 3 float32_t e4_CAScomp;
 4 do {
 5     e2_CASin = permBuff2[uint32_t(globalID)];
 6     e3_CASin = uintBitsToFloat(mutBuff3[uint32_t(e1_mutIdx)]);
 7     e4_CAScomp = e3_CASin;
 8     e3_CASin = uintBitsToFloat(
 9                    atomicCompSwap(
10                        mutBuff3[uint32_t(e1_mutIdx)],
11                        floatBitsToUint(e4_CAScomp),
12                        floatBitsToUint((e2_CASin) * (e3_CASin))));
13 } while (e3_CASin != e4_CAScomp);
```

In the above code snippet, `mutBuff3` is of type `uint32_t`, but the combination expression is supposed to operate on a different type. Thus, on the sixth, eighth, eleventh, and twelfth lines, bits reinterpretations are called to ensure the correctness of both the type and the value.

3. If the bit-length of a type is not accepted by `atomicCompSwap`, the very last option is to compile the expression into a loop with a spinlock. Unlike the lock-free loop, the spinlock requires an additional buffer to document the lock state for each position of the default array. As such, the buffer needs to be initialized as unlocked on the top level of the shader. Then, within the loop, each thread would check the state of the target position and keep waiting until it's idle. After acquiring permission to access that position, the buffer will be locked and updated by the current thread. Finally, after finishing operations, the thread would unlock the buffer. Considering the same example from the last compiling option, `xs` is containing `Int16` now. The program is compiled into:

```
 1 bool keepWaiting = true;
 2 while (keepWaiting) {
 3     if (atomicCompSwap(
 4             lock[uint32_t((0) * mutDim1 + permIdx0)], 0, 1)  == 0) {
 5         int16_t e2_mutBuff = mutBuff3[uint32_t(e1_mutIdx)];
 6         mutBuff3[uint32_t(e1_mutIdx)] = (e0_permBuff) * (e2_mutBuff);
 7         memoryBarrier();
 8         keepWaiting = false;
 9         atomicExchange(lock[uint32_t((0) * mutDim1 + permIdx0)], 0);
10     }
11 }
```

In the above code example, `lock` is of the type 32-bit integer. The fourth line linearizes indices of buffer `lock`. Within the if-statement, the sixth line updates the protected buffer directly. Then, on the ninth line, `atomicExchange` is called to unlock the current position.

For the third way, there is a function – `glslInitSpinLock` in module `Data.Array.Accelerate.Vulkan.Kernel` to generate spinlock initialization code given the lock length and the initial value.

4.2.2 Vulkan context

After generating a complete GLSL code from the Partitioned IR, executable Vulkan objects should be built from it for the next step. Recalling the Vulkan programming model introduced in §2.3.2, all Vulkan objects are classified into the global context and the local context according to their visibility across all Vulkan kernels. The former is shared and visible for all kernels, while each kernel has its own local Vulkan context.

During compile-time, both the global context and all local contexts will be initialized. There is only one global context exists per application, and it will be shared across multiple local contexts. As shown in Listing 50, it is represented as a record notation containing a Vulkan instance, possible debugger, physical device, logical device, memory allocator, components related to descriptors, items about commands, execution queue, and a fence. In general, since all Vulkan pipelines and shaders are created with the help of the global context, it must be initialized on the top level of the kernel compilation. In addition, one can enable the debugging functionality by uncomment lines with the keyword "debug" or "validation" in the function `createInstance` of the module `Vulkan.Runtime` in our implementation.

```haskell
data GlobalVulkanResources = GlobalVkResources
    { inst                :: Instance
    , debugExt            :: Maybe DebugUtilsMessengerEXT
    , phys                :: PhysicalDevice
    , dev                 :: Device
    , allocator           :: Allocator
    , descriptorPool      :: DescriptorPool
    , descriptorSetLayout :: DescriptorSetLayout
    , descriptorSet       :: DescriptorSet
    , pipelineLayout      :: PipelineLayout
    , cmdPool             :: CommandPool
    , cmdBuffer           :: CommandBuffer
    , queue               :: Queue
    , fence               :: Fence
    }
```

Listing 50. *Global Vulkan context.*

Local context, on the other hand, is dedicated per kernel. It is built given a complete GLSL code and the global context. As presented in Listing 51, `LocalVulkanResources` only contains compute pipelines and shaders, where pipelines are the actual parts enclosing the shader that will be executed. Moreover, pipelines and shaders consume hardware resources, which need to be freed after execution.

Specifically, `LocalVulkanResources` has the field `initSpinLock`, which may contain a pipeline to initialize the spinlock buffer on the device and a tuple of metadata. This field is special, which is only used when permuting arrays in some cases. Under the situation, the same position in the mutable buffer may be updated by multiple threads at the same time during permutation; this field tells whether we need a spinlock and the tuple contains the length and initial value that such a lock should have. Recalling the Vulkan kernel (as shown in Listing 34), the first element of the tuple is a list of scalars, where the actual number should be the multiplication of them for a similar reason of `kernelThreads`. And the second one is the initial value. If the lock is not used, this field would remain `Nothing`.

```haskell
data LocalVulkanResources env = LocalVkResources
    { pipeline     :: Pipeline
    , shader       :: ShaderModule
    , initSpinLock :: Maybe (Pipeline, ShaderModule, ([Idx env Int], Int32))
    }
```

Listing 51. *Local Vulkan context.*

Though both global and local contexts are external objects, they are registered to the garbage collection (GC) with finalizers. As such, no extra work is required to manage such objects, and the GC can prevent memory leaks. Note that GC is not guaranteed to be triggered before exiting the program, so Vulkan may complain about not cleaning up resources.

Initializations Initializing global and local contexts is trivial and follows the same steps according to §2.3.2. We have made function `createGlobalResources` in module `Data.Array.Accelerate.Vulkan.Vulkan.Runtime` to create a `GlobalVulkanResources` given the maximum buffer number that a pipeline can take. This number is important since all pipelines are built based on the same `pipelineLayout`, and Vulkan resources are pre-allocated according to it. In our implementation, we just guess a large enough number for it. As shown in Listing 52, using `unsafePerformIO`, we can initialize the global context at any point with `createGlobalResources`. Note that `NOINLINE` should be attached to this stateful operation to avoid multiple Vulkan object creations.

```haskell
{-# NOINLINE vkGlobalResources #-}
vkGlobalResources :: Lifetime GlobalVulkanResources
vkGlobalResources = unsafePerformIO $ do
    vkGlobal <- createGlobalResources 50
    lt <- newLifetime vkGlobal
```

```
  -- Add finalizer.
  addFinalizer lt $ do
    destroyGlobalResources vkGlobal
  pure lt
```

Listing 52. *Making global Vulkan contexts.*

To create `LocalVulkanResources`, we have made function `compileGLSL` and `createLocalResources` to compile the GLSL code and initialize the context respectively in the same module. The third argument of the function `createLocalResources` is a maybe type for the GLSL code, which is used to build the spinlock initializing pipeline.

Additionally, since the local thread number is determined dynamically, when creating shaders this information should be given using the `SpecializationMapEntry` from the Vulkan-Haskell binding, which is similar to a buffer inlet.

Finalizers To register Vulkan contexts to the GC, each of them should have a finalizer, which deletes the context and deallocates the memory. `Lifetime` in Accelerate is an exposed interface for accessing the registered object. After creating a Vulkan context, the function `newLifetime` should be called to register and generate a `Lifetime` for it. Then, a finalizing function should be attached to it by calling `addFinalizer`. After accessing the `Lifetime`, the function `touchLifetime` should be called to keep it alive until here. When a `Lifetime` object has not been touched anymore, GC would step in and clean it up at a suitable time. However, it is possible that GC may never be called during the whole life cycle of a program, which Vulkan would complain about. Currently, there is no mechanism to automatically force triggering GC before the program exits, so the user can either manually `performGC` [24, §System.Mem], or ignore the segmentation fault and other Vulkan-related errors.

As an example, Listing 52 draws details of making a global context and attaching a finalizer. A finalizing function (`destroyGlobalResources`) for the global context consists of many freeing or destroying operations for objects nested in the global data type. The order of executing such operations is the opposite of the order of creation. Similarly, we have made function `destroyLocalResources` for local contexts. Note that local ones must be purged before the global one.

4.2.3 Making kernels

We have made the `VulkanKernel` instance for class `IsKernel` with the implementation of the function `compileKernel` to compile clustered operations into Vulkan kernels.

Listing 53 depicts restoring singleton operations from clustered operations, where fusion is disabled. From line 2 to line 6, a custom operation data type is defined to bypass the type checker and reflect Vulkan-specific operations. The function `compileKernel` runs two checks first: the operation enclosure check and the bit-length check. The 18th line represents the former check; it peeks inside the clustered operations checking the existence of Vulkan-specific operations. From line 19 to line 25, the byte sizes of `Int` and `Word` are checked to ensure the default length is 8 bytes. This is important since data transfer and GLSL code generation take 8 bytes as default for `Int` and `Word`; bit-length mismatch may result in strange values. If both checks pass, a singleton operation will be restored by `applySingletonCluster` and passed to `compileOperation` (defined in Listing 54) with operation arguments.

```haskell
 1 -- Create a custom data type to bypass the type checker.
 2 data WhichOp = GenerateOp | PermuteOp
 3 -- Map the operation.
 4 peekOp :: forall args. VulkanOp args -> WhichOp
 5 peekOp VkGenerate = GenerateOp
 6 peekOp VkPermute = PermuteOp
 7
 8 instance IsKernel VulkanKernel where
 9     type KernelOperation VulkanKernel = VulkanOp
10     type KernelMetadata VulkanKernel = NoKernelMetadata
11
12     compileKernel :: forall env args. Env AccessGroundR env
13                   -> Data.Array.Accelerate.AST.Partitioned.Clustered
14                         (KernelOperation VulkanKernel) args
15                   -> Args env args
16                   -> VulkanKernel env
17     compileKernel _ (Clustered clst bclst) clstArgs
18         | Just _ <- peekSingletonCluster peekOp clst
19         = case (bytesElt (TupRsingle (SingleScalarType
20                                      (NumSingleType
21                                      (IntegralNumType TypeInt)))),
22                 bytesElt (TupRsingle (SingleScalarType
23                                      (NumSingleType
24                                      (IntegralNumType TypeWord))))) of
25             (8, 8) -> applySingletonCluster (compileOperation uid)
26                                             clst
27                                             clstArgs
28             _ -> error "Only supports 64-bit integers as default"
29         where uid = hashOperation (Clustered clst bclst) clstArgs
```

Listing 53. *Compiling kernels.*

Generation When pattern matching over `VkGenerate` for the function `compileOperation`, its typed arguments are given. The goal for this case is to create a `VulkanKernel` from the input arguments. As shown in Listing 54, `body` is of type `PreOpenExp` and `outBuf` is constructed by `ArgArray` of data type `Arg` (as shown in Listing 26). Steps for making a `VulkanKernel` include:

```
compileOperation :: forall env a. UID
                 -> VulkanOp a
                 -> Args env a
                 -> VulkanKernel env
compileOperation uid VkGenerate args@( ArgFun (Lam lhs (Body body))
                                   :>: outBuf@(ArgArray m
                                                        (ArrayR sh t)
                                                        gv
                                                        gvb)
                                   :>: ArgsNil)
```

Listing 54. *Compiling generations.*

1. Processing the global index. The output array might be a multi-dimensional one, so as mentioned in §4.2.1, the global index needs to be unfolded and multi-dimensional indices are taken to build a compiling environment for `body` with `lhs`.

2. Compiling the scalar expression. `body` is of type `PreOpenExp`. With the newly built environment, it can be compiled by the procedure as stated in §4.2.1 giving a GLSL statement and a tuple of returned expressions.

3. Compiling called buffers. During the compilation, external buffers may be called. These buffers with the buffer argument `gvb` are compiled into GLSL buffer inlets according to §4.2.1. Then, returned expressions are bound to buffer arguments.

4. Creating a `LocalVulkanResources`. Resembling all the parts we compiled before, we have a complete GLSL code now. In the same way as Listing 52, local Vulkan contexts are created (depicted in Listing 55), whose input is the complete GLSL code.

```
spirvCode = unsafePerformIO $ compileGLSL glslCode
vkObjects = unsafePerformIO $ do
    vkLocal <- withLifetime vkGlobalResources (\vkGlobal ->
    createLocalResources vkGlobal spirvCode Nothing)
    lt <- newLifetime vkLocal
    addFinalizer lt $ do
        withLifetime vkGlobalResources (‘destroyLocalResources‘
    vkLocal)
    pure lt
```

Listing 55. *Making local Vulkan contexts.*

After finishing the above steps, now we have everything to fill the `VulkanKernel` record notation. `kernelID` is given as UID. The kernel shape only contains the shape of `outBuf`, which is `gv`. The kernel input arguments are called buffers of array indexing. The mutable argument is empty. As such, the spinlock field of the local Vulkan context is also empty. The output argument is just `gvb`. The kernel resource is the built local Vulkan context `vkObjects`. The thread number is the same as the kernel shape, since each position of `outBuf` should only be handled by one thread.

Permutation Permutation can be seen as the extension of generation since they share parts of the compilation. As illustrated in Listing 56, `VkPermute` has more complex arguments, where `combLhs1` binds an element from the source buffer `aIn` and `combLhs2` binds a value from the default buffer `aMut`. `combBody` and `permBody` are the combination expression and index permutation expression respectively. `aMut` and `aIn` stand for the default (mutable) array and source (input) array. We can compile this operation by:

```
compileOperation uid VkPermute args@( ArgFun fun@(Lam combLhs1
                                                   (Lam combLhs2
                                                        (Body combBody)))
                                      :>: aMut@(ArgArray m
                                                   (ArrayR sh t)
                                                   gv
                                                   gvb)
                                      :>: ArgFun (Lam permLhs (Body permBody))
                                      :>: aIn@(ArgArray m'
                                                  (ArrayR sh' t')
                                                  gv'
                                                  gvb')
                                      :>: ArgsNil)
```

Listing 56. *Compiling permutations.*

1. Compiling `permBody` in the same way as compiling a regular expression of generation.

2. Adding a check for indices. The returned expression of the last step is a tuple with type `(Int,((),sh))`, where its first element marks whether to exit or continue permuting the default array. If the integer is zero, the current thread should exit and do nothing; otherwise, continue to the next step with `sh` that represents the updating target position of the default array.

3. Compiling the combination expression. The compilation environment is made by pushing the source element and default element (indexing into the default array using `sh`) to an empty environment. As described in §4.2.1, the `combBody` is compiled by trying three different ways with the built environment.

Finally, we can make a `VulkanKernel` for the current operation. Again, the kernel ID is given as UID. The kernel shapes contain that for both `aMut` and `aIn`, which are `gv` and `gv'` in this case. The input arguments are indexed arrays and `aIn` (`gvb'`). The mutable argument is just the `aMut` (`gvb`). The output argument is empty. Similar to the generation, given a complete GLSL code we can make the local Vulkan context from it. Because all the positions in the source array are supposed to be accessed once, the thread number for permutation is the length of the source array, i.e. the `gv'`.

4.3 Execution

The execution stage presented in Figure 3 involves data inlet and transfer, environment creation, and scheduled operation execution. The last one contains array-level operations of type `SeqSchedule` (introduced in §3.11). The executable Vulkan kernels inside it will be executed by the function `compute` from the Vulkan runtime.

We have made `VulkanElement` data type to store a pair of ordinary Accelerate's buffers or scalars and their remote Vulkan buffers. So that ordinary buffers can be reused to reduce the overhead of allocating new buffers and freeing old buffers. Though ordinary Accelerate's buffers are related to remote Vulkan buffers, there is no data transfer in between unless the `SeqSchedule` operation performs a computation on the host side, or requires returns.

```haskell
data VulkanElement a where
  Scalar' :: ScalarType a
          -> a
          -> Lifetime ((Vk.Buffer, Allocation, AllocationInfo), Ptr ())
          -> VulkanElement a
  Buffer' :: ScalarType a
          -> (Buffer a)
          -> Lifetime ((Vk.Buffer, Allocation, AllocationInfo), Ptr ())
          -> VulkanElement (Buffer a)

type VulkanElements = TupR VulkanElement
type VulkanEnv = Env VulkanElement
```

Listing 57. *Vulkan elements for execution.*

Listing 57 depicts two data constructors `Scalar'` and `Buffer'` to contain scalars and buffers respectively. Their third field is a tuple containing a Vulkan buffer object managed by GC. The pointer inside the tuple exposed an interface to access the remote memory. `VulkanElements` wraps multiple `VulkanElement` into a tuple. `VulkanEnv` builds the execution environment for `SeqSchedule` operations.

4.3.1 Vulkan runtime

Vulkan runtime provides support for Vulkan operations including memory management, data transfer, and Vulkan kernel execution.

Buffers and memory We created a function `createBufferInfo` to simplify the buffer allocation and allocate multiple buffers once. Listing 58 shows that in the 17th line, the function `VMA.createBuffer` is called with the byte size, buffer usages, allocation-create flags, memory usages, and memory properties to allocate a new buffer. This process may fail due to insufficient memory space. Thus, in the 28th line, we try to perform GC to clean unreachable Vulkan buffers and then try to allocate the buffer again, if it fails. Apart from that, the 15th line checks whether the current byte size is zero and skips the creation if that's the case.

```
 1 createBufferInfo :: Allocator
 2                  -> [Int]
 3                  -> BufferUsageFlags
 4                  -> AllocationCreateFlags
 5                  -> MemoryUsage
 6                  -> MemoryPropertyFlags
 7                  -> IO [(Buffer, Allocation, AllocationInfo)]
 8 createBufferInfo _ [] _ _ _ _ = pure []
 9 createBufferInfo allocator
10                  (x:xs)
11                  buffUsage
12                  alloCreateFlag
13                  memUsage
14                  memProp = do
15     (buffer, allocation, allocationInfo) <- if x >= 1 then do
16         catch @IO @VulkanException (
17             VMA.createBuffer allocator
18                 zero { size  = fromIntegral x
19                      , usage = buffUsage
20                      }
21                 zero { flags = alloCreateFlag
22                      , usage = memUsage
23                      , requiredFlags = memProp
24                      }
25             )
26             $ \_ -> do
27                 say "Failed to create Vulkan buffer, trying to perform GC"
28                 performGC
29                 VMA.createBuffer allocator
30                     zero { size  = fromIntegral x
31                          , usage = buffUsage
32                          }
33                     zero { flags = alloCreateFlag
34                          , usage = memUsage
35                          , requiredFlags = memProp
36                          }
37     else do
38         say "Warning: buffer size less than 1, skipping this creation"
39         pure (zero, zero, zero)
40     rest <- createBufferInfo allocator
41                             xs
```

```
                              buffUsage
                              alloCreateFlag
                              memUsage
                              memProp
    return $ (buffer, allocation, allocationInfo) : rest
```

Listing 58. *Vulkan buffer allocations*

One of the most confusing parts of Vulkan is the flags for buffer creation. To have proper buffers, we need to set buffer usage flags [55, §12.1. Buffers], allocation create flags, memory usages, memory properties [4, §Memory allocation]. Buffer usage flags mark the buffer storage type and its copying permissions; allocation create flags specify the access pattern; memory usages are flags to set whether the buffer is accessed frequently by the host or the device; memory properties set the memory location and its visibility.

Data transfer Data transfer in Vulkan is quite flexible. Halladay [26, 27] studied five ways to transfer data. In general, there is no best practice to achieve this; each combination of hardware and software has different optimal flags and ways to perform data transfer.

Since all the Vulkan buffers in our implementation are allocated in the device memory, the most efficient way to transfer the data is to keep them mapped to the host and directly access them. They will be unmapped only before deconstructions. Copying data between Vulkan buffers and Accelerate buffers relays on the pointer, and the array peeking and poking function from the module `Foreign.Marshal.Array` [24]. Though we only present the implementation of transferring buffers in Listing 59, scalars are treated in the same way since they can be seen as size-one buffers.

```
newtype Ptr' a = Ptr' (Ptr (ScalarArrayDataR a))

copyBufferToPtr :: ScalarType a
                -> Buffer a
                -> Ptr' a
                -> IO ()
copyBufferToPtr (SingleScalarType (st :: SingleType a'))
                (Buffer size ua)
                (Ptr' destPtr)
    | SingleDict  <- singleDict st
    , SingleArrayDict <- singleArrayDict st
    = do
        withUniqueArrayPtr ua $
            \srcPtr -> copyArray (destPtr :: Ptr a) srcPtr size

copyPtrToBuffer :: ScalarType a
                -> Buffer a
                -> Ptr' a
                -> IO ()
copyPtrToBuffer (SingleScalarType (st :: SingleType a'))
                (Buffer size ua)
                (Ptr' srcPtr)
    | SingleDict      <- singleDict st
    , SingleArrayDict <- singleArrayDict st
    = do
      withUniqueArrayPtr ua $
          \destPtr -> copyArray destPtr (srcPtr :: Ptr a) size
```

Listing 59. *Data transfer.*

In the above code snippet, the first step for both transfer directions is to get a pointer to the Accelerate buffer by using `withUniqueArrayPtr`. Then `copyArray` shows up to copy data between pointers to Accelerate buffers and Vulkan buffers. After pushing data to the remote memory and before copying data back to the host, `flushAllocation` and `invalidateAllocation` from the Vulkan-Haskell binding are called separately to clean up the cache and make the change take effect.

Conducting computations To run each local Vulkan context, we have made the function `compute`, which is defined as:

```
compute :: GlobalVulkanResources
        -> LocalVulkanResources
        -> VulkanEnv env'
        -> Env (Idx env') env
        -> [Exists (VulkanArg env)]
        -> [Exists (VulkanArg env)]
        -> [Exists (VulkanArg env)]
        -> Int
        -> IO ()
```

Listing 60. *Kernel execution.*

In the above code snippet, `compute` takes both the global and local objects as the input. It is given two environments: the first one contains all Vulkan elements (i.e., buffers and scalars), some of which may not be used by the kernel; the second one is which the kernel parameterized over during compilation, where it contains de Bruijn indices for querying concrete Vulkan elements from the first environment. Instead of creating a sub-environment of the first one, we can save some memory space and overhead by having a separate indices environment. Then the three lists of `VulkanArg` contain input, mutable, and output Vulkan arguments. As shown in Listing 35, the `GroundVar` inside the `VulkanArg` data type constructs a de Bruijn index, which can be used to query a new index from the second environment for fetching a Vulkan element from the first environment. In the last place of the function `compute`, the thread number is given to execute the kernel. Because data interaction happens in the `IO` state rather than returning explicitly, the return of `compute` is nothing.

The main steps for executing a Vulkan context include:

1. Projecting de Bruijn indices twice to fetch Vulkan elements. As mentioned above, the first projection maps the `GrundVar` of Vulkan arguments to the second input environment (of type `Env (Idx env') env`) of `compute` to have new indices. Then the projection happens again to query concrete Vulkan elements (Vulkan buffers and scalars) from the first given environment (of type `VulkanEnv env'`) using new de Bruijn indices.

2. Creating temporary Vulkan buffers and copying mutable buffers to them. For permutations, mutable buffers are the updated target but they might also be indexed in scalar expressions. Elements from array indexing should remain the same during the execution per kernel. Thus, when executing permutation kernels, array updates only happen on temporary buffers, while mutable buffers remain the same. After execution, new values from temporary buffers are copied back to mutable buffers.

3. Binding all the buffers to the descriptor set. Since shaders access buffers via the descriptor set, one important step is to bind buffers to the descriptor set. Note that each buffer has a unique binding number that corresponds with the buffer inlet of generated GLSL code during compilation.

4. Pushing commands to the command buffer. We need to start the command buffer first to make it in the recording state, then push commands (including the possible spinlock initialization, mutable buffer copying, and kernel dispatch) to it and end the recording.

5. Submitting the command buffer to the compute queue and waiting for the fence. After the submission, the device would start execution immediately. After that, we need to wait until the fence is reached, which marks either the finish of the execution or errors.

6. Cleaning up. At this point, both the fence and command buffer should be reset to prepare for the next run. Temporary buffers need to be freed.

Some utilities of the runtime are inspired by expipiplus1's compute example [22].

4.3.2 Scheduling and execution

When running the Accelerate program with `run @Vulkan ...`, Kernel IR will be automatically scheduled sequentially. Each operation will be transformed into constructors of `SeqSchedule` data type. Inspired by Soest [50], we have made similar structures to execute sequential-scheduled Vulkan kernels.

The execution module starts from the instance for `Execute` class, which describes the step to execute `SequentialSchedule` that contains `VulkanKernel` (introduced in Listing 34).

```
instance Execute SequentialSchedule VulkanKernel where
  executeAfunSchedule :: GFunctionR t
                      -> SequentialSchedule VulkanKernel
                                            ()
                                            (Scheduled SequentialSchedule t)
                      -> IOFun (Scheduled SequentialSchedule t)
  executeAfunSchedule gf ss = executeVulkanElementsIOFun gf $
                                  executeSequentialSchedule Empty ss
```

Listing 61. *Implementation of instance `Execute SequentialSchedule VulkanKernel`.*

As shown in Listing 61, two functions are chained to execute `SequentialSchedule`. The first one – `executeVulkanElementsIOFun` interacts with Accelerate's input and output, which convert Accelerate buffers to and from Vulkan buffers. Receiving an ordinary buffer or a scalar value, `executeVulkanElementsIOFun` makes a `VulkanElement` data wrapped by a `Lifetime` and triggers the data transfer. Then, this data is given to `executeSequentialSchedule` for executing scheduled operations. After that, returned buffers and scalars from those operations are pushed to a `MVar` stack [24, §Control.Concurrent.MVar], where results from the computation are awaiting to be taken by other Accelerate components.

The second function of Listing 61 is `executeSequentialSchedule`, which takes Vulkan buffers to recursively build an execution environment and calls `executeSeqSchedule` (defined in Listing 62) with the typed environment to conduct the computing, i.e. `SeqSchedule`. Usually, the execution of `SeqSchedule` returns a tuple of `VulkanElement` data, so the second function is also responsible for triggering data copying from the remote memory to local buffers and wrapping them in a tuple handled by the first function.

To run `SeqSchedule` operations, we have made the following function:

```
executeSeqSchedule :: VulkanEnv env
                   -> SeqSchedule VulkanKernel env t
                   ->  IOFun (VulkanElements t)
```

Listing 62. *Executing `SeqSchedule`.*

It pattern-matches over constructors of `SeqSchedule`:

- **Exec**: To execute a Vulkan kernel, `Exec` not only contains the kernel but also carries indices to build a sub-environment. Instead of slicing the original Vulkan environment, we choose to make a specific environment for storing these indices. Then we called the function `executeKernel` with the whole Vulkan environment and the indices environment (their usage is described in §4.3.1).

```haskell
executeKernel :: VulkanEnv env
              -> Env (Idx env) env'
              -> VulkanKernel env'
              -> IOFun (VulkanElements ())
executeKernel env env' kernel = do
    let st = SingleScalarType (NumSingleType
                              (IntegralNumType
                              TypeInt))
    let sh' = map (\idx ->
                  Exists (VulkanArg (Var (GroundRscalar st) idx)))
                  (kernelShapes kernel)
    let threads' = product $
        map (\idx ->
            let (Scalar' _ s _)
                = prj' (prj' idx env') env in s)
            (kernelThreads kernel)
    withLifetime vkGlobalResources
        (\vkGlobal ->
            withLifetime
                (kernelResources kernel)
                (\vkLocalObj' ->
                    compute vkGlobal
                            vkLocalObj'
                            env
                            env'
                            (sh' ++ kernelInArgs kernel)
                            (kernelMutArgs kernel)
                            (kernelOutArgs kernel)
                            threads'))
    touchVulkanArgs env env' $ sh'
                            ++ kernelInArgs kernel
                            ++ kernelMutArgs kernel
                            ++ kernelOutArgs kernel
    touchLifetime (kernelResources kernel)
    touchLifetime vkGlobalResources
    return TupRunit
```

Listing 63. *Executing kernels.*

As shown in Listing 63, `executeKernel` first wraps the shape indices as Vulkan scalars to have `sh'`. Then the thread number is fetched by multiplying all the scalars together. After that, `compute` from the Vulkan runtime is called and it keeps updated values in the buffer. Finally, Vulkan buffers and contexts are touched to keep them alive.

- **Return**: This constructor asks us to build and return a tuple from the environment, given ground variables. Usually, `Return` is used to pick desired buffers from the environment and return special pointers to them wrapped in a tuple. We can simply use `mapTupR` with index projection to build such a tuple. After that, returned pointers might be used to copy data back to the host by the function that calls `executeSeqSchedule`, i.e. `executeSequentialSchedule`.

- **Compute**: Since not all computations can be executed on the Vulkan device, this constructor marks an expression that is supposed to be calculated on the host side. To achieve this, the function `evalExp` from the Accelerate interpreter is called. Its first argument is the expression, and this second argument is a function that handles array instructions. As such, we have made the following function:

```
evalArrayInstr :: VulkanEnv env
               -> ArrayInstr env (s -> t)
               -> s
               -> IO t
evalArrayInstr env (Index (Var _ idx)) i
    = case prj' idx env of
        Scalar' {} -> error "evalArrayInstr: Index impossible"
        b@(Buffer' st _ _) -> do
            b' <- returnVulkanElement b
            pure $ indexBuffer st b' i
evalArrayInstr env (Parameter (Var _ idx)) _
    = case prj' idx env of
        Scalar' _ e _ -> pure e
        Buffer' {} -> error "evalArrayInstr: Parameter impossible"
```

Listing 64. *Executing array instructions.*

`evalArrayInstr` pattern-matches over `ArrayInstr`'s constructors. For array indexing, it first queries the buffer and copies the data back to the ordinary buffer. Then, it accesses that buffer with a built-in Accelerate function. For parameters, since all scalars within Accelerate are immutable, the value can be simply returned.

- **Alet**: Local bindings for arrays are similar to those for scalar expressions. As presented in Listing 65, the first step is to execute the bound scheduled operations. Next, push returned values into the Vulkan environment recursively according to the left-hand-side. Then, execute the body operation with the new environment.

```
executeSeqSchedule env (Alet lhs _ bnd body) = do
    newVars <- executeSeqSchedule env bnd
    let env' = updateEnv env lhs newVars
    executeSeqSchedule env' body
```

Listing 65. *Array local bindings.*

- **Alloc**: Allocating a new buffer requires the scalar type and the buffer length. Although the scalar type is given explicitly, the length should be computed from the shape representation and expression variables. As such, we make `getSize` to calculate the buffer size. As listed in Listing

66, it recursively projects indices to the environment for dimensional numbers and then multiplies them together to have the buffer length measured in element count.

```haskell
getSize :: VulkanEnv env -> ShapeR sh -> ExpVars env sh -> Int
getSize _ ShapeRz _ = 1
getSize env (ShapeRsnoc shr) (TupRpair sh (TupRsingle (Var _ idx)))
  = if n <= 0 then 0
    else getSize env shr sh * n
    where Scalar' _ n _ = prj' idx env
getSize _ _ _ = error "getSize: Impossible"
```

Listing 66. *Computing array sizes.*

- **Use**: Lifting an ordinary buffer to the Vulkan buffer consists of two steps: first, creating a Vulkan buffer using `createBufferInfo`; second, calling `copyBufferToPtr` to copy data from the buffer to the Vulkan buffer.

- **Unit**: `Unit` aims to lift a scalar value to the Vulkan buffer, which is a special case of `Use` with only one element. Thus, we make an ordinary buffer first to capture such a scalar. Then, the buffer is lifted to Vulkan in the same way.

- **Acond**: As shown in Listing 67, array-level flow controls execute the scheduled operation of either the true branch or false branch. This depends on the results from executing the condition, which is of type `PrimBool` but can be converted to an ordinary boolean value by `toBool`.

```haskell
executeSeqSchedule env (Acond (Var _ idx) tExp fExp)
    = if toBool cond then
            executeSeqSchedule env tExp
      else executeSeqSchedule env fExp
    where (Scalar' _ cond _) = prj' idx env
```

Listing 67. *Array flow-controls*

- **Awhile**: Array-level while-loops keep executing the body operation until the condition check fails. We have made such a function to execute the loop:

```haskell
executeAwhile :: VulkanEnv env
              -> SeqScheduleFun VulkanKernel env (t -> PrimBool)
              -> SeqScheduleFun VulkanKernel env (t -> t)
              -> VulkanElements t
              -> IOFun (VulkanElements t)
executeAwhile env
              cond@(Slam condLhs (Sbody condBody))
              body@(Slam expLhs (Sbody expBody))
              xs
= do
```

```
    -- Check condition.
    (TupRsingle condVar) <- executeSeqSchedule (updateEnv env
                                                          condLhs
                                                          xs)
                                               condBody
    condVar' <- returnVulkanElement condVar
    -- If true, continue the loop.
    if toBool condVar' then do
        xs' <- executeSeqSchedule (updateEnv env expLhs xs) expBody
        executeAwhile env cond body xs'
    -- If false, return values.
    else return xs
executeAwhile _ _ _ _ = error "executeAwhile: Impossible"
```

Listing 68. *Array while-loops*

`executeAwhile` takes Vulkan elements as the initial values. Then it binds them to the environment and executes the body to get the condition. After that, the condition is used to judge whether breaking the loop and returning values, or updating the value and looping along the body expression.

5 Evaluation and discussion

To answer the third research question and explore the limitation of the Vulkan backend, we pick the Multigrid Method, Naive N-body, and FlashAttention from the CFAL-bench [15] as benchmarks. Though CFAL has many other testing items, only these three support both the old and new pipeline of Accelerate.

5.1 Experiments

Throughout all the tests, we use the same testing environment:

- **CPU**: 2x Intel Xeon Platinum 8260L 24C48T @2.40GHz
- **GPU**: NVIDIA GeForce RTX 3090 24GB
- **RAM**: 12x 32GB DDR4 @2933MHz
- **Operating System**: Ubuntu 22.04
- **GPU Driver**: 550.90
- **Vulkan**: 1.3.277

- **CUDA**: 11.8
- **LLVM**: 15.0
- **GHC**: 9.2.5
- **GCC**: 12.3

The CPU backend (modified based on commit 165a9e7 of Accelerate-LLVM [7]) and the Vulkan backend were tested under the Accelerate new pipeline (modified based on commit 55e0dfd of Accelerate [6]), while the PTX backend was benchmarked in the old pipeline. Both the CPU backend and the PTX backend are evaluated with and without fusion. Fusion is enabled by default for the CPU and PTX backend, so we have to disable it manually. For PTX, disabling fusion is simple. One way to do it, is by adding the following lines to the cabal project file:

```
package accelerate
flags: +debug
```

and running the executable program with extra flags `+ACC -fno-fusion -ACC`. For the CPU backend, it is slightly difficult as the new pipeline is still in development. However, setting `mkGraph _ _ _ = mempty` and `finalize = finalizeNoFusion`, making

```
labelLabelledArg vars l (L x@(ArgArray In  (ArrayR shr _) _ _) y)
    = LOp x y (vars M.! InDir  l, rank shr)
labelLabelledArg vars l (L x@(ArgArray Out (ArrayR shr _) _ _) y)
    = LOp x y (vars M.! OutDir l, rank shr)
labelLabelledArg _ _ (L x y)
    = LOp x y (0,0)
```

in the `instance MakesILP NativeOp` in the file `Native/Operation.hs` of Acclerate-LLVM should do the work. For the Vulkan backend, fusion is not implemented.

Since the compilation of the primitive scalar expression does not support double-precision floating-point numbers for all the operations, we have changed all these numbers for three benchmarks into single-precision floating-point numbers.

5.1.1 Naive N-body simulations

N-body simulations model the interactions between a large number of particles, such as the influence of gravity on stars or molecules in fluids. The naive implementation refers to the simplest implementation of N-body simulations, where the gravitational force is the only force that affects particles.

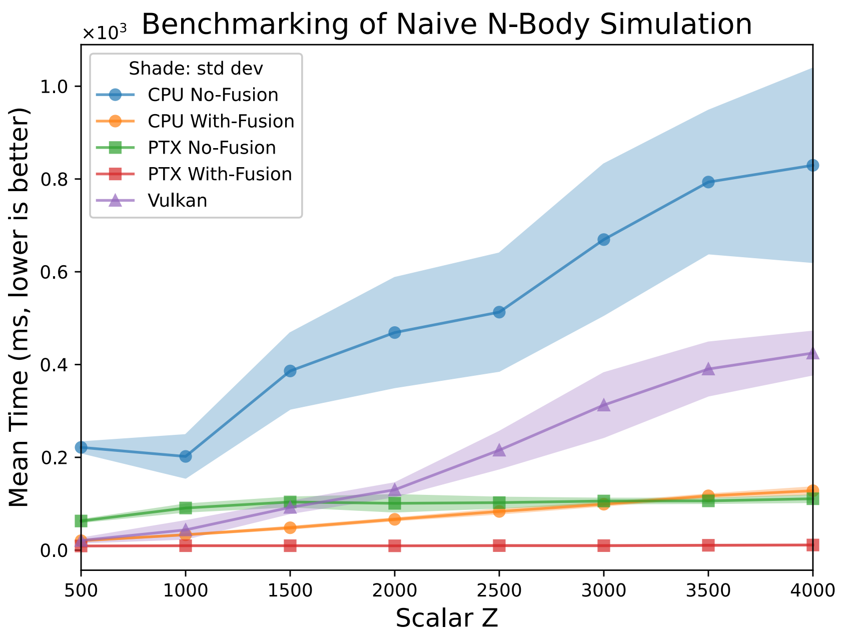

Figure 4. *Results of N-body simulations.*

This benchmark tests the raw computational power and efficiency of handling a large number of floating-point operations.

Figure 4 and Table 3 (in the Appendix) illustrate the testing results of N-body simulations. The Vulkan backend keeps its advantage in smaller scalars, while after 1500, its performance is worse than the PTX backend. This means it cannot maintain a consistently lower mean execution time compared to the PTX backend and also the CPU backend with fusion. Though its execution time increases rapidly after 1500, its performance remains competitive and consistently better than the CPU backend without fusion. As for the standard deviation, PTX backend and CPU backend with fusion have a quite narrow execution time distribution. Though the standard deviation of execution time increases along with the input size for the Vulkan backend, it does not change a lot compared to the CPU backend without fusion.

5.1.2 Multigrid methods

The multigrid method is used to solve partial differential equations (PDEs) on multiple levels of grid resolutions. This benchmark tests the memory hierarchy performance including local cache efficiency and access patterns across different level of the memory hierarchy.

As shown in Figure 5 and Table 4 (in the Appendix), the Vulkan backend has the lowest mean time and standard deviation in most cases compared

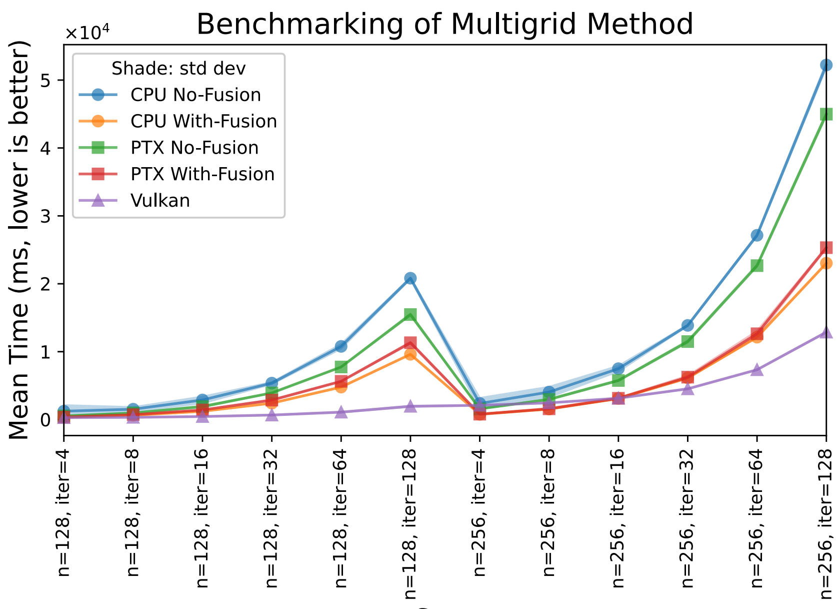

Figure 5. *Results of Multigrid Method.*

to all other backends. Moreover, its performance is significantly better for larger input sizes and more iteration steps. The CPU backend without fusion is still the slowest and most unstable one in all testing cases.

5.1.3 FlashAttention

FlashAttention is an optimized implementation of the attention mechanism, which is used in transformer models of artificial intelligence. This benchmark tests how well a backend can handle the computation and memory access patterns.

The testing results are shown in Figure 6 and Table 5 (in the Appendix), `n` and `d` are the sizes of the attention layer. The tile size `m` for all these three backends is 16384, except that the Vulkan backend on the specification `(n=2048, d=64)` and `(n=2048, d=128)` uses the tile size 8192. In general, we ran tests on different input sizes, which are the combination of `n` and `d`, with different tile sizes ranging from 8 to 16384, then chose the best result among all tile sizes for each input size on individual backends.

From the figure and table, we can tell that the Vulkan backend shows competitive performance, outperforming the CPU and PTX backends without fusion. They perform similarly in all cases for the mean execution time, but the standard deviation of the CPU backend with no fusion is still the worst among all. For larger input sizes, Vulkan's performance is closer to

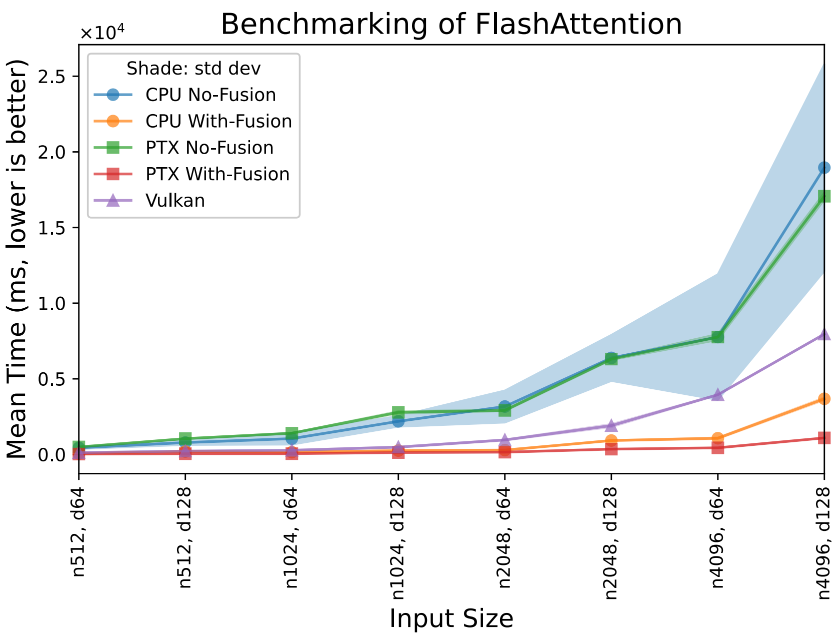

Figure 6. *Results of FlashAttention. Test results for Vulkan backend on input size* `(n=2048, d=64)` *and* `(n=2048, d=128)` *use the tile size* `m=8192`*; others use* `m=16384`*.*

CPU with fusion but still slightly lags behind.

5.1.4 Discussion

From the experiment results, we can observe that in most cases, the Vulkan backend provides very competitive performance even compared to the PTX backend. Specifically, in the multigrid method testing, our implementation outperforms all other backends in most cases. However, in the naive N-body simulation, the execution time of Vulkan increases linearly, while the PTX backend holds a constant time and remains lower than that of Vulkan. Due to the lack of comparative studies, we are not aware of the factor that makes the Vulkan backend significantly outperform the PTX backend in the multigrid method testing. For example, whether this strength comes from the low overhead of launching and executing Vulkan shaders, or it is caused by aggressive fast-math optimizations [16, §Analysis] (discussed in §5.2).

Moreover, we have discovered that the impact of fusion is great enough to eliminate the advantage brought by massive parallel hardware, i.e., the GPUs. Regardless of backends and benchmarks, implementations benefit significantly from the fusion that reduces the execution time by minimizing the overhead of multiple kernel launches and improving data locality.

What's interesting is that the execution time of the CPU backend seems to vary greatly on the same benchmark and the same specification, i.e., the standard deviation is pretty high, while this situation does not happen for the PTX backend. This may be due to the CPU frequency being adjusted in a wide range more dynamically when executing a large number of small unfused kernels compared to the GPU, resulting in the computing speed varying as well.

5.2 Limitations

Though the Vulkan backend performs competitively with the PTX backend, some limitations in both the Vulkan and our implementation cannot be overlooked. There are also some disadvantages to using Vulkan for GPU computing instead of rendering, which are hard to overcome and compensate for.

Extension supports The greatest limitation comes from the uneven extension supports across all hardware vendors. Vulkan takes Spir-V code as its low-level representation to build shaders. However, it only supports 32-bit integers and 32-bit single-precision floating-point numbers by default. Extra data types are introduced via enabling extensions that are not widely supported by vendors. This could increase the complexity of the code generation for a Vulkan backend since unsupported types and operations are required to be handled differently.

Moreover, some operations are extended by enabling certain extensions, while their support is various on different platforms. For example, `EXT_shader_atomic_float` introduces support for single- and double-precision floating-point numbers operated on atomic addition – `atomicAdd`, while their support on atomic min and max operations – `atomicMin` and `atomicMax` is enabled by a separate extension – `EXT_shader_atomic_float2`. However, no NVIDIA GPU supports the latter currently, while most recent AMD GPUs support both. Thus, to increase the compatibility across various platforms, the latter extension is not enabled in our implementation. When running such operations on AMD GPUs, their implementations are the same as that of NVIDIA GPUs, i.e. using a lock-free loop or a spinlock, which limits the performance. Or we can generate GLSL code to enable such extensions and utilize those native operations, which could fully exploit AMD GPUs but would increase the complexity of development.

Limited data types on instructions Currently, there is no such extension that provides support for operating complex mathematical functions on double-precision floating-point numbers, including exponent, power, trigonometric, and logarithm. This is because most GPUs don't have specific double-precision instruction units to compute such functions. Though some GPUs, e.g. recent NVIDIA and AMD GPUs, have the capability to conduct complex double-precision floating-point mathematical operations, to maintain the cross-platform feature and due to non-technical reasons, native supports or extra extensions are not included in Vulkan. Manually implementing such functions for not-supported hardware would require a considerable amount of work, and the performance may not be good.

In comparison, all modern x86 architecture CPUs have full instruction units to support complex mathematical functions, which also happen to be more transparent internally than GPUs. Thus, many scientific libraries for CPUs can achieve such tasks. For NVIDIA GPUs that support CUDA, there is no gap for supporting single- and double-precision types on those operations [18, §6. Double Precision Mathematical Functions], though it could be slow when dealing with double-precision numbers.

Instruction precisions Interestingly, when coding the shader for Vulkan in GLSL, though common mathematical operators support single-precision decimals, their executing results may not be as accurate as those of CUDA, where PTX is the low-level representation of the CUDA language.

Table 2. *Unit in the last place (ULP) errors of single-precision floating-point instructions [17, §13. Mathematical Functions],[55, §Precision of Individual Operations]. ULP error ranges of CUDA in the table are only for NVIDIA GPUs realised after 2014, and in Vulkan, GLSL is of version 4.50.*

Instruction	**CUDA**	**Vulkan**
addition	0	0
subtraction	0	0
multiplication	0	0
division	≤2	≤2.5
exponent	≤2	≥3
logarithm	≤1	≤3
power	≤4	≥3
square-root	≤1	≤4.5
arc-tangent	≤2	≤4096

Table 2 shows the instruction errors measured in units in the last place (ULPs). The smaller it is, the more accurate the result is. Allowing larger ULP errors means more aggressive fast-math optimizations can be employed to execute instructions faster. We can find that although both CUDA and Vulkan can provide error-free results from addition, subtraction, and multiplication, in other cases, Vulkan tolerates greater errors. Nevertheless, CUDA has a flag to disable all fast-math optimizations providing more accurate results that are nearly error-free, while Vulkan does not have that option at all. For CPUs, modern mainstream x86 ones can execute those instructions with errors less than 1 ULP when disabling fast math, and unlike GPUs' proprietary ecosystems, there are many open-source compilers built for x86 CPUs providing the option to disable fast math.

Programs built with CUDA language are compiled into PTX code and even lower-level representations. Though Table 2 only shows the situation when coding in CUDA instead of PTX, the idea is the same since CUDA code will be transformed to PTX.

Moreover, double-precision floating numbers are only required to be as accurate as single-precision numbers [55, §Precision and Operation of SPIR-V Instructions], which means some Vulkan drivers simply downcast the precision to compute and convert the result back. This does not ensure the double-precision worsening the case for using Vulkan to conduct high-precision scientific computations.

The implementation of Vulkan backend There are a few spots in the current implementation that can be improved in future work. The first one is the Vulkan buffer management. All involved data is pushed to the Vulkan device memory during runtime, yet with the increasing computation scale it is possible to exceed the maximum device memory when keeping all the data there. Though they are managed by GC and will be freed if they are not used anymore, the smarter way is to use the least recently used (LRU) caching mechanism [47], i.e. only keep the newest data in the remote memory and transfer the old data to the device when it is called.

Secondly, the generated GLSL code from our implementation lacks boundary checks in the array-indexing operation. Since there is no built-in way to throw and catch an error during the execution of Vulkan shaders, we discard the boundary check for array access, resulting in possible undefined behaviors. However, extra buffers can be bound to shaders to catch runtime errors, and custom runtime utilities can receive such buffers and handle those errors.

Thirdly, Accelerate's default integer type is 64-bit, but operating on this type for Vulkan devices requires extra computing resources and enabling additional extensions since the default integer type for them is 32-bit. Though users can explicitly define arrays of `Int32` and convert `Int64` to

`Int32` for speeding up computations, operations related to indices are always operated under `Int64` since it is the only indices type of Accelerate. Implicit conversions or modifications of Accelerate indices can relax this issue, but more work and complex analysis are required.

6 Conclussion

This thesis has explored the development of a Vulkan backend for Accelerate, with the goal of expanding its GPU acceleration to more platforms. In this project, we have successfully implemented an OpenGL Shading Language (GLSL) code generator, a backend runtime, and other modules. As an alternative to CUDA, Vulkan does provide suitable APIs to execute host-side and device-side operations, including memory management, data transfer, and kernel execution.

The experimental evaluations show that compared to the PTX backend, our Vulkan backend is a competitive player in most cases and significantly outperforms the no-fusion CPU backend, proving its effectiveness and potential. All these results reveal a unique option for executing array computations using a cross-platform and high-performance library.

Despite the promising results, the limitations of the Vulkan backend are also significant. Due to the uneven support for external GLSL extensions on various platforms, achieving more portability for our implementation could be difficult. Apart from that, certain data types are not supported by advanced mathematical operations. Even though they are supported by other functions, the computing results are not as accurate as that of the PTX backend. Future work could focus on the refinement and optimization of the backend to ensure broader compatibility and portability.

As mentioned in §2.3.2, developing a Vulkan program is extremely verbose since Vulkan exposes various low-level details and has a large number of configurable flags. To have great efficiency, developers need to do more work in optimizations. In addition, there is no best practice for building a Vulkan program. Various hardware architectures and Vulkan flags influence the performance differently. In other words, the best practice is to use a profiler to optimize the program per device.

"Every pit holds the promise of a mighty tree", so does the Vulkan. Though it is still young and immature, we just have to let it grow.

References

[1] AccelerateHS. *Accelerate: High-performance parallel arrays for Haskell.* `https://github.com/AccelerateHS/accelerate`. 2024. (Visited on 08/05/2024).

[2] AccelerateHS. *LLVM backends for the Accelerate array language.* `https://github.com/AccelerateHS/accelerate-llvm`. 2024. (Visited on 08/05/2024).

[3] achirkin. *vulkan-api: Low-level low-overhead vulkan api bindings.* `https://github.com/achirkin/vulkan`. 2021. (Visited on 08/05/2024).

[4] AMD. *Vulkan Memory Allocator (Version 3.1.0).* `https://gpuopen-librariesandsdks.github.io/VulkanMemoryAllocator/html/`. (Visited on 07/07/2024).

[5] Mike Bailey. "Introduction to the Vulkan graphics api". In: *SIGGRAPH Asia 2018 Courses.* 2018, pp. 1–225.

[6] David van Balen. *Accelerate: High-performance parallel arrays for Haskell.* `https://github.com/dpvanbalen/accelerate`. 2024. (Visited on 08/05/2024).

[7] David van Balen. *LLVM backends for the Accelerate array language.* `https://github.com/dpvanbalen/accelerate-llvm`. 2024. (Visited on 08/05/2024).

[8] Stefan Berghofer and Christian Urban. "A head-to-head comparison of de Bruijn indices and names". In: *Electronic Notes in Theoretical Computer Science* 174.5 (2007), pp. 53–67.

[9] Lars Bergstrom and John Reppy. "Nested data-parallelism on the GPU". In: *Proceedings of the 17th ACM SIGPLAN international conference on Functional programming.* 2012, pp. 247–258.

[10] Guy E Blelloch. *NESL: a nested data parallel language.* Carnegie Mellon University, 1992.

[11] Guy E Blelloch. *Nested Data-Parallelism and NESL.* `https://www.cs.cmu.edu/~scandal/cacm/node4.html`. (Visited on 05/19/2024).

[12] Guy E Blelloch and Gary W Sabot. "Compiling collection-oriented languages onto massively parallel computers". In: *Journal of parallel and distributed computing* 8.2 (1990), pp. 119–134.

[13] Manuel M T Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. "Accelerating Haskell array codes with multicore GPUs". In: *DAMP '11: The 6th workshop on Declarative Aspects of Multicore Programming.* ACM, 2011.

[14] Gang Chen, Guobo Li, Songwen Pei, and Baifeng Wu. "High performance computing via a GPU". In: *2009 First International Conference on Information Science and Engineering.* IEEE. 2009, pp. 238–241.

[15] Department of Computer Science at the University of Copenhagen. *CFAL-bench.* `https://github.com/diku-dk/CFAL-bench`. 2024. (Visited on 07/30/2024).

[16] University of Copenhagen. *Comparing the performance of OpenCL, CUDA, and HIP.* `https://futhark-lang.org/blog/2024-07-17-opencl-cuda-hip.html`. 2024. (Visited on 08/08/2024).

[17] NVIDIA Corporation. *CUDA C++ Programming Guide (Release 12.5).* `https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf`. 2024. (Visited on 07/30/2024).

[18] NVIDIA Corporation. *CUDA Math API Reference Manual (Release 12.6).* `https://docs.nvidia.com/cuda/pdf/CUDA_Math_API.pdf`. 2024. (Visited on 08/05/2024).

[19] Nicholas J Curtis, Kyle E Niemeyer, and Chih-Jen Sung. "Using SIMD and SIMT vectorization to evaluate sparse chemical kinetic Jacobian matrices and thermochemical source terms". In: *Combustion and Flame* 198 (2018), pp. 186–204.

[20] Olivier Danvy. "Folding left and right over Peano numbers". In: *Journal of Functional Programming* 29 (2019), e6.

[21] Prashanta Kumar Das and Ganesh Chandra Deka. "History and evolution of GPU architecture". In: *Emerging Research Surrounding Power Consumption and Performance Issues in Utility Computing.* IGI Global, 2016, pp. 109–135.

[22] expipiplus1. *vulkan: Bindings to the Vulkan graphics API.* `https://github.com/expipiplus1/vulkan`. 2024. (Visited on 08/02/2024).

[23] Massimiliano Fatica. "CUDA toolkit and libraries". In: *2008 IEEE Hot Chips 20 Symposium (HCS).* 2008, pp. 1–22. DOI: `10.1109/HOTCHIPS.2008.7476520`.

[24] Andy Gill and Oregon Graduate Institute of Science and Technology. *base-4.20.0.1: Core data structures and operations.* `https://hackage.haskell.org/package/base-4.20.0.1`. 2024. (Visited on 08/05/2024).

[25] Peter N. Glaskowsky. *NVIDIA's Fermi: the first complete GPU computing architecture.* `https://www.nvidia.com/content/pdf/fermi_white_papers/p.glaskowsky_nvidia's_fermi-the_first_complete_gpu_architecture.pdf`. 2009. (Visited on 07/01/2024).

[26] Kyle Halladay. *Comparing Uniform Data Transfer Methods in Vulkan.* `https://kylehalladay.com/blog/tutorial/vulkan/2017/08/13/Vulkan-Uniform-Buffers.html`. 2017. (Visited on 07/30/2024).

[27] Kyle Halladay. *Improving Vulkan Breakout.* `https://kylehalladay.com/blog/tutorial/vulkan/2017/08/30/Vulkan-Uniform-Buffers-pt2.html`. 2017. (Visited on 07/30/2024).

[28] ANCA HAMURARU. *Atomic operations for floats in OpenCL – improved.* `https://streamhpc.com/blog/2016-02-09/atomic-operations-for-floats-in-opencl-improved/`. 2016. (Visited on 08/05/2024).

[29] Charles R Harris, K Jarrod Millman, Stéfan J Van Der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J Smith, et al. "Array programming with NumPy". In: *Nature* 585.7825 (2020), pp. 357–362.

[30] W Daniel Hillis and Guy L Steele Jr. "Data parallel algorithms". In: *Communications of the ACM* 29.12 (1986), pp. 1170–1183.

[31] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. "A history of Haskell: being lazy with class". In: *Proceedings of the third ACM SIGPLAN conference on History of programming languages.* 2007, pp. 12–1.

[32] Intel. *Intel Intrinsics Guide (Version 3.6.9).* `https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html`. 2024. (Visited on 08/05/2024).

[33] Intel. *SPIR-V: Default Interface to Intel Graphics Compiler for OpenCL Workloads.* `https://www.intel.com/content/dam/develop/external/us/en/documents/spirv-for-publishing.pdf`. (Visited on 07/04/2024).

[34] Gabriele Keller, Manuel MT Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, and Ben Lippmeier. "Regular, shape-polymorphic, parallel arrays in Haskell". In: *ACM Sigplan Notices* 45.9 (2010), pp. 261–272.

[35] Gabriele Cornelia Keller. “Transformation-based implementation of nested data parallelism for distributed memory machines”. PhD thesis. Technische Universität Berlin, 1999.

[36] Chris Lattner and Vikram Adve. “LLVM: A compilation framework for lifelong program analysis & transformation”. In: *International symposium on code generation and optimization, 2004. CGO 2004.* IEEE. 2004, pp. 75–86.

[37] LLVM. *User Guide for NVPTX Back-end.* `https://llvm.org/docs/NVPTXUsage.html`. (Visited on 03/20/2024).

[38] Xinliang Lu. *accelerate-vulkan.* `https://github.com/largeword/accelerate-vulkan`. 2024. (Visited on 08/05/2024).

[39] Simon Marlow, Simon Peyton Jones, and Wolfgang Thaller. “Extending the Haskell foreign function interface with concurrency”. In: *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell.* 2004, pp. 22–32.

[40] Ricardo Marroquim and André Maximo. “Introduction to GPU Programming with GLSL”. In: *2009 Tutorials of the XXII Brazilian Symposium on Computer Graphics and Image Processing.* IEEE. 2009, pp. 3–16.

[41] Trevor L McDonell, Manuel MT Chakravarty, Vinod Grover, and Ryan R Newton. “Type-safe runtime code generation: accelerate to LLVM”. In: *ACM SIGPLAN Notices* 50.12 (2015), pp. 201–212.

[42] Trevor L McDonell, Manuel MT Chakravarty, Gabriele Keller, and Ben Lippmeier. “Optimising purely functional GPU programs”. In: *ACM SIGPLAN Notices* 48.9 (2013), pp. 49–60.

[43] NVIDIA. *NVIDIA GPU microarchitecture.* `https://www.ece.lsu.edu/koppel/gp/notes/set-nv-org.pdf`. 2023. (Visited on 05/19/2024).

[44] NVIDIA Corporation. *PTX ISA (Release 8.5).* `https://docs.nvidia.com/cuda/pdf/ptx_isa_8.5.pdf`. 2024. (Visited on 07/04/2024).

[45] Michael Oneppo. “HLSL shader model 4.0”. In: *ACM SIGGRAPH 2007 courses.* 2007, pp. 112–152.

[46] John D Owens, Mike Houston, David Luebke, Simon Green, John E Stone, and James C Phillips. “GPU computing”. In: *Proceedings of the IEEE* 96.5 (2008), pp. 879–899.

[47] Junseok Park, Hyunkyong Choi, Hyokyung Bahn, and Kern Koh. “Buffer Caching Algorithms for Storage Class RAMs”. In: *International Journal of Computers* 3.1 (2009), pp. 41–52.

[48] Jon Peddie. *The History of the GPU-Eras and Environment*. Springer Nature, 2023.

[49] Amanda K Sharp. *Memory Layout Transformations*. `https://www.intel.com/content/www/us/en/developer/articles/technical/memory-layout-transformations.html`. 2019. (Visited on 07/30/2024).

[50] Lars van Soest. *Compiling Second-Order Accelerate Programs to First-Order TensorFlow Graphs*. `https://studenttheses.uu.nl/handle/20.500.12932/44636`. 2023. (Visited on 03/20/2024).

[51] Stephan Soller. *GPGPU origins and GPU hardware architecture*. `https://nbn-resolving.org/urn:nbn:de:bsz:900-opus4-45000`. 2011. (Visited on 03/20/2024).

[52] Robert Strzodka. "Abstraction for AoS and SoA layout in C++". In: *GPU computing gems Jade edition*. Elsevier, 2012, pp. 429–441.

[53] The Accelerate Team. *accelerate-1.3.0.0: An embedded language for accelerated array processing*. `https://hackage.haskell.org/package/accelerate-1.3.0.0/docs/Data-Array-Accelerate.html`. 2020. (Visited on 08/05/2024).

[54] The Khronos Vulkan Working Group. *SPIR-V Specification*. `https://registry.khronos.org//SPIR-V/specs/unified1/SPIRV.pdf`. 2024. (Visited on 07/04/2024).

[55] The Khronos Vulkan Working Group. *Vulkan 1.3 - A Specification (with all registered extensions)*. `https://registry.khronos.org/vulkan/specs/1.3-extensions/html/vkspec.html`. 2024. (Visited on 06/24/2024).

[56] Jeyarajan Thiyagalingam, Olav Beckmann, and Paul HJ Kelly. "An exhaustive evaluation of row-major, column-major and Morton layouts for large two-dimensional arrays". In: *Performance Engineering: 19th Annual UK Performance Engineering Workshop*. University of Warwick Coventry, UK. 2003, pp. 340–351.

[57] Richard Vuduc and Jee Choi. "A brief history and introduction to GPGPU". In: *Modern Accelerator Technologies for Geographic Information Science* (2013), pp. 9–23.

Appendix

Table 3. *Mean time of each backend tested on N-body simulations with standard deviation in the parenthesis, lower is better.*

	Mean Time and **Standard Deviation** (ms)				
Scale Z	CPU		PTX		Vulkan
	With fusion	No fusion	With fusion	No fusion	No fusion
500	20.35 (1.091)	221.5 (11.09)	8.956 (0.7241)	62.79 (2.342)	20.86 (4.418)
1000	33.13 (1.024)	202.0 (46.07)	9.549 (0.9729)	90.80 (7.916)	43.60 (19.12)
1500	48.24 (2.329)	386.2 (81.47)	9.544 (0.9057)	103.8 (9.819)	92.17 (12.43)
2000	66.36 (2.358)	469.0 (117.9)	9.336 (0.5844)	100.9 (17.95)	130.1 (14.26)
2500	83.38 (4.741)	513.0 (126.6)	9.775 (1.288)	102.4 (10.97)	215.5 (39.57)
3000	99.00 (3.328)	669.4 (162.3)	9.671 (1.295)	105.9 (5.258)	312.7 (68.93)
3500	117.1 (3.286)	793.4 (153.9)	10.44 (1.252)	106.2 (4.738)	390.4 (57.28)
4000	128.1 (7.566)	829.5 (208.6)	11.05 (1.016)	110.8 (7.335)	424.8 (46.38)

Table 4. *Mean time of each backend tested on Multigrid Method with standard deviation in the parenthesis, lower is better.*

	Mean Time and **Standard Deviation** (ms)				
Spec	CPU		PTX		Vulkan
	With fusion	No fusion	With fusion	No fusion	No fusion
n=128, iter=4	384.1 (119.3)	1234 (891.9)	358.8 (15.73)	529.9 (29.45)	283.7 (15.07)
n=128, iter=8	635.5 (43.45)	1527 (312.8)	726.7 (22.84)	976.0 (12.52)	330.0 (6.531)
n=128, iter=16	1218 (31.64)	2889 (493.5)	1419 (31.90)	1918 (113.7)	451.0 (18.58)
n=128, iter=32	2418 (28.49)	5367 (151.0)	2870 (74.90)	3926 (24.41)	669.0 (29.44)
n=128, iter=64	4785 (37.31)	10780 (342.5)	5635 (31.20)	7745 (11.90)	1095 (53.57)
n=128, iter=128	9613 (22.87)	20820 (118.3)	11320 (132.0)	15470 (143.5)	1956 (90.63)
n=256, iter=4	772.2 (10.57)	2375 (811.9)	798.0 (48.12)	1577 (23.70)	2107 (20.38)
n=256, iter=8	1551 (7.988)	4055 (769.8)	1590 (96.92)	2968 (73.80)	2460 (10.74)
n=256, iter=16	3113 (112.7)	7497 (409.5)	3138 (86.11)	5758 (82.22)	3147 (27.12)
n=256, iter=32	6144 (89.98)	13870 (81.82)	6248 (219.3)	11490 (154.8)	4530 (31.41)
n=256, iter=64	12140 (106.7)	27140 (142.0)	12640 (415.4)	22670 (147.0)	7341 (22.28)
n=256, iter=128	23030 (19.15)	52220 (380.0)	25320 (121.2)	44980 (265.5)	12880 (52.27)

Table 5. *Mean time of each backend tested on FlashAttention with standard deviation in the parenthesis, lower is better. Test results for Vulkan backend on input size* `(n=2048, d=64)` *and* `(n=2048, d=128)` *use the tile size* `m=8192`*; others use* `m=16384`*.*

Input Size	**Mean Time** and **Standard Deviation** (ms)				
	CPU		PTX		Vulkan
	With fusion	No fusion	With fusion	No fusion	No fusion
n=512, d=64	69.42 (2.743)	439.4 (74.10)	27.06 (3.200)	496.7 (12.23)	123.6 (14.32)
n=512, d=128	194.1 (15.27)	788.5 (157.9)	52.31 (3.420)	1048 (21.97)	225.2 (7.242)
n=1024, d=64	154.3 (7.474)	1049 (393.0)	53.08 (1.885)	1407 (26.07)	277.5 (30.28)
n=1024, d=128	257.1 (12.34)	2191 (352.6)	127.8 (2.462)	2787 (77.66)	488.0 (33.68)
n=2048, d=64	290.6 (20.30)	3170 (1059)	154.2 (18.92)	2915 (55.63)	960.8 (32.96)
n=2048, d=128	922.7 (21.82)	6383 (1522)	353.7 (7.687)	6307 (63.66)	1909 (102.7)
n=4096, d=64	1075 (48.56)	7762 (4154)	440.4 (16.10)	7762 (202.0)	3950 (21.21)
n=4096, d=128	3681 (94.66)	18960 (6839)	1095 (12.90)	17070 (247.6)	7959 (26.38)