UNIVERSITY OF COPENHAGEN
FACULTY OF SCIENCE

MSc Thesis

A WebGPU backend for Futhark

Sebastian Paarmann

Supervisor: Troels Henriksen
Department of Computer Science

May 31, 2024

Abstract

In this thesis project, we create a new backend for the Futhark compiler. Futhark is a functional data-parallel array programming language whose optimizing compiler can generate efficient GPGPU code. Our backend targets the WebGPU API, enabling Futhark programs to be run in web browsers while still taking advantage of GPU compute capability.

As part of the backend, we implement code generation of shaders in the WebGPU Shading Language (WGSL) from the Futhark compiler's internal kernel representation as well as an appropriate implementation of the host-side runtime system for the WebGPU API. Additionally, we provide tooling to use Futhark's built-in testing and benchmarking tools with our backend by interacting with a browser to run the programs under test.

WebGPU and WGSL have many restrictions not present in the APIs used by Futhark's existing GPU backends. We devise and implement workarounds for many but not all of them. As a result, our backend can successfully run some Futhark programs, but some other valid programs are unsupported. We also investigate the remaining limitations and discuss potential future solutions.

Contents

1 Introduction

Futhark [11, 12] is a functional array programming language designed to handle compute-intensive tasks as part of a larger program. Its optimizing compiler can generate code using a variety of backends, such as multicore C code or OpenCL. Of particular interest are the GPU backends — OpenCL, CUDA, and HIP — which make it possible to take advantage of modern massively parallel GPU hardware while writing high-level functional Futhark programs instead of low-level C. OpenCL, CUDA, and HIP are different APIs that allow using the GPU. They are focused on general-purpose computation, as opposed to APIs like DirectX or Vulkan that have a focus on using GPUs for their original graphics rendering purpose. All of Futhark's existing GPU backends generate native code, designed to run directly on user's operating systems.

WebGPU [16] is a new API being developed for the web platform to allow web sites to make use of GPU capabilities in a portable manner. It can be used for graphics rendering, but unlike its predecessor WebGL, it also explicitly supports general-purpose computation.

In this project, we implemented a new backend for the Futhark compiler targeting WebGPU that allows embedding compiled Futhark programs in web pages. Running efficiently in browsers opens up new use cases for Futhark outside of native programs. There is an existing WebAssembly backend [14] that allows running Futhark programs in a browser, but it only uses the CPU, similar to Futhark's C backend. Our new backend uses WebGPU to make use of the compute power of GPUs, similar to the OpenCL and CUDA backends.

Another possible benefit of a WebGPU backend is increased portability, even for native programs. WebGPU is designed to be portable and run on a wide variety of platforms and underlying hardware, and it is possible to use it in native applications in addition to the web. For example, recent macOS versions have deprecated OpenCL in favour of Apple's own Metal API, which Futhark does not currently support. Native WebGPU implementations support running WebGPU applications on top of Metal however. In this project we have focused on the web use case, leaving potential native WebGPU targets to future work.

In chapter 2 we give an introduction to the relevant background about general-purpose GPU programming in general, as well as Futhark and its compiler. Chapter 3 gives a more detailed description of the relevant aspects of the WebGPU API and associated WGSL programming language. With the important background established, we describe the implementation of our backend in chapter 4. Chapter 5 contains a description of how we implemented support for automatic testing and benchmarking of the backend. Finally, we discuss current limitations of our backend, evaluate its usability as well as the suitability of WebGPU as target, and describe a demonstration web page we built in chapter 6.

2 Background

2.1 GPU Programming Model

We will start by giving an introduction to general-purpose GPU compute in general. The Futhark compiler's existing GPU backend is originally designed around the capabilities of OpenCL [9], so we will focus on that. It also supports targeting CUDA [6] and has adopted some of CUDA's terminology, so we will also include it. OpenCL and CUDA provide very similar basic functionality and the fundamentals are core to how modern GPUs work. Most of it is also applicable to WebGPU's compute support, though we will later see that there are some differences. (Unlike CUDA and OpenCL, WebGPU also supports using GPUs' graphics rendering pipelines, which do not exactly match this model, but we will focus only on general purpose computing here.)

When using GPUs for compute, we refer to the normal CPU as the *host*, while the GPU is often referred to as the *device*. The host controls the overall execution, interacting with the rest of the system and setting up data for the GPU to use. In the general case, the host and device have separate memory spaces not directly accessible to the other side. The host can allocate memory on the device as well as initiate transfers in either direction using functions provided by the compute API.

Many operations take the form of asynchronous operations submitted by the host to some form of queue, to then be executed by the device. An explicit host-device synchronization point can be introduced to ensure that queued operations have completed before continuing on the host. In OpenCL and CUDA, control flow on the host is still simple sequential code, with synchronization taking the form of a function that blocks until the device is finished.

To perform work on the GPU, the host can invoke code to be executed by the GPU, called a *kernel*. The host passes all parameters required by the kernel to the compute API, usually including references to memory it has previously allocated on the device. Typically, a kernel runs for a relatively short duration and the overall program involves the host queuing many kernels to be executed. These generally run in sequence on the device.

Kernels are typically supplied by the host code to the compute API in source code form. Final compilation of kernels only happens during runtime of the program, with the API runtime or device driver being responsible for compiling it for the device actually in use. While there are models where kernels are first compiled to some intermediate representation at program build time, all APIs considered here support supplying plain source code at runtime, and this is the only model we will use here. OpenCL and CUDA kernels are given in C or C++ with some language extensions.

A kernel running on the device is comprised of potentially many threads, organized in a two-level grid. The total amount of threads and their structure is determined by the host when launching the kernel and remains static while the kernel is executing. Individual threads are are grouped into *thread blocks* and there is a *grid* of such blocks. Each thread can access its own position in its block, as well as which block it is part of. OpenCL and CUDA allow specifying the block and grid sizes in up to three dimensions as this is often a convenient model, but this is purely an affordance for programmers and only affects how the positions are expressed.

(This is one of the larger terminology differences between the APIs. In OpenCL and WebGPU, thread blocks are referred to as *workgroups*. A single thread is a *work-item* in OpenCL and an *invocation* in WebGPU. We will generally use the CUDA terminology

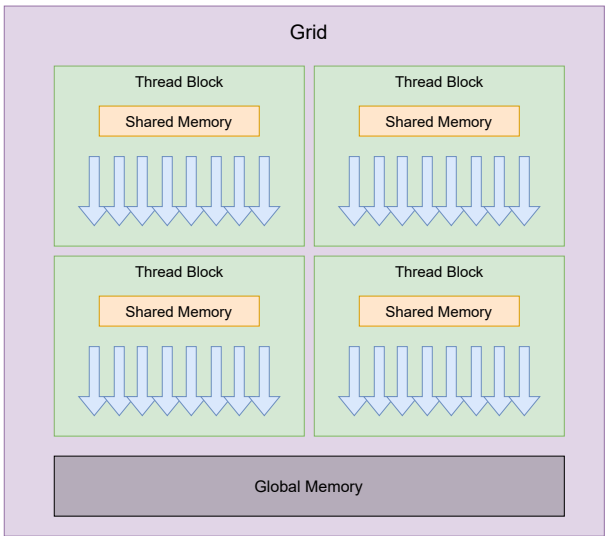

Figure 2.1: GPU Memory Hierarchy. Individual threads are illustrated using a single arrow and contain their own private memory and registers.

presented above as that is what the Futhark compiler uses internally, except when referring to specific WebGPU functions or similar.)

Unlike the division into up to three dimensions, the division into blocks of threads and the grid of blocks is very meaningful. Threads in the same block can communicate and cooperate with the ability to access shared memory and synchronize with each other, while threads from different blocks execute completely independently. A block is generally scheduled on the device as a single unit and threads in it share certain resources. As a result, the choice of block size can have effects on performance. It is also limited, with a generic maximum block size for all kernels, but also a kernel-specific maximum depending on how much of the shared resources each thread requires.

There are several different types of memory available on GPUs: global device memory accessible to all threads, private memory for each thread, as well as some *shared memory* (CUDA; *local memory* in OpenCL and *workgroup memory* in WebGPU) that is shared among the threads in the same block. Private and shared memory are much more limited in size than global memory, but also much faster. Memory transfers to and from the host can only access global device memory, with the other two being only directly accessible from within kernels. The thread hierarchy as well as the memory hierarchy is illustrated in Figure 2.1.

Like for block size, OpenCL and CUDA allow the host to determine the size of shared memory used when invoking the kernel. On CUDA, only a single dynamic shared memory allocation is permissible, but it can be split up freely in the kernel using normal C pointer arithmetic and casting.

In reality, there is also a level of thread grouping below the block level: Threads don't execute independently, but instead as part of a *warp* (CUDA; *sub-group* in OpenCL). The size of a warp is fixed by the hardware, and all threads inside a warp execute *in lockstep*, i.e. in a SIMT/SIMD fashion. This means that they always execute the same instruction at the same time, allowing parts of hardware such as instruction decoding to be shared between threads in a warp. This has certain performance implications, and CUDA and OpenCL also provide some warp-level primitive operations, such as scans or reductions. WebGPU does not currently specify the existence of warps, nor any warp-level operations, though there is a proposal to add them in the future.[1]

[1] `https://github.com/gpuweb/gpuweb/issues/4306`

```
// Kernel:
extern "C" __global__ void plus2(int *input, int *output, int n) {
  int idx = threadIdx.x + blockIdx.x * blockDim.x;
  if (idx < n) {
    output[idx] = input[idx] + 2;
  }
}

// On the host:
void call_plus2(CUfunction kernel, int *input, int *output, int n) {
  CUdeviceptr input_d, output_d;
  cuMemAlloc(&input_d, len * sizeof(int));
  cuMemAlloc(&output_d, len * sizeof(int));
  cuMemcpyHtoD(input_d, input, len * sizeof(int));

  int block_size = 256;
  int grid_size = (len + block_size - 1) / block_size;
  void **args = malloc(3 * sizeof(void *));
  args[0] = (void *)&input_d;
  args[1] = (void *)&output_d;
  args[2] = (void *)&len;
  cuLaunchKernel(kernel,
    grid_size, 1, 1,
    block_size, 1, 1,
    0, 0,
    (void *)args, NULL);

  cuMemcpyDtoH(output, output_d, len * sizeof(int));

  cuMemFree(input_d);
  cuMemFree(output_d);
}
```

Listing 2.1: Simple CUDA kernel that adds 2 to each element of an array, as well as an example of invoking it. Error handling is omitted.

In OpenCL and CUDA, a kernel is simply a function that is invoked in each thread on the device. Arguments passed from the host to the device are declared as formal function parameters. On the host side, arguments are simply set in order using a separate function or passed as an array when dispatching the kernel. In WebGPU, the kernel interface is somewhat more complicated, as we will see in the next chapter.

Listing 2.1 shows a simple example CUDA kernel that populates an output array by adding two to every number in an input arrray. We can see that it is a relatively normal C function, though it has access to the built-in values `threadIdx`, `blockIdx`, and `blockDim`, describing the current thread's location in the two-level grid introduced earlier. Since the number of threads must be a multiple of the block size, there can be an access of threads at the end which should not read from or write to the arrays, making the conditional necessary. The example also contains some host code to invoke this kernel, though it assumes that initialization, including compiling the kernel, has already been done. Note that in the full program, the kernel source could would be embedded as a string and handed to the CUDA API to be compiled, not just a function next to the host code like it is shown in the example. We can see how the host allocates memory on the device to correspond to the host-side `input` and `output` buffers and is responsible for copying data back and forth. It also invokes the kernel, passing in the required arguments as well as the block and grid sizes. In the next chapter, we will see a WebGPU program performing the same task to compare.

2.2 Futhark

2.2.1 The Futhark Language

Futhark is a purely functional data-parallel array programming language [11, 12]. A major design goal is the ability to compile it to efficient parallel code, including to run with high

```
-- Our running 'plus2' example.
def plus2 (xs: []i32) = map (+2) xs

-- We can also annotate arrays with size types. Here, we also could have
-- used `length xs` in the body instead.
def average [n] (xs: [n]f32) = reduce (+) 0 xs / f32.i64 n

-- Here the size types enforce that both arrays are the same size. This
-- example also illustrates the support for nested parallelism, with
-- both maps having the potential to be run in parallel.
def mat_add [n] [m] (xss: [n][m]i32) (yss: [n][m]i32) =
  map2 (\xs ys -> map2 (+) xs ys) xss yss

-- Size-dependent types are also supported, with the input parameter
-- appearing as the size in the output type. Functions can also be
-- generic over types.
def my_replicate 't (n: i64) (v: t) : [n]t = replicate n v
```

Listing 2.2: Some simple Futhark programs.

performance on GPUs while being easier than manually writing CUDA or OpenCL code. It comes with an optimising ahead-of-time compiler that supports various targets, such as CUDA and OpenCL as well as sequential or multi-threaded C code. The language is in the ML family and has a static type system.

Futhark is not intended to be a general-purpose language for writing whole program in. Instead, it can be used to implement particularly compute-intensive parts of an application that are amenable to parallel computation. These can then be used by the overall program by integrating with the code that the Futhark compiler generates.

Notably, the Futhark compiler does not parallelize code written in a sequential style. Instead, the language provides ways of expressing data-parallel algorithms, chiefly using *Second-Order Array Combinators* (SOACs). These are functions such as `map`, `filter`, `reduce`, and `scan` that allow operating on entire arrays in bulk. The language is designed to allow these to be executed in parallel, including in advanced scenarios such as nested parallel operators.

The focus on generating efficient parallel code imposes some restrictions on the language. For example, multi-dimensional regular (rectangular) arrays are supported, but non-regular arrays are not allowed. There is also no support for recursive calls in functions, although a sequential `loop` construct can be used to effectively express tail-recursion.

See Listing 2.2 for some simple example Futhark programs. We will not describe the entire language in-depth because, as we will see next, in this project we do not work much with the surface language, other than for testing and evaluating what Futhark programs we may not be able to support in our backend.

2.2.2 Futhark Compiler Overview

Like many other compilers, the Futhark compiler can be roughly divided into three major parts: the frontend, the middle-end, and the backend. The frontend and middle-end together consume Futhark programs, type-check and optimise them, while transforming them into a variety of *intermediate representations* (IRs). At this stage there are already multiple potential *pipelines* consisting of a sequence of compiler passes. Which is used depends on which target the program is being compiled for, with different pipelines using and ultimately producing different IR variants that for example express parallelism in different manners.

For our purposes, there is a single GPU pipeline used for the existing GPU targets such as CUDA and OpenCL. This pipeline produces an IR variant called `GPUMem` that contains parallel flat segemented operations designed to match the semantics of GPU kernels and explicit GPU memory information.

The first step in the backend is compiling this IR futher to `ImpCode` which is a relatively simple imperative language. It is parameterized for different use-cases that extend it with different operations. The kernel code running on a GPU and the host code running on a CPU and invoking the kernels are both expressed as `ImpCode` but with different extensions. For example, kernel code can get the current thread and block indices and use atomic operations while host code can launch kernels with a special operaton.

Most of the structure is identical to all `ImpCode` variants. There are `for` and `while` loops, `if` statements, and function calls optionally returning values. Variables are split into scalars and memory blocks. Scalars have a fixed primitive type, such as an integer of a certain size or a boolean. Memory blocks are untyped, though they are annotated as being located in a specific memory *space*, which is used to express for example the difference between global and shared memory on GPUs.

The supported primitive types are integers (8-, 16-, 32-, and 64-bit), floating point numbers (16-, 32-, and 64-bit) and booleans. There is also an informationless `Unit` type. Integer types are not associated with a signedness, instead individual operations assume a certain signedness when relevant. For example, addition is ultimately the same operation for signed and unsigned integers and so no sign information is necessary. On the other hand, comparison operators exist in both signed and unsigned variants.

As mentioned, memory blocks are untyped. Instead, each access to memory happens at a specific type. `ImpCode` contains explicit `Read` and `Write` statements to interact with memory that contain both a type and an index to access. Arbitrary expressions cannot access memory buffers, requiring a `Read` into a scalar local first. There are also statements for allocating and freeing memory blocks, though there are restrictions on these depending on the context. For example, host code can allocate and free global device memory, while kernel code can allocate shared memory but not futher global memory. Memory variables are effectively treated as pointers, with a `SetMem` statement that can change which memory block a name refers to. A `Copy` statement exists specifically to copy entire blocks of memory from one buffer to another.

Lastly, there are some additional miscellaneous statements for debugging and tracing as well as an `Assert` for error checking. This will be discussed in more detail later.

2.2.3 The GPU Backends

The translation from `GPUMem` to the intial GPU `ImpCode` variant, which we will call `ImpGPU` for short, is largely identical for all GPU backends with only small differences for different ultimate targets, and we were able to use the existing setup without further modification.

The final step is compiling the `ImpGPU` program to an output program invoking the corresponding APIs. For the CUDA and OpenCL backends, the kernels are compiled to the respective C variant used by the API while the host code is typically also compiled to C, calling into the API, though there is also support for generating Python host code instead. Implementing these two steps, compiling `ImpGPU` to output kernel code and corresponding host code targeting WGSL and the WebGPU API respectively, is the main contribution of this thesis.

An `ImpGPU` program is initially comprised of a set of functions and constants for the host side. Embedded in function bodies, in the context where they are to be invoked, the are kernels to be run on the GPU. The conceptual model is that the kernels have access to values from the surrounding host code, both scalar values such as local variables and arrays allocated in device memory. In addition to the kernel body itself, expressed itself in `ImpCode`, the `ImpGPU` kernel also contains metadata about which values are used, as well as other information such as expressions to compute the correct block and grid sizes. All free variables in the kernel body, i.e. those used but not declared in it, come from the surrounding host code and are turned into kernel parameters. The backend is responsible

both for compiling the kernel body and for generating the code required to pass all these values to the kernel and/or compute API.

Compiling from this form to the final output happens in two steps. First, one pass compiles the kernels to their final output source code, while assembling various information required to generate the final host code. This pass has some backend-specific parts, but there is only one for all the existing GPU backends, since it largely works the same way for all of them. It outputs a program largely identical to the input `ImpGPU` program, except with the embedded kernels replaced with a simple `LaunchKernel` operation and the afore-mentioned metadata and kernel source code attached.

Then, another pass generates C host code from this intermediate program. This includes, again, translation from `ImpCode` to C, but this time for the host. In principle, there could be one pass doing this for every GPU API supported, but to avoid the associated code duplication, it instead targets an internal GPU abstraction layer providing a uniform interface. There are hand-written C implementations of this abstraction layer for each of the compute APIs, so that the code generation does not need to differentiate much between them.

In the remainder of this section, we will briefly explain some more concepts in the `ImpGPU` representation to already give some context on why we are interested in certain WebGPU features. We will then go into more detail on our translation of them into WebGPU and WGSL later in chapter 4, once we have introduced both properly in chapter 3.

2.2.4 Kernel Uses

The values that should be made accessible to the kernel are called *kernel uses*. They are divided into three categories. Scalar uses and memory uses are what their names indicate, both referring to local variables available when the kernel is launched. The third category are *constant uses*. They do not name an existing variable on the host but instead given an expression and a name for the value of the expression that should be available in the kernel. The name stems from the fact that the expression can be evaluated already when the program is started, when compiling the kernel at runtime, instead of requiring the local context of where it is ultimately launched. In addition to constant literals, these expressions can also refer to so-called *size values*. This encompasses certain quantities queried from the GPU device, such as available shared memory or cache sizes, as well as tuning values. The latter is a mechanism to adjust how the program chooses between several different options for exploiting parallelism. For our purposes we can just treat these as more constant values. See [15] for details.

2.2.5 Block and Grid Size

The grid size, i.e. the amount of thread blocks in each dimension, is always given as simple expressions that must be evaluated on the host where the kernel is launched. The block size comes in two flavours: Either a plain expression like the grid size, or a constant expression like those explained above for constant uses. Each dimension can individually be either an expression or a constant expression. We will later see that constant block sizes are more compatible with WebGPU, so it is convenient that the compiler already distinguishes between the two options.

3 WebGPU

WebGPU is an API designed to allow accessing the capabilities of GPUs on the web. It is currently defined by a work-in-progress W3C Working Draft and is eventually intended to become a W3C Recommendation [16]. There already exists a related API on the web platform, WebGL. WebGL is modelled closely on the OpenGL API and focused on graphics rendering use-cases [13]. On the other hand, WebGPU is modelled more closely on a newer generation of native GPU APIs such as Khronos' Vulkan, Microsoft's DirectX 12, and Apple's Metal. It matches the capabilities of modern GPUs more closely, and also includes explicit support for general compute workloads in addition to graphics rendering.

WebGPU is still relatively new and support in browsers is limited. Chromium-derived browsers have the most support, with it enabled by default in Chrome and Edge on Windows. Support in other major browsers and on other platforms is in-progress and can often be enabled using experimental flags.[1]

As a web API, WebGPU has some unique design goals compared to the other APIs mentioned. It is intended to be portable across a wide variety of hardware and operating systems and thus designed to be implementable on top of all of the native APIs mentioned earlier, and restricted to capabilities supported by all of them. There are also very strict security requirements, as the primary use case involves running potentially untrusted code from the web. The underlying native APIs can produce undefined behaviour, both on the host and on the device, leading to unpredictable results and potential security vulnerabilities. As a result, all WebGPU commands must strictly validate their inputs, and the API must be restricted enough to ensure that no undefined behaviour can occur if this validation succeeds. It may also be necessary to limit an individual page's use of GPU resources to keep the overall system responsive.

In addition to security, the web also has stricter privacy requirements than most applications. A web page should not be able to fingerprint individual users based on their hardware, so a browser may be required to not expose exact hardware capabilities. And lastly, some of the API design is to discourage use of the API that could degrade the user experience, for example by only providing asynchronous versions of some operations to lead developers away from patterns that could make the site or browser unresponsive.

3.1 The WebGPU API

Designed as a web platform API, WebGPU is most commonly a JavaScript API, though as we will see later, there are bindings for other languages. There is a top-level `GPU` object exposed as `navigator.gpu` in browsers. This is mainly used to a acquire an *adapter* and then a *device*. An adapter represents an overall implementation of the WebGPU API consisting of the underlying native API and the browser implementation on top of it. A device is created from an adapter and represents a logical device, enabling concurrent and isolated use of a physical device by more than one logical device. Once created, it is the main object used to interact with the WebGPU API.

While compute-focused GPU APIs such as CUDA and OpenCL typically directly offer primitive operations such as launching kernels and copying memory between the host and the device, the structure of WebGPU is closer to that of modern graphics APIs. Most commands interacting with the GPU are recorded into a *command buffer* which is then submitted to a queue to be executed. Compute kernels are part of a shader module that

[1] `https://caniuse.com/webgpu`

has to be invoked as part of a *compute pipeline* with a variety of meta information specified. In the rest of this section, we will present the parts of the WebGPU API most relevant to the implementation of the Futhark backend and point out the most relevant differences to OpenCL and CUDA.

Throughout this section, we will include JavaScript code snippets illustrating using the API to implement a simple example program. Like our earlier CUDA kernel from Listing 2.1, it involves an input and output buffer, as well as a single kernel computing the output by adding 2 to each element of the input. As before, error handling and some details are omitted to keep the example concise and more readable.

3.1.1 Initialization

As already discussed, from the top-level API object we first acquire an adapter, and using that, we can request a device. As part of requesting a device, we can request additional optional features, and request limits past the default values.

Optional features are a way to provide additional features that may not be available on all WebGPU implementations. Their availability can be queried on the adapter before requesting a device. They must be explicitly requested if the application wants to make use of them. The current list of features [WebGPU §3.6.1] includes, among others, support for additional texture and buffer formats, and most relevantly for us, support for half-precision floating point numbers in kernels.

Limits describe numerical maximum values supported for various properties. There are many limits [WebGPU §3.6.2]. The ones most relevant for our purposes include maximum size of buffers, the maximum size of kernel shared memory, and the maximum thread block size. The specification provides a default value for each limit that every implementation must support, but higher values can be requested by an application. This enables more capabilities but potentially at the cost of portability, since not all implementations may support the higher limits.

Acquiring an adapter and a device are both asynchronous operations. In JavaScript, they return a promise that will resolve at some later point. As we will see later, asynchronous operations are not entirely straightforward for us to make use of as we do not directly generate JavaScript with its `async/await` syntax.

Once a device has been acquired, we can start using it to create other API objects such as buffers and pipelines. Each device also comes with an associated primary *queue* that work can be submitted to. It will then be completed asynchronously. To synchronize with work being finished on the device, a callback can be registered that is invoked once all work, that has been submitted to the queue when the callback is registered, is done. Notably, unlike in OpenCL and CUDA, there is no built-in synchronous way to wait on the host until device work has completed.

```
1 const adapter = await navigator.gpu.requestAdapter();
2 const device = await adapter.requestDevice();
```

3.1.2 Shaders and Kernels

Programs to be run on the GPU are called *shaders* in general. There are several flavors of shaders, relevant mostly for graphics rendering. For this project, we are only concerned with compute shaders, which can be used for general-purpose computation without involving the GPU graphics pipeline. In this thesis, we will often refer to compute shaders, or specific entry points in them, as *kernels* which is how the equivalent concept is generalled called in compute-focused APIs such as OpenCL and by extension also internally in the Futhark compiler.

Shaders are given in source code form and written in the *WebGPU Shading Language* (*WGSL*, see section 3.2) designed for this purpose. An entire shader is also called a *module*, consisting of one or more *entry points* that can be invoked by the host and additional declarations describing the shader's *interface*, i.e. what inputs it receives and what resources it has access to. For our purposes, the interface consists *pipeline-overridable constants* and bound buffers, both of which are discussed in more detail later in this section.

```
1 // Let `shader` be the text of Listing 3.1.
2 const shaderModule = device.createShaderModule({ code: shader });
```

3.1.3 Buffers

GPU memory is represented by buffer objects. Buffers are of a fixed size specified when creating them. Additionally, at creation time, a set of valid *buffer usages* must be specified, restricting how the buffer may be used later. Example usages include vertex and index buffers used for rendering, or uniform buffers for uniform data based to shaders (graphics or compute). These are the usages relevant in our context:

- `UNIFORM`: Can be used to pass uniform read-only data to shaders.
- `STORAGE`: Can be bound as general read-write memory in shaders.
- `COPY_SRC`: Data can be copied from this buffer to other buffers.
- `COPY_DST`: Data can be copied from other buffers to this one.
- `MAP_READ`: Can be mapped as read-only memory.
- `MAP_WRITE`: Can be mapped as read-write memory.

Mapping a buffer refers to making it available as normal memory to the host side that it can read (and potentially write to) directly. While a buffer is mapped, it cannot be used for other purposes until explicitly unmapped. (Note that this does not imply the ability to directly access device memory from the host, the API implementation can copy the data to host memory when mapping and, for read-write mappings, copy it back when unmapped.)

Notably, if a buffer has the `MAP_READ` usage, the only other usage permissible for the same buffer is `COPY_DST` [WebGPU §5.1.2]. The same is true for `MAP_WRITE` but with `COPY_SRC`. This means that any buffer that can be bound in a shader cannot also be used for mapping on the host. An intermediate buffer is required instead. These restrictions are unlike OpenCL and CUDA, where allocated global memory can be freely used in kernels and copied to and from.

Buffers can be written to from the host either by mapping them or by using the `writeBuffer` method of a `GPUQueue`. The latter method allows copying data from host memory to a GPU buffer and only requires the `COPY_DST` usage. On the other hand, mapping is the only way to read data from the GPU back to the host. Thus, to read any data computed by a kernel back to the host, we must first copy on the device side to a separate `MAP_READ` readback buffer, which can then be mapped on the host.

It is also important to note that mapping a buffer is currently always an asynchronous operation, like some of the initialization steps discussed above. There are discussions around relaxing this restriction, but it is unclear whether a synchronous/blocking variant will be provided in the future.[2]

[2] `https://github.com/gpuweb/gpuweb/issues/2217`

```
const input = device.createBuffer({
  size: BUFFER_SIZE,
  usage: GPUBufferUsage.STORAGE | GPUBufferUsage.COPY_DST,
});
const output = device.createBuffer({
  size: BUFFER_SIZE,
  usage: GPUBufferUsage.STORAGE | GPUBufferUsage.COPY_SRC,
});
const readback = device.createBuffer({
  size: BUFFER_SIZE,
  usage: GPUBufferUsage.MAP_READ | GPUBufferUsage.COPY_DST,
});

const params = device.createBuffer({
  size: 4,
  usage: GPUBufferUsage.UNIFORM | GPUBufferUsage.COPY_DST,
});
```

3.1.4 Pipelines and Layouts

Invoking shader entry points happens as part of executing a pipeline. There are two types of pipelines, `GPURenderPipeline` and `GPUComputePipeline`. We deal only with compute pipelines. A pipeline specifies the shader module and entry point to use, as well as what inputs and outputs are available in the form of a *pipeline layout*. This must match the declarations in the shader source code.

Giving an entry point access to a memory buffer is done by binding it to a specific slot. Buffer slots (variables in the shader module that buffers can be bound to) are divided into groups. Each slot is annotated with a group index and an index within the group in the shader source. On the host, a *bind group layout* is created that must match the shader. Each entry in the layout declares what binding index it is for, what type of buffer can be bound, and which shader stage it is visible to. (We only consider the `compute` shader stage here.) The layout does *not* define the specific buffer that will eventually be bound however.

In scenarios where the different entry points are executed after each other and share some, but not all, of the same bindings, the grouping mechanism allows reducing overhead by only changing some of the groups. We do not make use of this functionality however, and only ever use a single bind group.

Since a single module can contain multiple entry points, not all entry points may use all declared bind slots. It is valid to use a pipeline layout that omits slots declared in the module that are not *statically used* by the entry point specified. All slots declared in the layout must also later be supplied with a concrete buffer when the kernel is dispatched.

As we have mentioned before, in Futhark's GPU backends, some values are supplied to the kernel as constants when it is compiled. For this, we use *pipeline-overridable constants* which are simple scalar values passed to the shader as read-only values. Their values are already specified when the pipeline is created, not when it is queued for execution.

The block size for compute kernels must be specified either as a constant in the shader source, or as a pipeline-overridable constant. This means that it is fixed once the pipeline has been created.

While there is an explicit compilation step of shader source text into a shader module and this should report problems such as syntax and type errors, the WebGPU specification explicitly allows for shaders to only be fully compiled to their final form when creating a pipeline, with access to the values of pipeline-overridable constants and to the bind group layouts. As a result, pipeline creation is a potentially expensive operation.

```
const bgl = device.createBindGroupLayout({
  entries: [
    { binding: 0, visibility: GPUShaderStage.COMPUTE,
      buffer: { type: "read-only-storage" } },
    { binding: 1, visibility: GPUShaderStage.COMPUTE,
      buffer: { type: "storage" } },
```

```
    { binding: 2, visibility: GPUShaderStage.COMPUTE,
      buffer: { type: "uniform" } },
  ]
});

const blockSize = 256;
const pipeline = device.createComputePipeline({
  layout: device.createPipelineLayout({ bindGroupLayouts: [bgl] }),
  compute: {
    module: shaderModule,
    entryPoint: "plus2",
    constants: {
      "block_size": blockSize,
    }
  },
});
```

3.1.5 Invoking Kernels

With a pipeline in place, a kernel can be invoked. First, a set of bind groups must be created that match the bind group layouts specified when creating the pipeline. These match each slot in the layout with a concrete buffer to be used. Then, a *command buffer* is created that sets the pipeline, the bind groups, and then dispatches the kernel. While the block size for the kernel was specified as part of pipeline creation at the latest, the number of blocks is dynamically given as part of the dispatch. The finished command buffer is then submitted to a queue and executed asynchronously. Command buffers can be submitted multiple times, though we do not make use of this ability.

```
const hInput = new Int32Array(BUFFER_SIZE / 4);
for (var i = 0; i < BUFFER_SIZE; i++) { hInput[i] = i; }

const hParams = new Int32Array(1);
hParams[0] = BUFFER_SIZE / 4;

device.queue.writeBuffer(input, 0, hInput, 0);
device.queue.writeBuffer(params, 0, hParams, 0);

const bg = device.createBindGroup({
  layout: bgl,
  entries: [
    { binding: 0, resource: { buffer: input } },
    { binding: 1, resource: { buffer: output } },
    { binding: 2, resource: { buffer: params } },
  ]
});

const commandEncoder = device.createCommandEncoder();
const passEncoder = commandEncoder.beginComputePass();
passEncoder.setPipeline(pipeline);
passEncoder.setBindGroup(0, bg);
passEncoder.dispatchWorkgroups(Math.ceil((BUFFER_SIZE / 4) / blockSize));
passEncoder.end();
commandEncoder.copyBufferToBuffer(output, 0, readback, 0, BUFFER_SIZE);
device.queue.submit([commandEncoder.finish()]);

await readback.mapAsync(GPUMapMode.READ, 0, BUFFER_SIZE);
const readbackMapped = readback.getMappedRange(0, BUFFER_SIZE);
const data = new Int32Array(readbackMapped);

console.log("Ran plus2, got: ", data);

readback.unmap();
```

3.1.6 Emscripten and Native Implementations

As mentioned before and clearly indicated by its name, WebGPU is designed mainly to make GPU capabilities available on the web. It is however also more generally a very portable and reasonably simple GPU graphics and compute API, and has found use in other contexts. Dawn[1] and wgpu[8] are the WebGPU implementations of Chrome and

Firefox, written in C++ and Rust respectively. While they are used to implement WebGPU on the web by the two browsers, they also expose a native API that other programs can use to portably make use of GPU capabilities.

While both projects provide their own specific APIs and extensions past the current WebGPU specification, a shared C WebGPU header has also been created that aims to standardize a C interface mirroring the web specification[7]. Both Dawn and wgpu provide implementations of this header. Additionally, Emscripten[4] also supports the same header. Emscripten is a toolchain for compiling native code (typically C and C++) to JavaScript and/or WebAssembly that can be run on the web. Using this, a C application can be written against the native WebGPU header and then be compiled to JavaScript and WebAssembly that make use of the browser's WebGPU implementation.

3.2 WGSL

The WebGPU Shading Language[2], abbreviated WGSL, has been developed along with the WebGPU standard itself for writing shaders to be used with WebGPU. Like the API itself, it is currently specified as a W3C Working Draft. It is a relatively simple statically typed imperative language with syntax close to that of the Rust language.

A simple example compute shader is shown in Listing 3.1. It continues our "plus 2" example, adding 2 to each element of an input array and writing the result to a separate output array, illustrating many of the concepts discussed below.

The basic built-in scalar types are `bool`, `i32`, `u32`, and `f32`. 16-bit floating point numbers are supported as `f16` as an extension. Additionally, there are 2–4 element vectors of the scalar types and 2–4 x 2–4 element matrices, as well as the `atomic<i32>` and `atomic<u32>` types for atomic operations. Shaders can also define custom `struct` types. There are two variants for arrays, `array<T>` for runtime-sized arrays and `array<T, N>` for fixed-size arrays. There is also a `ptr<AS,T,AM>` type that will be discussed in more detail later. Lastly, there are some additional built-in types that are not relevant in our context, for example for handling textures when rendering graphics.

There are several types of *value* declarations:

- `const` declarations give a name to the value of an expression that must be constant at shader creation time. They can appear in module and function scope.
- `override` declarations give a name to a value that is a pipeline-overridable constant, where the value is either specified by the host code at pipeline creation time as discussed earlier, or the value of an *override expression* (effectively a constant expression possibly referring to other `override` names). They can only appear in module scope.
- `let` declarations can only appear in function scope and always have an initializer expression. They give a name to the value of the initializer expression as evaluated at runtime when the `let` statement is executed.

In addition to these value declarations, there are also *variable* declarations using the `var` keyword. These associate a name with a corresponding backing memory allocation that is allocated for the variable, which can be mutable at runtime (unlike the value declarations). Variables can be in one of multiple *address spaces*, depending on how they are declared and whether they are declared in module or function scope. See section 3.2.1 for more details.

Many items — among others module-scoped variables, functions, and function parameters — can be annotated with *attributes*. For example, the `group` and `binding` attributes are used to specify how to map the bind groups set on the host side to shader-side variables. Functions are declared as a compute shader entry point using the `compute` attribute.

```
@group(0) @binding(0)
var<storage, read> input: array<i32>;
@group(0) @binding(1)
var<storage, read_write> output: array<i32>;

struct Params {
  count: u32,
}

@group(0) @binding(2)
var<uniform> params: Params;

override block_size: i32;

@compute @workgroup_size(block_size)
fn plus2(
  @builtin(global_invocation_id)
  global_id: vec3<u32>,
) {
  if (global_id.x >= params.count) {
    return;
  }

  output[global_id.x] = input[global_id.x] + 2;
}
```

Listing 3.1: A simple WGSL shader

Access to certain built-in values such as the thread ID is given by annotating entry point parameters with the `builtin` attribute, for example as `@builtin(global_invocation_id)` in Listing 3.1. Note that entry point function parameters can only be used for built-in values like this; arguments from the host are always passed through items declared at module scope.

Otherwise, WGSL largely supports a set of statements and expressions common for imperative languages, including, among others, structured control flow with `if` statements and `for` and `while` loops, as well as function definitions and calls. As is common for shader languages, it also supports a variety of built-in functions. For our purposes, the numeric, atomic, and synchronization built-ins are most relevant and will be discussed later, but there are others, for example for sampling textures. There is also a built-in `bitcast` function to convert between the different built-in types.

3.2.1 Address Spaces, Arrays, and Pointers

WGSL's handling of arrays and pointers deserves special mention. We have seen earlier that in Futhark's GPU backends, memory is generally untyped and has reference-like semantics where the specific buffer a name refers to can be changed at runtime. Thankfully, it does not assume the ability to perform arithmetic on pointers, as is valid in C. Even so, WGSL is a challenging target in this regard, as its array and pointer types are very restricted in their capabilities.

The language explicitly supports several address spaces in which the memory backing a variable may be located in to reflect the different types of GPU memory available to shaders. In Listing 3.1 we can explicitly see the `storage` and `uniform` address spaces used in the form of the `var<storage, _>` and `var<uniform>` declarations. The storage buffer declarations contain explicit *access modes*: `read` or `read_write`. For `uniform` variables, read-only access is implicit.

There are also several more address spaces, all implicitly with a `read_write` access mode:

- `private` for module-scoped variables private to a single invocation,
- `function` for function-scoped variables (always private to a single invocation), and

- `workgroup` for memory shared between threads in the same workgroup (CUDA *shared memory*).

All `var` declarations inside function bodies implicitly have the `function` address space, while declarations in module scope must have an address space explicitly declared. (There is also a `handle` address space, but it is only relevant when dealing with textures and samplers for graphics rendering.)

Pointer types contain information about both address space and access mode of the memory pointed to, leading to the fully general type `ptr<AS, T, AM>`, although for all address spaces other than `storage`, the access mode must not be written out, instead being implicitly determined as above.

As we have seen before, array types come in two forms: `array<T>` and `array<T, N>` for runtime-sized and fixed-sized arrays respectively. Runtime-sized arrays are only valid when used with the `storage` address space. This distinction carries into pointers, as the pointer type used to refer to an array is either `ptr<AS, array<T, N>, AM>` or `ptr<storage, array<T>, AM>`, including the array type as the pointee's type.

Pointers can be used within a function and passed as parameters to functions, but have various restrictions. They cannot be used as the return value of a function. Pointers to composite types such as vectors, arrays, or structures can be projected into pointers to the elements making them up, but there is otherwise no explicit pointer arithmetic or any ability to cast between pointers of different types. There are also no conversions between pointers and integers.

As an example, in C, an array of integers can be pointed to using an `int*`, a pointer to *an* integer, because it is possible to add to the pointer value to acquire pointers to other elements of the array. In WGSL, a `ptr<function, i32>` only gives access to memory storing exactly one integer. A `ptr<function, array<i32,5>>` gives access to memory storing five integers. The former can be acquired from the latter using normal indexing expressions.

Pointer types are also not so-called *storable* types [WGSL §6.4.1]. This means it cannot be the type of a *variable* declaration, only of a *value* declaration such as `let` declarations and function parameters [WGSL §7.3]. As a result, no mutable pointer variables are allowed in WGSL.

Lastly, there are no generic functions and pointer arguments must be fully specified. This makes it impossible to write functions that can for example take any kind of buffer as an argument. Functions taking a shared memory buffer can only work for a specific size of buffer since runtime-sized arrays in the `workgroup` address space are not allowed.

3.2.2 Uniformity

WGSL provides some built-in operations that involve cooperation between the threads in a thread block, most notably the `storageBarrier` and `workgroupBarrier` functions. They can be used to ensure that all memory and atomic operations that are executed by a thread (affecting the `storage` or `workgroup` address spaces respectively) before the barrier are visible to all threads in the same block after the barrier.

Both functions are a form of *control barrier*, which means that the program is executed as if all threads in the block execute the barrier instruction concurrently. Some compute APIs also support memory *fence*s that only have an effect on memory operation ordering, without the control flow synchronization. They are currently not exposed in WGSL, due to a lack of them in the Metal API, but could be added in the future if Metal starts supporting them.[3]

[3] `https://github.com/gpuweb/gpuweb/issues/1374`

In all compute APIs, executing a control barrier is only valid in *uniform control flow*. This means that all threads in the block must reach the barrier. For example, a barrier in a conditional branch that only some threads reach is not allowed. In general, this is a *dynamic* property: What is important is that, in any given run of the program, all threads reach the barrier. In APIs such as CUDA and OpenCL, the behaviour when this property is violated is simply not defined. In practice, it can lead to kernel execution hanging indefinitely or returning unexpected results.

WebGPU and WGSL are explicitly designed to prevent this kind of undefined behaviour in the underlying native API, even in the presence of maliciously crafted input. As a result, the WGSL specification defines a *static* uniformity analysis to ensure that barriers are only placed in uniform control flow. Explaining the analysis in detail [WGSL §14.2] would exceed the scope of this section, but importantly the analysis must inherently be conservative. There are necessarily some programs where, dynamically at runtime, all barriers would be reached in uniform control flow, for example when given the correct input data. But this cannot be proven statically, and so WGSL cannot permit such programs.

We will see later how this static uniformity analysis caused problems while implementing the WebGPU backend, as well as how we solved them.

4 WebGPU Futhark Backend

In this chapter we will describe the bulk of the practical work of this project, actually implementing the new WebGPU backend. As we have described before, this consists of two main parts: Generating WGSL shader code for the GPU kernels, and getting working host-side code for the WebGPU API using a mixture of code generation and hand-written C code.

4.1 Generating WGSL Shaders

In contrast to all existing GPU backends where kernels are written in a variant of C, WebGPU shaders must be provided in WGSL as described earlier. At first glance, `ImpCode` and WGSL match up reasonably well. They are both imperative languages and so overall control flow in the kernel function body involving constructs such as `if` blocks and `while` and `for` loops can be translated straightforwardly. We have implemented a Haskell module with an AST definition for the subset of WGSL that we need to be able to generate, along with a pretty-printer to turn ASTs into WGSL source text we can include in the generated program. With that, the basis of compiling the body of a kernel is recursively traversing the statements and expressions making up the `ImpCode` kernel body and producing equivalent WGSL output.

However WGSL is in many ways more limited than the previous targets of CUDA or OpenCL C kernels in what it supports and this has required various bits of more involved translation. The remainder of this section will describe many of the challenges and how we solved them. There are also some problems that we were not able to resolve in this project, and as a result our backend does not support all valid Futhark programs. These will mostly be discussed separately in section 6.1, but we will mention some of them here in passing.

We have included a fragment of an example `ImpCode` program with an embedded kernel in Listing 4.1 to illustrate the general structure. It implements the same program as our earlier CUDA and WebGPU examples, adding 2 to each number in an array.

4.1.1 Integer Signedness

One of the more simple-to-resolve mismatches between `ImpCode` and WGSL is that WGSL, like most common programming languages, has signed and unsigned integer *types* and the semantics of operations such as comparisons or bit-shifts depend on the types of the involved variables. In `ImpCode`, signedness is always a property of individual operations, not of variables in general. We chose to always declare variables in WGSL as being signed and have then implemented functions that perform the correct unsigned operation on the signed integer type that get called when required. For example, a `CmpSlt` (signed less-than) expression gets translated to a simple expression using the `<` operator in WGSL, while a `CmpUlt` expression turns into a function call to `ult_i32`:

```
fn ult_i32(a: i32, b: i32) -> bool {
  return bitcast<u32>(a) < bitcast<u32>(b);
}
```

This is the same approach used in existing backends when generating C code, where signedness is also a property of the type.

```
var segmap_tblock_size_5228: i64
segmap_tblock_size_5228 <- get_size(main.segmap_tblock_size_5220,
    thread_block_size)
var segmap_usable_groups_5229: i64
segmap_usable_groups_5229 <- sdiv_up64 (d0_5197) (segmap_tblock_size_5228)
var mem_5238: mem@device
mem_5238 <- malloc(mul_nw64 (4i64) (d0_5197))@device
kernel  {
  blocks  { [segmap_usable_groups_5229] }
  tblock_size  {
    [const get_size(main.segmap_tblock_size_5220, thread_block_size)]
  }
  uses  {
    scalar_copy(d0_5197, i64)
    mem_copy(a_mem_5235)
    mem_copy(mem_5238)
    const(segmap_tblock_size_5228, get_size(main.segmap_tblock_size_5220,
    thread_block_size))
  }
  body  {
    var local_tid_5242: i32
    var block_id_5243: i32
    local_tid_5242 <- get_local_id(0)
    block_id_5243 <- get_tblock_id(0)
    var gtid_5231: i64
    gtid_5231 <- add_nw64 (mul_nw64 (sext_i32_i64 (block_id_5243)) (
    segmap_tblock_size_5228)) (sext_i32_i64 (local_tid_5242))
    if slt64 (gtid_5231) (d0_5197) then {
      var eta_p_5233: i32
      eta_p_5233 <- a_mem_5235<i32@global>[gtid_5231]
      var lifted_lambda_res_5234: i32
      lifted_lambda_res_5234 <- add32 (2i32) (eta_p_5233)
      mem_5238<i32@global>[gtid_5231] <- lifted_lambda_res_5234
    } else {
      skip
    }
  }
}
```

Listing 4.1: Example `ImpCode` fragment with an embedded kernel adding 2 to every element of an array.

4.1.2 Missing Primitive Types

More significantly, the only scalar numeric datatypes supported in WGSL are 32-bit signed and unsigned integers (`i32` and `u32`), as well as 16-bit and 32-bit floating point numbers (`f16` and `f32`). The `f16` type is only available as an optional WebGPU feature, which we require unconditionally. To increase compatibility, this could be adjusted to only require the feature if 16-bit floats are used in the Futhark program being compiled in the future. According to an investigation performed when the `f16` feature was proposed in 2022[1], it can be supported by implementations on a decent variety of common recent hardware and software combinations, but is still much less portable than WebGPU overall is.

Futhark, and by extension also `ImpCode`, support 8-bit, 16-bit, and 64-bit integers as well as 64-bit floating point numbers in addition to what is exposed in WGSL. We have not implemented support for 64-bit floats, which are only required if the source Futhark program actually usues them itself, but do support the other integer sizes.

64-bit integers are represented in WGSL as a 2-element vector of `i32`s. We have created, in hand-written WGSL, a library of functions operating on this representation to implement almost all of the required 64-bit operations, such arithmetic like addition or multiplication, and comparisons. These functions are used by the code generator to translate the equivalent expressions from `ImpCode`. An advantage of this approach is that the same buffers can be correctly interpreted as containing native 64-bit integers on the host and this `vec2<i32>` representation on the device, as they use the same memory layout (assuming little-endian architectures). This means we do not have to add in additional conversion steps or buffer (re-)allocation. The largest missing piece of our `i64` support is 64-bit division. This is somewhat complicated to implement by hand with only 32-bit division as a primitive, and we have assessed that our limited time was better spent elsewhere, so currently 64-bit divisions will only return correct results when the values involved fit into 32 bits.

Smaller integer types are represented as a full `i32` in WGSL, with their own associated library of operations. There is however an additional complication: With 64-bit integers we get the identical in-memory representation between host and device effectively for free, but representing each 8- or 16-bit integer with a full 32 bits clearly does not give the same memory layout. In order to support the representation without significantly modifying the host-side code generation, and to avoid increasing the memory footprint of arrays of 8- and 16-bit integers, we have decided to keep the in-memory representation the standard one also used on the host. In the kernel, memory buffers are always declared as containing `i32`s and all accesses to them go through functions that operate on the correct fourth or half of the full value respectively.

This is straightforward for reading, but more difficult for writing: The expectation is that different threads on the GPU are able write to different 8/16-bit elements of the array without synchronization, even when they happen to be stored as part of the same `i32` value. Thus all writes must happen using atomic operations instead, to avoid racing with writes to adjacent elements. The implementation of this for 8-bit integers is shown in Listing 4.2. Note that there is no expectation that the entire write itself is atomic here, so using two sequential atomic operations is fine. If two threads were to write to the same element, this would already be a data race in the input kernel.

This solution for smaller integers is very similar to how `f16` is handled in the existing C CPU backends, since 16-bit floats do not exist in standard C. In local variables, they are treated like 32-bit floats, exactly how we handle the integers. On the CPU, a 16-bit *integer* type is however always available, so that can be used as the type in memory, so that reading and writing only involves a cast instead of our more complicated functions.

[1] `https://github.com/gpuweb/gpuweb/issues/2512`

```
alias i8 = i32;

fn read_i8(buffer: ptr<storage, array<atomic<i8>>, read_write>, i: i32
) -> i8 {
  let elem_idx = i / 4;
  let idx_in_elem = i % 4;

  let v = atomicLoad(&((*buffer)[elem_idx]));
  return norm_i8(v >> bitcast<u32>(idx_in_elem * 8));
}

fn write_i8(buffer: ptr<storage, array<atomic<i8>>, read_write>,
            i: i32,
            val: i8
) {
  let elem_idx = i / 4;
  let idx_in_elem = i % 4;

  let shift_amt = bitcast<u32>(idx_in_elem * 8);

  let mask = 0xff << shift_amt;
  let shifted_val = (val << shift_amt) & mask;

  // First zero out the previous value using the inverted mask.
  atomicAnd(&((*buffer)[elem_idx]), ~mask);
  // And then write the new value.
  atomicOr(&((*buffer)[elem_idx]), shifted_val);
}
```

Listing 4.2: `read_i8` and `write_i8`

Boolean variables also require some extra consideration. WGSL has a `bool` type, but it is not defined as *host-shareable* [WGSL §6.4.2]. This means that in local variables we can simply translate `ImpCode`'s booleans to WGSL booleans, but we cannot use them as the element type of memory buffers. Instead we employ the same memory layout as for `i8` variables, with `read_bool` and `write_bool` functions implemented in terms of the corresponding `i8` variants.

Integers both larger and smaller than 32 bits are generally supported on GPUs in hardware, as can be seen from OpenCL's and CUDA's support for them. WGSL decided to hold off on them out of compatibility concerns with some potential underlying APIs[2], but will hopefully add support for them at some point in the future[3], which would make these workarounds unnecessary.

4.1.3 Typed Memory

WGSL also requires annotating memory buffers with their element type, unlike CUDA and OpenCL where pointers to memory buffers can simply be cast to the appropriate type when necessary. `ImpCode` does not contain type annotations for buffers. Instead the memory is accessed at types given in the individual `Read` or `Write` statements. In order to generate a declaration for the buffers used by kernels, we thus have to search the kernel body for all accesses to the buffers and record the types they are accessed at. If a single buffer is accessed at multiple types, we currently generate an error, as we have to declare a single type in WGSL. This could potentially happen if the same buffer is re-used for different purposes within a single kernel. This is generally not required, but sometimes done by the Futhark compiler as an optimization to reduce memory usage, or to re-use limited shared memory. As a potential future extension, this behaviour could potentially be disabled for the WebGPU backend entirely. A buffer can be re-used at different types in different kernels, since we generate a completely separate set of bindings for each kernel. We also investigated the possibility of instead generating multiple bindings at different

[2] `https://github.com/gpuweb/gpuweb/issues/229`
[3] `https://github.com/gpuweb/gpuweb/issues/273`

types for a single buffer, but WebGPU only allows this kind of aliasing if all bindings are declared read-only [WebGPU §14.1], limiting the usefulness of this approach.

4.1.4 Kernel Interface

In section 2.2.3, we described the three categories of values passed from the host to kernels: constant uses, scalar uses, and memory uses. On the C-based backends, scalar and memory uses are implemented as function parameters for the kernel, while constant uses are implemented as preprocessor definitions passed to the kernel compiler. In WGSL, kernel entry point parameters can only be used for access to built-in values such as the thread ID. For everything passed by the host, we need to generate appropriate module-scoped WGSL declarations instead.

Memory uses are the simplest: We have already discussed finding an appropriate type for them, and once we have that, we generate straightforward storage buffer declarations. One last complication to deal with are potential name collisions. We generate all kernels in a program into a single WGSL module, so we have to ensure that we do not generate bindings with the same name for two separate kernels. We do this by prefixing the module-scoped name with the kernel name to ensure it is unique. We then have to be careful during translation of the kernel body to replace all mentions of the name by the prefixed name. Another possible approach would have been to declare a local pointer to the module-scoped buffer in the entry point's body with the original name, and then always treat memory buffer names as being pointer-typed instead of array-typed while translating the kernel body. Bind slot indices are simply assigned in sequential order across all generated kernels. We do not make use of multiple bind groups, though if large programs ever exceed the maximum amount of slots permitted per group, we could use them to gain additional slots.

Constant uses fit WebGPU's pipeline-overridable constant concept fairly well. Their values are available when the kernels are being compiled and do not change, so we can pass them in when creating the pipelines. A major difference between pipeline-overridable constants and the preprocessor definitions used in the C-based backends is that override constants must be statically typed like everything else in WGSL. In practice, the Futhark compiler currently only generates 64-bit integer constant uses, so that is what we generate. In the future, it would also be possible to determine the type of the `ImpCode` expression if necessary. The only other complication is that, as described above, 64-bit integers are represented as a `vec2<i32>` and override constants can only be of scalar type. Thus we generate two `i32` override declarations and combine them at the start of the kernel body. The host-side expression is duplicated and the high and low halfs respectively extracted. We also have to ensure the uniqueness of the override declarations, like for memory above, though here we can avoid keeping track of the renames by using the original name for the function-scoped combined `vec2<i32>`.

The remaining category is scalar uses. In principle, these could also be implemented using override constants, but this would require creating a new pipeline and thus potentially triggering shader re-compilation every time a kernel is invoked, since the scalar values are more dynamic. We will see that we cannot always avoid creating a new pipeline for each dispatch, but we attempt to do this as little as possible. Thus, we make use of `uniform` buffers for the scalars. For every kernel, we generate a `struct` definition in the module containing a field for every scalar use and declare a `uniform` buffer binding containing a single instance that type. At the start of the kernel body, we copy every scalar into a local variable of the expected name.

4.1.5 Block Size

As described previously, WebGPU is unlike the other compute APIs supported by Futhark in that the block size must be provided in the shader itself using the `@workgroup_size` attribute. The value provided in the attribute can however be an override expression (for our purposes: the name of an override constant), saving us from having to actually modify the shader source at runtime. When the Futhark compiler generates kernels, some block sizes will be constants, while others may need to be computed at runtime when dispatching the kernel. On the shader side, we simply generate one override expression for each dimension specified in the kernel. The host code is ultimately responsible for dealing with the difference between constant and dynamic block sizes.

4.1.6 Shared Memory

This is another area where the WebGPU backend is fairly different from the other GPU backends. The kernel `ImpCode` contains `DeclareMem` and `SharedAlloc` statments that are used to declare shared memory to actually assign some region of shared memory to a previously-declared variable respectively. The `SharedAlloc` contains an expression for the size required for the buffer, which can actually be evaluted *on the host* before launching the kernel, rather than in the kernel itself. On the other GPU backends, a single dynamic shared memory allocation is declared in the kernel code. Its size is set to be the sum of all the allocations, computed on the host. In the kernel, this is then split into the required allocations using, effectively, pointer arithmetic. On WebGPU, this splitting is not possible as soon as more than one type is involved. Instead, we declare an array variable in the `workgroup` address space for each `SharedAlloc` encountered in the kernel body. The size is declared using yet more override constants, one per allocation, so that the host can compute each size and pass it in as part of pipeline creation.

An additional issue is that our functions for reading and writing 8- and 16-bit integers, presented earlier, do not work with these shared memory allocations. In Listing 4.2, we can see that the pointer is (and must be) explicitly declared with `storage` address space, so we cannot pass in a `workgroup` buffer. The obvious fix is to write a second set of functions taking a `ptr<workgroup, array<atomic<i8>>>` instead. We would then need to track which names refer to shared memory instead of normal memory buffers, so that for each `Read` and `Write` statement, we can pick which function to use. Unfortunately, this is also not possible, because runtime-sized arrays are only allowed with the `storage` address space (see section 3.2.1), so `ptr<workgroup, array<atomic<i8>>>` is not a valid type. On top of that, we will see in a moment that we need to support using a WGSL function called `workgroupUniformLoad` with our shared memory buffers, and it cannot be used with atomic types.

Since shared memory buffers are never copied directly to or from the host, we instead relax our requirement that they should keep the same memory layout. In shared memory, every `i8` and `i16` is represented by a full `i32`, just like in local functions. We can then use normal indexing and assignment expressions/statements to read/write from them. This also makes it possible to use the `workgroupUniformLoad` function. The main downside is that we waste limited shared memory space for what is effectively padding with such arrays.

4.1.7 Atomics

Kernels can also make use of a set of atomic operations on primitive types. WGSL only supports atomics with the `atomic<i32>` and `atomic<u32>` types, while `ImpCode` can generally express atomic operations on any integer types, and even supports atomic floating point addition. Our backend currently only supports kernels that exclusively use atomics

for 32-bit integers. Since we represent smaller integer types using a full `i32`, it would be possible to extend this to support some operations for those too, like an atomic compare-and-exchange. We can not, in general, use the built-in atomic arithmetic operations for them, since the behaviour when overflowing the limited number of bits would be incorrect. They could however be implemented by emulating them using a compare-exchange loop, potentially degrading performance to some extent. For 64-bit integers, even this is not an option. Without any atomic primitives that can affect 64 bits at once, implementing support for them efficiently is basically impossible. The only feasible implementation would be using the smaller atomics to implement a simple spin-lock guarding an ordinary `vec2<i32>` buffer, but it is unclear whether even this would be possible for the general case, since such a lock would necessarily involve barriers and/or memory fences. We will see in the next section that these are already somewhat problematic in and of themselves.

Even for the 32-bit integers we do support, there is an extra complication. In the existing backends, atomic operations are done on normal memory buffers that can also be read and written non-atomically. In WGSL, atomic operations are only possible on the `atomic<T>` types. Thus, when determining the type of a buffer, any atomic operation causes it to be promoted to have atomic elements. We must then generate an `atomicLoad` and `atomicStore` for every read and write to that buffer, even where the other backends perform ordinary reads and writes. Due to the lack of casting, unlike for normal buffers, we also have to choose between `atomic<i32>` and `atomic<u32>` for each buffer. The only atomic operations where the sign is relevant are the minimum and maximum operations. As long as only one version of these is used, we can determine a type to use. If both are, our backend produces an error.

4.1.8 Barriers, Fences, and Uniformity

As kernel-specific extensions to `ImpCode`, there are also statements for memory fences and barriers. As we have explained in section 3.2.2, these can be used to synchronize between the different threads in a single block. The Futhark compiler emits them for various kernels, for example for an efficient reduction kernel.

Barriers must be only be executed in uniform control flow, as discussed earlier. This restriction applies to CUDA and OpenCL too, and so the kernels output by the compiler already respect it. However, unlike in those APIs, WGSL's static uniformity analysis [WGSL §14.2] must also come to the result that they are in uniform control flow. Unfortunately, the kernels generated by the compiler contain patterns that do not pass the analysis. Concretely, barriers sometimes appear in conditional branches that depend on a value read from shared memory. All threads in the block read the same value, so we know that this control flow is uniform, but the analysis does not consider it to be.

WGSL does however provide a function that lets us provide extra information to the analysis: `workgroupUniformLoad` [WGSL §16.11.4]. This function can only be used in uniform control flow itself, but in that case it can be used to read a value from shared memory in all threads, that is then considered uniform by the analysis. This is exactly the scenario we need to support. In order to make use of it, we added a new kernel-specific `ImpCode` statement for uniform loads, for which we generate a call to this function. We then adjusted the kernel construction code to emit this new statement where required. On the existing GPU backends, a uniform load can simply be implemented as a normal load.

`workgroupUniformLoad` cannot be used to read atomic values. This means any given shared memory buffer can either be accessed atomically or with a uniform load, but not both. We generally perform the same analysis as for other buffers to determine whether it is accessed atomically or not, and promote normal reads and writes to atomic ones accordingly. A uniform load from a buffer determined to be atomic will result in a compiler error.

In addition to barriers, the compiler also generates *fences*. These can be used to ensure a consistent ordering on memory accesses visible between different threads. Their effect is weaker than that of a barrier, because they only affect the ordering of different memory accesses, and don't ensure the visibility of them to other threads like a barrier does. On the other hand, they are not required to be in uniform control flow, making them more flexible.

Unfortunately, WGSL does not currently support any memory fences. In our current implementation, they are simply discarded, but this does have the potential to lead to incorrect results. The plan for the future is to take advantage of the limited scenarios where the Futhark compiler generates them to generate different, but equivalent code. Specifically, they are used just after a write to memory in a way that the required effect can potentially also be achieved by using an atomic store instruction instead. It is however not entirely clear whether this is technically correct according to WGSL's memory model, and we have not finished a prototype implementation of this due to time constraints.

4.2 Host Code

The main goal of adding a WebGPU target to Futhark is making it possible to run Futhark programs on the web. This requires ultimately generating either JavaScript or a mixture of JavaScript and WebAssembly (Wasm) to be run by a web browser. One possible approach for this would have been to directly add the capability to generate this code to the Futhark compiler, but generating an entire new output language requires significant implementation effort. On the kernel side, doing this for WGSL was unavoidable, but for the host code there is another option, also used by the existing WebAssembly backend [14] for Futhark.

As mentioned in section 3.1.6, the Emscripten compiler can compile C and C++ code to JavaScript and Wasm. With its support for the shared `webgpu.h` header, it is possible for us to re-use large parts of the existing C host code generation and C runtime system of the Futhark compiler, but compiling it for the web platform instead of to native executables.

This means we did not have to re-implement code generation from `ImpCode` again for the host code, as it can be shared with the other C-based backends. The Futhark compiler has a small internal C GPU abstraction layer in the generated runtime system so that the compiler mostly generates code targeting this abstraction layer, which then has multiple implementations for e.g. CUDA and OpenCL. We thus added a new implementation the abstraction layer targeting the native WebGPU header.

4.2.1 `ImpCode` to C Host Code

We have already described the overall structure of the GPU backends in section 2.2.3. We first compile an `ImpGPU` program to WGSL shader source code, as described above, while also collecting a variety of meta-information about the generated kernels required to then generate the host code. All the existing GPU backends share a single pass for this, but WebGPU is sufficiently different to require its own implementation.

The overall output from this pass, including all the metadata, is also specific to the WebGPU backend. Initially, we also defined our own `ImpCode` variant by defining a custom `HostOp` type to extend it with. However we were later able to actually re-use the same type used by the existing backends instead, so we extracted it and some related types used for kernel metadata into a shared module now used by all GPU backends. This shared representation is what is compiled to C host code targeting the internal C GPU abstraction layer, so keeping it the same enabled us to re-use this step from the existing backends without further modifications.

This generates most of the program required. In order to actually implement the C GPU abstraction, we do however have to generate additional C code specific to the WebGPU

backend. This contains the metadata mentioned earlier, required to correctly compile and interact with the WGSL shader generated by the earlier step. It takes the form of a `wgpu_kernel_info` struct and an array of instances of this struct, one for each kernel. We will see what information this contains and how it is used in the next section.

4.2.2 The Runtime System

The compiled Futhark program consists not only of the code generated by the compiler for the specific program being compiled, but also of a runtime system that is written by hand in C. This includes some backend-agnostic bits, such as dealing with Futhark's binary data format or a generic device memory management API. The backend-specific part handles initialization of the underlying API and implements the GPU abstraction layer. The backend is required to define some types and functions used by the runtime system. What we need to implement can broadly be divided into three categories: Initialization and global context, memory operations, and kernel operations.

The context type used by our backend is very similar to that of the other backends. It contains various bits used by the runtime system, such as a free list of allocated GPU memory, and also the WebGPU context, including the adapter, the device, and the compiled shader module. Initialization is mostly straightforward and similar to the other backends, with the exception of handling the asynchronous nature of some of the involved WebGPU operations. This is discussed in detail in section 4.2.3. The context must also provide some information about the current device, such as the maximum thread block size. On WebGPU these will always be the default limit value from the standard, unless more is explicitly requested. In the future, we could request the maximum available and use that value, but currently we always just report the default.

GPU Memory

Memory operations mostly involve delegating to the corresponding WebGPU functions. For example, allocating memory is simply creating a new buffer object. Copying memory from the host to the device and between buffers on the device also has straightforward WebGPU analogues. Copying from the device to the host is the most complicated. General memory buffers are created with the `STORAGE` use so they can be bound in kernels, but as described in section 3.1.3, this is incompatible with mapping them to be read on the host. As a result, we have to first create an intermediate buffer with the `MAP_READ` usage and issue an on-device copy, before we can then map this buffer and copy the data to the destination. The GPU abstraction layer also defines a separate function that can be used to copy individual scalar values instead of larger parts of buffers; here we use a single pre-allocated readback buffer since we have a guaranteed maximum size. A potential future improvement would be to also re-use the readback buffers for general copies, only allocating a new one when the existing one is too small. Some care has to be taken to avoid a very large readback buffer permanently occupying memory that might be needed elsewhere however. Mapping a buffer for reading also has the asynchronicity problem discussed in more detail later.

One final complication is that both buffer sizes and the size of copy operations must always be a multiple of 4 in WebGPU. We can simply round up buffer sizes, and the same works for common cases of copying entire buffers. When copying from the device to the host, we can simply round up all operations until the final copy from the mapped buffer to the destination, which is just an ordinary `memcpy` without the multiple-of-4 requirement. There remain two problematic cases: Host-to-device and device-to-device copies, where the size is not already a multiple of 4, and the target range is somewhere in the middle of the target buffer, not at the end of it. Rounding up here risks overwriting unrelated data.

One possible workaround would be to issue the copy with a size that is rounded *down*, and then dispatch a kernel that performs the remaining writes. Out of time constraints, we have for now instead implemented a simple check for the problematic cases that aborts the program instead of silently corrupting data, since they are not all that common.

Kernels

The most complex part of the WebGPU-specific runtime system is creating and dispatching kernels. The GPU abstraction layer defines two functions that will be called as appropriate: `gpu_create_kernel` and `gpu_launch_kernel`. Each kernel gets created once as part of runtime initialization and then launched potentially many times, as required by the program. We have discussed the details of what is required to be able to dispatch a kernel on WebGPU in section 3.1. What remains is how this maps to the expectations and requirements of Futhark in practice.

Some tasks can always be done as part of creating a kernel: We create the bind group layout, we create a uniform buffer that is used to hold the scalar parameters for each launch, and we create the pipeline layout. We also allocate entries for all of the pipeline-overridable constants and set those that are truly constant. This already hints at one of our limitations: Some of our pipeline-overridable constants are not actually fixed at kernel creation time, namely those used for dynamic block sizes and for shared memory allocation sizes. For kernels where none of these are present, we already create the pipeline when creating the kernel. Otherwise, we have to defer this until the kernel is launched. Some pseudocode to illustrate this process is included in Listing 4.3.

All this is where all the kernel metadata mentioned previously comes in. We start by finding the right `wgpu_kernel_info` instance. It contains information on the total size of the scalars buffer, how many binding slots the kernel uses and what indices are assigned to them, and which, if any, override constants are used for dynamic block sizes and shared memory sizes. As of the writing of this report, it also contains a list of all the override constants used by the kernel in general. This should not be required, as we should be able to simply set all override constants in the entire module, no matter which kernel they are intended for. However, Chrome/Dawn currently has a bug in its validation checks where it only allows override constants actually used by the kernel to be specified when creating a pipeline for it, so we have to collect this extra information and filter which constants are set. We have reported this bug[4] and the relevant W3C working group has confirmed that the spec behaviour as written is intended[5] and will be clarified. As a result, we expect to be able to remove this workaround in the near future.

The other half of the equation is actually launching a kernel. If the pipeline has not been created yet, this is the first step. Here we can now add values for the dynamic block size and shared memory size constant overrides. These are passed as arguments to the `gpu_launch_kernel` function. Its interface assumes a single total shared memory size, as for the other backends, so our code generation simply passes the size of the individual blocks as part of the generic argument array for arguments to the kernel itself.

Dealing with those is next. From the kernel metadata, we know how many arguments to expect, as well as which are scalars and which are memory buffers. We create a bind group, matching the layout created earlier, containing the memory buffers. For the scalars, we create a buffer containing all of them to copy into the uniform scalars buffer created previously. This has to match the memory layout of the generated WGSL struct. For this purpose, the kernel metadata also contains the offset and size of each scalar field, computed during kernel translation by implementing WGSL's struct layout algorithm. Once all of

[4] `https://issues.chromium.org/issues/338624452`
[5] `https://github.com/gpuweb/gpuweb/issues/4624`

this is in place, we can finally record a command buffer dispatching the kernel and submit it to the device.

4.2.3 Asyncify

In section 3.1 we have seen that some operations in WebGPU are asynchronous on the host side, returning JavaScript promises. On the web, it is important to avoid blocking script execution for too long, as this can hang the entire site, degrading the user experience, and so the WebGPU API currently forces developers to handle asynchronously completing promises here. When using Dawn or wgpu to target native applications, they provide extensions that make it straightforward to block until a promise completes, but as we are targeting the web, this is not an option for us.

In the generic WebGPU C header, the asynchronous functions take an extra callback parameter that is invoked once the result is available. This presents a problem, as the GPU abstraction layer we would like to provide an implemention for does not make any affordances for this and expects to be provided simple functions that return synchronously.

Some of the asynchronous operations are only invoked once when initializing, namely acquiring an adapter and a device. These could perhaps be worked around without requiring major modifications to the existing abstraction layer by special-casing initialization. However we also have two more problematic asynchronous operations: Mapping a buffer to be read (or written) on the host side, and waiting for work submitted to a queue to be complete. These can, and do frequently, appear at any point during the program's runtime, right in the middle of generated code, possibly multiple calls deep. Adapting the existing abstraction layer and code generation to deal with a callback-based approach here would be difficult.

Luckily, variants of this problem occur frequently when using Emscripten to write native code that interacts with web APIs, since many modern web APIs involve asynchronous operations that are easily handled using `async-await` syntax in JavaScript. As a result, Emscripten contains a compiler pass and associated APIs called *Asyncify* [5, 18] that makes this problem easier to deal with.

Asyncify instruments the code while compiling to support unwinding out of the Wasm VM back to the controlling JavaScript code and then rewinding back to where unwinding started once the JavaScript side re-invokes the Wasm VM, with the option of passing some result value back. This lets the JavaScript await an asynchronous operations, for example a network `fetch`, before returning control to the Wasm code.

However we cannot quite make use of the mechanism as described directly, because we do not have an asynchronous *JavaScript* API, but instead the callback-based C API, so there is not immediately something the JavaScript side could await before returning to Wasm. We investigated two approaches for working around this.

First, we tried directly using the JavaScript WebGPU API to start the asynchronous operations, resulting in a promise that could be awaited. The Emscripten-compiled Wasm invokes glue JavaScript code whenever interacting with the WebGPU API, so in principle the device and other API objects like buffers created from the Wasm side have valid corresponding JavaScript objects that we can use. And in fact, Emscripten provides library functions to facilitate passing such API objects between JavaScript and Wasm/native code. Unfortunately, the Emscripten-provided glue code that is used when WebGPU API calls happen from the Wasm side interferes with this idea. It implements its own layer of state and validation tracking, with the result that, for example, if a buffer is mapped using the JS WebGPU API, we cannot directly get the mapped memory on the Wasm side. It would likely be possible to work around this, for example by manually invoking parts of the glue code from hand-written JavaScript, or changing its interal state. However, the generated glue code is not part of Emscripten's stable interface and liable to change between versions,

```
CreateKernel(ctx, name):
  info = GetKernelInfo(name);
  kernel = {info: info};
  kernel.scalars = CreateBuffer(UNIFORM, info.scalarSize);
  bgls = new BindGroupLayoutEntry[info.numBindings + 1];
  bgls[0] = { binding: info.scalarBinding, type: UNIFORM };
  for (i < info.numBindings) {
    bgls[i+1] = { binding: info.bindings[i], type: STORAGE };
  }
  kernel.bgl = CreateBindGroupLayout(bgls);
  kernel.pipelineLayout = CreatePipelineLayout([kernel.bgl]);

  kernel.overrides = new Override[info.numOverrides];
  CopyConstantOverrides(kernel.override, ctx.overrides);

  kernel.staticPipeline =
    info.numDynamicBlockSizes == 0
    && info.numSharedMemorySizes == 0;

  if (kernel.staticPipeline) {
    kernel.pipeline = CreatePipeline(kernel.pipelineLayout,
      kernel.overrides, ctx.module, kernel.name);
  }

LaunchKernel(ctx, kernel, gridSizes, blockSizes, args):
  info = kernel.info;
  shmemArgs = args[..info.numSharedMemorySizes];
  scalarArgs = args[shmemArgs .. shmemArgs + info.numScalars];
  memArgs = args[scalarArgs..];

  scalars = malloc(info.scalarSize);
  CopyScalarArgs(scalars, scalarArgs, info.scalarOffsets);
  CopyToDevice(kernel.scalars, scalars);

  bges = new BindGroupEntry[info.numBindings + 1];
  bges[0] = { binding: info.scalarBinding, buffer: kernel.scalars };
  for (i < info.numBindings) {
    bges[i+1] = { binding: info.bindings[i], buffer: memArgs[i] };
  }
  bg = CreateBindGroup(bges);

  if (kernel.staticPipeline) {
    pipeline = kernel.pipeline;
  } else {
    overrides = kernel.overrides
      + MakeBlockSizeOverrides(info, blockSizes)
      + MakeSharedMemOverrides(info, shmemArgs);
    pipeline = CreatePipeline(kernel.pipelineLayout,
      overrides, ctx.module, kernel.name);
  }

  cmds = CreateCommandEncoder();
  cmds.SetPipeline(pipeline);
  cmds.SetBindGroup(bg);
  cmds.Dispatch(gridSizes);
  ctx.device.Submit(cmds.Encode());
```

Listing 4.3: Pseudo-code illustrating creating and launching kernels.

```
1
2  typedef struct wgpu_wait_info {
3    bool released;
4    void *result;
5  } wgpu_wait_info;
6
7  void wgpu_map_sync_callback(WGPUBufferMapAsyncStatus status,
8                              void *info_v) {
9    wgpu_wait_info *info = (wgpu_wait_info *)info_v;
10   *((WGPUBufferMapAsyncStatus *) info->result) = status;
11   info->released = true;
12 }
13
14 WGPUBufferMapAsyncStatus wgpu_map_buffer_sync(WGPUInstance instance,
15                                               WGPUBuffer buffer,
16                                               WGPUMapModeFlags mode,
17                                               size_t offset, size_t size) {
18   WGPUBufferMapAsyncStatus status;
19   wgpu_wait_info info = {
20     .released = false,
21     .result = (void *)&status,
22   };
23
24   wgpuBufferMapAsync(buffer, mode, offset, size,
25                      wgpu_map_sync_callback, (void *) &info);
26
27   while (!info.released) {
28     emscripten_sleep(0);
29   }
30
31   return status;
32 }
```

Listing 4.4: Synchronous wrapper for `wgpuBufferMapAsync`.

making this a fragile solution. Another option would be to perform even more operations directly in JavaScript and manually deal with things such as copying mapped data into the Wasm heap and back out, but this comes with a potential performance overhead and not insignifcant complexity and maintenance costs.

Instead, we implemented a different option. As part of the Asyncify API Emscripten provides an `emscripten_sleep` function. As can be gleaned from the name, this function pauses execution of the code that calls it for a given amount of time. Crucially however, it does this by using the Asyncify infrastructure to return back to JavaScript and yield control to the normal browser event loop until the sleep time has elapsed. This means that events are processed, and this includes the callbacks for our asynchronous operations being called.

Thus we can wrap all of the asynchronous operations in blocking functions in our C implementation that call the asynchronous functions and repeatedly call `emscripten_sleep` until the corresponding callback is involved and sets a flag to end the loop. The callback also writes the result of the operation into a location where the synchronous wrapper can access it after its loop. An example implementation of this pattern for mapping a buffer is included in Listing 4.4.

In some ways, this gets us the best of both worlds: From the Futhark runtime's perspective, running in the WASM VM, the calls become simple blocking function calls. At the same time, the browser event loops still gets to run, so we achieve this without actually freezing the site for the user. The main downside is the performance implications. The Asyncify-instrumented code is slower than the original would be, although given that most of the actual work is going to happen on the GPU, performance of the host code is not as big a concern for us. Additionally, when the actual time that must be waited is short, the overhead of unwinding out of the WASM code and almost immediately resuming it is likely significant. As long as the web platform offers no blocking options for these calls, it is unlikely this can be avoided.

4.3 JavaScript Interface

The generated C code compiled by Emscripten can in principle be used directly. Exported functions are accessible through an Emscripten-generated module object and many of them can be called as relatively normal JavaScript functions. However they are not very developer-friendly: One has to do entirely manual memory management, like that in C, accessing the Wasm heap through a set of `TypedArray` objects provided by Emscripten. Correctly creating a Futhark array from a simple JavaScript array (or `TypedArray`) involves several copies and steps that are not very obvious. Additionally, all functions that can potentially be asynchronous must be called with a special `ccall` wrapper provided by Emscripten to correctly handle the Asyncify hooks.

To make the compiled program easier to use, we also generate a plain JavaScript interface to it. The interface can be seen in practice in section 6.4, where we present a simple demo application that makes use of this interface to invoke a Futhark library calculating a Mandelbrot set image.

The interface is packaged into a single `FutharkModule` class. Once instantiated, it provides simple JavaScript functions for every entry point in the Futhark program, as well as some utility functions such as `malloc` and `free` wrappers.

The class also contains nested classes for every Futhark array type used by the program. This mirrors the C API, where a separate type is generated for every combination of array rank and element type that appears in the public API. The JavaScript wrapper around this class makes it easier to construct these arrays as well as get ordinary `TypedArray` objects containing the same data. The C API also contains functions for working with *opaque* types such as tuples and records. We have not yet implemented wrappers for them, though we expect it to be straightforward to add them.

For both the entry point wrapper functions and the array wrapper classes, we take advantage of a *manifest* file generated by the Futhark compiler. This is an existing feature where the compiler can generate a JSON file describing the C API generated for a Futhark program. We embed a copy of this JSON file in the generated JavaScript, and then generate the wrapper functions and classes based on information in the manifest at runtime, taking advantage of the dynamic nature of JavaScript. This reduces the amount of JavaScript code we have to generate in the compiler. Generating JavaScript currently happens by effectively performing string interpolation, as opposed to the fairly sophisticated infrastructure for generating C code, or even the simple AST module we have defined for WGSL. As a result, minimizing the amount of JavaScript we have to generate directly is generally preferable.

5 Testing

The Futhark compiler comes with a built-in testing tool for Futhark programs, `futhark test`. In its basic form, it reads test annotations in Futhark source files and then invokes the given entry points with the specified data and compares to the expected output. (There are more options, such as generating random data or invoking further code to generate the input data, but the details are not particularly relevant here.) In this chapter, we will describe how we supported running these tests with our backend from a technical standpoint. Discussion of testing and benchmarking results is part of chapter 6, where we evaluate the completeness of our backend.

There is a comprehensive test suite of such tests to validate that the compiler itself works as expected. As part of this project, we would like to support running these tests using the new WebGPU backend. Eventually, that is what we did, with more details in section 5.2. However this method of testing requires that the backend already works end-to-end, even if it does not necessarily have to support all Futhark features yet. Getting to that point is already a significant portion of the entire project, so we would like to be able to somehow test our in-progress backend while developing it. Chronologically, we started with the WGSL shader code generation and continued on to generating usable host code only when that was reasonably complete. Thus, we developed a solution to test intermediate progress on the shader generation, described in the next section.

5.1 Ad-hoc Shader Testing

It is impossible to run, and thus to test, shaders without host-side code to set up all the required objects like pipelines and bind groups as well as the input data. Implementing and/or generating all the required code for that is itself a major part of the project, so we would like to first test the generated shaders in isolation.

Without generated host code, we must write it by hand. Writing host code for the entire test suite would be infeasible, but we can take advantage of some common structures in the test suite to run a decent amount of tests without much variety in the host code. Specifically, there are many tests that take one or more input arrays and map a relatively simple function over all of them. As long as there is no nested parallelism inside the mapped function, this will compile into a single kernel with multiple input arrays and a single output array. Notably, this structure applies to (almost[1]) all of the `primitives` tests, testing correct implementation of the arithmetic, comparison, and logical operators, as well as built-in functions, for Futhark's built-in primitive types. An example of one of these simple tests is shown in Listing 5.1. Being able to run these tests immediately while working on generating the corresponding shader code was a very helpful tool and passing them gives us some confidence in the correctness of the generated shaders before continuing on to proper host code generation.

We created a new subcommand for the `futhark dev` command that is used for various tasks relevant for working on the Futhark compiler itself. It runs the compilation pipeline on the given input file up to and including generating the WGSL shader. The shader generation collects some extra information, which would normally be implicit in the generated host code, that describes the generated kernel interface, such as the names of pipeline-overridable constants, the binding indices of buffers, and the entry point name. Other properties of the

[1] Some tests, e.g. testing the absolute value function for unsigned integer types, get optimized to not involving a GPU kernel invocation at all. These are not supported by the ad-hoc testing setup.

```
-- Test comparison of i32 values.
--
-- ==
-- entry: lt
-- input { [0i32, 1i32, -1i32, 1i32, -2i32 ]
--         [0i32, 2i32, 1i32, -1i32, -1i32] }
-- output { [false, true, true, false, true] }


-- ==
-- entry: eq
-- input { [0i32, 1i32, -1i32, 1i32, -2i32 ]
--         [0i32, 2i32, 1i32, -1i32, -1i32] }
-- output { [true, false, false, false, false] }

-- ==
-- entry: lte
-- input { [0i32, 1i32, -1i32, 1i32, -2i32 ]
--         [0i32, 2i32, 1i32, -1i32, -1i32] }
-- output { [true, true, true, false, true] }


entry lt (x:[]i32) (y:[]i32)= map2 (<) x y
entry eq (x:[]i32) (y:[]i32)= map2 (==) x y
entry lte (x:[]i32) (y:[]i32)= map2 (<=) x y
```

Listing 5.1: `tests/primitive/i32_cmpop.fut`, testing `i32` comparison operators.

kernel interface are implicitly assumed based on the limited structure of tests supported, such as all input and output arrays having the same length and the only required scalar argument being that length. The subcommand then outputs a JavaScript file defining several global variables containing all this information and the shader source code in a structured format. It also parses the test annotations in the input file and converts them into corresponding JavaScript literals that are also included in the output. Here we have to be careful to convert 64-bit numbers to JavaScript `BigInt` literals instead of normal numbers.

We then wrote a JavaScript program / simple web page that can include these test specification files. It creates a shader module, a pipeline, as well as the required buffers and then iterates over the embedded test cases, sets the input buffer contents, and runs the kernel. The overall process is similar to the final host code implementation described earlier, but specialized to the kind of kernel we support here. The only flexibility is in the number of input buffers, as well as their sizes and element types. The resulting output is compared to the expected and a result is output on the web page. This enables us to, for example, run all of the primitive tests and get an overview over what passes and what does not, as a useful intermediate check before we have proper host code generation.

5.2 `futhark test` Support

While this intermediate testing solution was appropriate to check our work while the host code was not implemented yet, ultimately we want to support the entire test suite using the normal `futhark test` command. However, this is not as straightforward as it would be for most other backends, because `futhark test` needs to be able to run the compiled executable in order run the tests. For most backends, the output is a native executable file that can just be run, but in our case, the target platform is the web, which adds some complications.

We initially considered adding functionality for compiling to a native executable with the WebGPU backend using one of the native implementations of the WebGPU API mentioned before — Dawn and wgpu — instead of compiling using Emscripten. That would enable simply using `futhark test` without further work. However we ultimately decided that

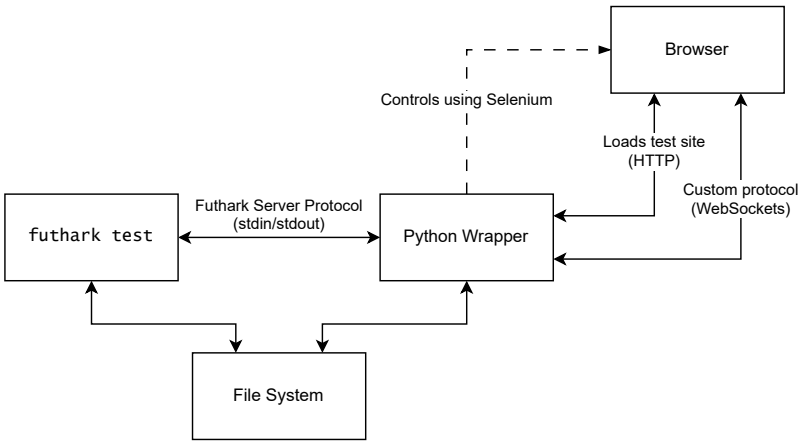

Figure 5.1: Architecture of our `futhark test` support.

since the main target for this backend is the web, the testing should ideally reflect that environment to avoid inconsistencies between the implementations.

Let us start by describing how `futhark test` interacts with the programs being tested. It first compiles them to a so-called *server* executable. Whereas the default executables produced by Futhark run a single entry point once with input and output happening via the standard IO streams, server executables allow several runs, potentially of different entry points, in a single execution. This is helpful because there is a not insignificant overhead associated with starting the program, most notably compiling the shaders/kernels that are generally embedded into the executable in source code form. Server executables are controlled using a simple line-oriented plain text protocol via the standard IO streams, with the server accepting commands on standard input and printing responses to standard output.

The server process maintains a set of variables that can be interacted with using the commands. The `store` and `restore` commands are used to write and read values to/from files at a given path respectively. They are stored in Futhark's own binary data format[2]. The `call` command is used to invoke an entry point with the arguments given in the form of previously-defined variables, and stores the results in the named output variables. There are some additional commands, such as for printing certain metadata about the program and available entry points, but the details are not all that important for our purposes.

`futhark test` supports specifying a separate *runner* program which it runs instead of attempting to run the compiled Futhark program directly. We have implemented a Python wrapper program that can be used as a runner for programs with the WebGPU program. It starts a local web server serving the compiled program as a simple web site and then uses Selenium [3] to start a browser and have it connect to the wrapper. The served website then contains an implementation of a custom protocol very similar to the normal Futhark server protocol that connects to the Python wrapper with a WebSocket connection. This overall structure is illustrated in Figure 5.1.

The Python wrapper mostly just relays the Futhark server protocol messages it receives. The protocol used between the wrapper and the server implementation running in the browser uses JSON messages instead of the simple line-oriented plaintext format of the Futhark server protocol to make it easier to work with in JavaScript and avoid any potential confusion on which part of the stack a message comes from. The only other major difference is that the `store` and `restore` commands contain the data to use directly in the message

[2] `https://futhark.readthedocs.io/en/stable/binary-data-format.html`

instead of containing file paths. This is necessary because, running inside a web browser, the JavaScript server implementation cannot open arbitrary local files. The Python wrapper is responsible for intercepting all `store` and `restore` commands and doing the required file I/O. It sends the raw file contents to the JavaScript implementation as a base64-encoded string.

In JavaScript, we used the interface our backend generates to implement the required protocol commands, including variable management and running entry points. We also attempt to catch any potential runtime errors and report them back to `futhark test` through the Python wrapper. While testing we have however discovered some cases where we trigger a browser bug that crashes the entire tab, which we cannot recover from in JavaScript. To deserialize incoming test data and serialize the resulting output when communicating with the wrapper, we have also created a JavaScript implementation of Futhark's binary data format for scalar and array values.

With this infrastructure in place, no modifications to `futhark test` itself are required. We can run the compiler test suite as normal, just adding an extra command line argument to use the Python wrapper as runner program.

5.3 `futhark bench` Support

Futhark also has a benchmarking tool called `futhark bench`. It uses the same Futhark server protocol as `futhark test`, with some additional commands, so we can use the same infrastructure for supporting it too.

The compiled server executable is responsible for measuring the runtime of running entry points and reporting it to the benchmarking tool. On the web, we use the *High Resolution Time* [17] web API, with the `performance.now()` function. Using it, time can be measured to a resolution of 100μs in general, although in so-called *isolated contexts*, a resolution of 5μs is available. We can turn our local site into an isolated context by ensuring the HTTP server portion of the Python wrapper includes specific headers when serving the site, giving us access to the improved resolution.

6 Evaluation & Limitations

Having described many things that we have implemented, in this chapter we will discuss some things that are not implemented in our backend. Some of these stem from limitations in WebGPU or WGSL, while others could be implemented given more time. A few limitations have already been mentioned in passing while discussing related features, and so will only be briefly mentioned here. We will also discuss what this means qualitatively in terms of what kinds of Futhark programs can already be expected to work properly, and what will not work. To finish, we will show a simple but complete example of a Futhark program being embedded in a web page and performing a non-trivial calculation on the GPU.

6.1 Limitations

There are still many things our backend does not support compared to the existing GPU backends. Some of these have a clear and relatively straightforward path towards resolving them, we simply lacked the time to implement all of those solutions. Overall, our focus was on attempting a relatively wide set of features expected from the backend rather than completing, say, every aspect of arithmetic and conversion for all primitive types before moving on. This approach is better suited to finding, and at least prototyping solutions for, as many challenges imposed by WebGPU and WGSL as possible in the limited time available.

6.1.1 Primitive Types and Operations

We have already discussed this topic to some extent in section 4.1. Futhark's `f64` type is not supported at all. The compiler will not introduce it as long as it is not present in the source program, so this is at least an easy-to-explain restriction. In principle it would be possible to implement 64-bit floating point operations entirely in software, in WGSL, and thus extend the backend to support the type, but this would be a significant undertaking and potentially signficantly slower than exceptions for primitive types.

Another limitation regarding floating point arithmetic in particular is the set of built-in functions, for example trigonometry functions or logarithms. WGSL provides some built-ins itself, but Futhark's standard library is more extensive. We have not made any attempt to manually implement the missing functions, so any Futhark program using them will not work. The same applies to some integer arithmetic, for example `mul_hi` which computes the high half of the product of two numbers. There are also some functions that *do* have WGSL equivalents, where we have simply not had the time to hook them up properly. These will at least be easy to add in the future.

Floating point arithmetic also has a more fundamental issue: WGSL explicitly deviates from the typical IEEE-754 floating point standard in a few ways [WGSL §14.6]. This includes some rounding differences, but the most salient issue is handling of overflow, infinities, and NaN (*not a number*) values. WGSL implementations may assume that all of these are never present at runtime, and permits any indeterminate final result in case they do appear during evaluation. Any expressions involving them that are evaluated before runtime (i.e. constant expressions at shader-creation time and override expressions at pipeline-creation time) are required to result in an explicit error. On the other side, Futhark provides explicit access to infinity and NaN values in its standard library. Futhark

- Primitive Types and Arithmetic:
  - `f64` type
  - `i64` division
  - Some built-in arithmetic and conversion functions
  - Full support for floating-point `NaN` and infinities
- In-kernel error handling
- The `SetMem` statement in `ImpCode`
- Atomics for types other than 32-bit integers
- Uniform control flow analysis violations, conflicts with atomic types in shared memory
- Memory Fences
- Built-in transpose and copy kernels

Figure 6.1: Summary of current limitations of our backend.

programs using them will compile to WGSL shaders that will either fail to compile or potentially return incorrect results at runtime. This includes relatively common operations such as `f32.minimum` which use infinities as neutral elements.

Lastly, aside from the built-in functions already mentioned, there are some more operations on primtive types we simply have not yet implemented. We have already mentioned the only partially-implemented 64-bit integer division. In addition, some conversion operations between primitive types are not implemented, such as boolean $\leftrightarrow$ float conversions and float $\rightarrow$ signed integer conversions. Once more, these should be straightforward to implement in the future, we just have not gotten to them.

6.1.2 Error Handling

Futhark is a safe programming language that produces runtime errors when programs perform erroneous operations such as accessing an array out of bounds. This is relatively straightforward to implement on the host, and here the WebGPU backend simply uses the existing code generation and works like all others. Such errors can also occur in code running on the GPU however. Here, a single thread detecting an error cannot generally abort the entire kernel dispatch, let alone with a descriptive error message. (CUDA does support aborting a kernel execution, but invalidates the entire driver context, making handling such a failure challenging.) Terminating only the thread detecting the error is also problematic, for example due to the uniform control flow requirements we have discussed in earlier chapters.

The compiler takes advantage of the restricted structure of the kernels it generates to implement an efficient on-device error handling approach [10]. Explaining it in detail is out of scope for this thesis, but it involves a global on-device failure flag that can be set and checked in kernels and a carefully designed mechanism that lets all threads in a kernel early-exit in a synchronized manner in case of an error, avoiding the any issues from non-uniformly executed barriers.

We have not implemented this error handling at all. The code generation for the existing GPU backends involves C `goto` statements which do not exist in WGSL, so a different method of achieving the same result would have to be found. Additionally, the code generation would have to ensure that the involved barriers are not just placed in uniform control flow, but so that WGSL's static analysis can conclude that they are in uniform control flow.

When such dynamic errors, such as out-of-bounds array accesses, occur without being caught by the discarded checks, WGSL specifies that a *dynamic error* occurs [WGSL §6.4.6]. This can have various effects including termination of the running kernel, an arbitrary other location the buffer being read or written, or the whole statement being discarded, among others.

6.1.3 Platform Limitations

In addition to what features specifically WebGPU and WGSL are missing, there are also some restrictions imposed by the current state of the web platform and our use of Wasm. As of writing this, Wasm in the browser is restricted to being a 32-bit target. Support for a 64-bit target is in progress, but not currently stably implemented by any major browser. This restricts the size of data that can be worked on to at most 4GB. In practice it will be less, as there is of course some memory taken up for the stack and various data other than the actual program input. Additionally, Emscripten currently defaults to a maximum memory size of 2GB due to limited browser support for more. In the future, we could support an option to use a larger maximum size when compiling the program.

Violating this limit unfortunately results in a very bad user experience at the moment. In our testing, when using the `futhark test` support described earlier with a test case that is too large, the entire browser tab crashes in Chrome. This currently not handled well by the test runner because the WebSocket connection is not shut down gracefully, so the `futhark test` invocation just hangs. In the future, some detection method for such cases could likely be added to the runner.

6.1.4 Pointers

In kernels, our backend generally expects a memory name to refer to a module-scoped variable declaration with some kind of array type. This mostly works fine, but is different to the other GPU backends, where such a name always refers to a (mutable) *pointer* to memory. We have seen that pointer types in WGSL are fairly restricted compared to those in C and most other languages, notably the lack of mutable pointer variables.

The main consequence of this difference is our complete lack of support for the `SetMem` statement in kernels. This is intended to change which buffer a name refers to, essentially equivalent to simple assignment of pointer-typed variables. As an example, this can be generated when the source Futhark program has an `if` condition that evaluates to an array. There are also some cases where the compiler generates kernels that internally use a form of double-buffering, which also uses `SetMem`.

Workarounds for this could be implemented. For example, since there is no ability to allocate entirely new memory buffers from kernel code, the set of available buffers is ultimately static. Instead of real WGSL array or pointer types, we could simply track what buffer a name refers to using an index value, and generate code that accesses the correct variable depending on its value at runtime. Alternatively, `SetMem` statements could be translated to copies of the entire buffer, though this would be even slower.

6.1.5 Miscellaneous

For completeness, we will again mention our limited support for atomic operations, as well as barriers and memory fences. This was explored in much more detail in section 4.1.

On the host code side, the backend is expected to report some device capabilities that WebGPU does not expose, like L2 cache size or the amount of threads running in lockstep (*warp size*). We can generally provide safe hardcoded values for these that will not impact correctness, though programs may be less efficient as a result. In the future, we could employ some heuristics to guess more accurate values based on information such as device

manufacturer and name, but these may not be provided by all browsers out of privacy concerns.

And finally, we are missing the built-in kernels that the compiler expects backends to provide. While most kernels are generated in the compiler customized to the specific program being compiled, some are very generic and instead hand-written and included in all programs. Specifically, there is a variety of kernels for transposing multi-dimensional arrays in different scenarios, and kernels for performing on-device copies that perform more complicated indexing than a simple contiguous `memcpy`. We see no reason it would be particularly challening to implement WGSL versions of these by hand, but have simply not had the time to do so.

6.2 Evaluation

Unfortunately some of these limitations have meant that we were not able to systematically run the entire compiler test suite despite the `futhark test` support described in the previous section. This is because some of the more exotic failure modes are not easily detected by the test runner (such as the entire browser tab crashing), and would have either required more engineering effort (and time) to deal with robustly, or manually going through the test suite and excluding all problematic tests.

As a result, this section is based on some hand-picked example tests, along with reasoning about how the described limitations affect various features. Overall, while the list of limitations is not exactly short we can nevertheless run a fair amount of Futhark programs. Our testing has mostly focused on relatively small self-contained programs. The larger and the more complex a Futhark program is, the more likely it is, that it will not work using our current backend.

The most straightforward problems are those around the primitive types and missing operations relating to them. Any Futhark program explicitly using something we have described above as not support will not work, but that is all. All the arithmetic and similar introduced by the compiler itself, that is not present in the source program, is supported by our backend and will not lead to unexpected errors.

More interesting are how the other limitations affect generated the kernels generated by Futhark. Very simple patterns such as programs only using the `map` SOAC (and its variants) and perhaps some built-ins like `replicate` and `iota` should generally work fine. Experimentally, we can also successfully run *some* programs involving more complicated SOACs such as reductions, scans, and even Futharks' generalized histogram function, `reduce_by_index`. On the other hand, not *all* such programs work, for example due to the missing built-in kernels or a host-issued copy operation that trips the rounding check. Some uses of these functions, as well as nested combinations of them, generate kernels we cannot compile, for example due to unsupported atomic operations or using a single buffer at multiple types. It is not at all easy to predict which programs will work and which will not, as things like these are not readily apparent from the Futhark source code.

There is also an unfortunate mix of of potential failure modes. Some Futhark input programs simply cause an internal compiler error when they are not supported. Others produce output containing a WGSL shader that will result in compiler errors when initializing the Futhark runtime. Yet other programs only produce errors at runtime, potentially depending on the input data.

A potentially worse outcome is a program that appears to run without errors but produces incorrect results. This is especially due to our lack of support for memory fences and in-kernel error checking and handling. Our backend currently simply discards fences instead of raising a compiler error for kernels containing them. This increases the number of programs that we can run, but at the cost of possibly incorrect results for them. In our

Test	Size	WebGPU	CUDA
`map_i32`	100	5662	123
	1000	5619	120
	10000	5499	292
`reduce_i32`	100	5662	123
	1000	5619	120
	10000	5499	292
`scan_i32`	100	5662	123
	1000	5619	120
	10000	5499	292

Table 6.1: Benchmark Results. Benchmarking was performed using the `futhark bench` tool on a Windows machine with an AMD Ryzen 5 2600 6-core CPU and a NVIDIA GTX 1660 Ti GPU. For the WebGPU benchmarks, Google Chrome 125.0.6522.112 was used. Runtime is reported in microseconds.

testing so far, results even for kernels that should contain fences are often correct, but ultimately this is non-deterministic and dependent on factors such as hardware and the underlying GPU API in use. In a similar vein, refusing to compile programs that should have in-kernel error checking would reduce support for programs a lot, so we just discard them. This is fine for Futhark programs that do not perform any erroneous operations, but for any that do, our backend will return indeterminate results instead of an error message.

As a result, our backend is certainly not production-ready. With care, it does make it possible to embed at least some Futhark programs in a web page, reliably taking advantage of hardware acceleration for compute-heavy workloads. Many of the current limitations can be resolved with further engineering effort on the backend, though some of them will require either modifications to the programs generated by the Futhark compiler, possibly with performance impacts, or waiting for WebGPU and WGSL to continue their evolution and gain more features. None of these changes should require significant re-engineering of the current backend, but instead simply extend it with more functionality.

In their current state, WebGPU and WGSL are less capable than existing targets like OpenCL and CUDA. Nevertheless, we were able to implement a significant part of a full Futhark backend on top of them. For many of the remaining issues, there is a straightforward path to resolving them even with WebGPU as it is now. Others would require further adjustments in parts of the Futhark compiler other than the backend, to ensure it generates output that is implementable on WebGPU. In some cases, this would likely mean generating less efficient kernels than currently for the other GPU backends. Some limitations are unlikely to be fixed without further evolution in WebGPU or WGSL, such as the lack of support for 64-bit floating point numbers or the general differences in floating point semantics.

6.3 Performance

We have performed very little benchmarking of the WebGPU backend, focusing instead on correctness and supporting as many features as we could in the time given. The Futhark compiler has a set of benchmarks designed to allow comparing performance with other implementations of the same parallel algorithms. We were unable to test these properly in time. Many of them cannot run at all under our backend due to its limitations, though some should be supported already. However Table 6.1 does contain results for three very simple benchmarks, effectively micro-benchmarks of individual SOACs. The corresponding Futhark file is included in Listing 6.1.

```
-- ==
-- entry: map_i32
-- random input { i32 [100]i32 }
-- random input { i32 [1000]i32 }
-- random input { i32 [10000]i32 }

-- ==
-- entry: reduce_i32 scan_i32
-- random input { [100]i32 }
-- random input { [1000]i32 }
-- random input { [10000]i32 }

entry map_i32 [n] (x: i32) (xs: [n]i32): [n]i32 = map (+x) xs
entry reduce_i32 [n] (xs: [n]i32): i32 = reduce (+) 0 xs
entry scan_i32 [n] (xs: [n]i32): [n]i32 = scan (+) 0 xs
```

Listing 6.1: Futhark program used for benchmarking.

Even apart from the simple nature of the programs under test, the benchmarking is very limited, most noticeably in terms of dataset size. Unfortunately we were not able to reliably benchmark with more data due to some of the issues related to loading too much data. We hope to resolve these at least to some extent soon, enabling us to acquire more interesting numbers.

We can draw some very limited conclusions even from these benchmarks. Most obviously, even the very small datasets are significantly slower than using the CUDA backend. However the runtime does not increase much with more data. This indicates a high constant overhead, likely from the mixture of JavaScript and Wasm host code, including switching back and forth between them. At the same time, the data size is too small to make real conclusions about how runtime would scale with more data. This is in part due to the massively parallel nature of GPUs, where a decent amount of data is required in order to actually exhaust the hardware's resources. We can also see this in the CUDA results, where runtime for the two smaller datasets does not vary consistently with dataset size at all.

6.4 Demo

As a relatively simple practical end-to-end example of how a website could make use of an embedded Futhark program, we have created a demo site. It renders an image of the classic Mandelbrot fractal into a canvas on the page. The Futhark program calculates the entire image and some JavaScript glue code is responsible for invoking the Futhark program and copying the result into the canvas. A screenshot of the result can be seen in Figure 6.2.

We have also made this demo available on the web[1]. It will only work on a browser with WebGPU support. As of writing, this includes recent Google Chrome (on Windows, Mac, and Android) and Microsoft Edge (on Windows) versions without any special configuration. Support on other browsers and platforms is frequently still gated behind experimental features.[2]

The page itself is a very simple HTML page referencing two scripts: The JavaScript output from running the Futhark compiler with the WebGPU backend, and the JavaScript glue code presented below. The page also contains a `canvas` element to display the result graphically.

The JavaScript glue is shown in Listing 6.2. It demonstrates the JavaScript interface we described in section 4.3. Initialization consists of loading the Emscripten-provided `Module` object (which is brought into scope by including the JavaScript file output by the Futhark compiler into the same HTML page) and creating the `FutharkModule` from it.

[1] `https://s-paarmann.de/futhark-webgpu-demo/`
[2] See `https://caniuse.com/webgpu` for up-to-date information.

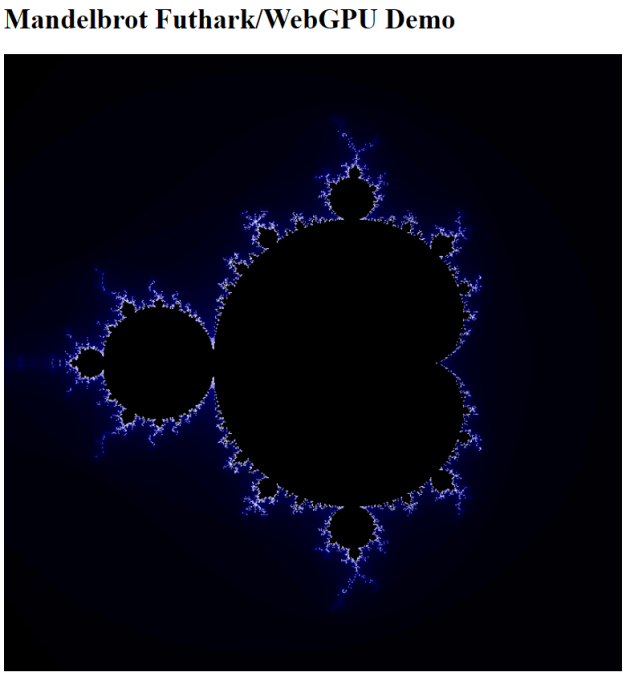

Figure 6.2: Screenshot of the Mandelbrot demo page.

```
async function run() {
  const m = await Module();
  const fut = new FutharkModule();
  await fut.init(m);

  const canvas = document.getElementById("canvas");
  const ctx = canvas.getContext("2d");

  const width = canvas.width;
  const height = canvas.height;
  const max_depth = 50;
  const [xmin, xmax] = [-1.7, 1.1];
  const [ymin, ymax] = [-1.4, 1.4];

  const [buf] = await fut.entry.main(width, height, max_depth, xmin, ymin,
    xmax, ymax);
  const vals = await buf.values();

  const colors = new Uint8Array(vals.buffer, vals.byteOffset, vals.length *
    4);
  setColors(ctx, colors, width, height);

  buf.free();
}

function setColors(ctx, colors, width, height) {
  const imgData = ctx.createImageData(width, height);
  imgData.data.set(colors);
  ctx.putImageData(imgData, 0, 0);
}

window.addEventListener("load", run);
```

Listing 6.2: JavaScript glue code for the Mandelbrot demo.

```
-- Adapted from futhark/tests/rosettacode/mandelbrot.fut

type complex = (f32, f32)

def dot ((r, i): complex): f32 =
  r * r + i * i

def multComplex ((a, b): complex) ((c, d): complex): complex =
  (a*c - b * d, a*d + b * c)

def addComplex ((a, b): complex) ((c, d): complex): complex =
  (a + c, b + d)

def divergence (depth: i32) (c0: complex): i32 =
  (loop (c, i) = (c0, 0) while i < depth && dot(c) < 4.0 do
     (addComplex c0 (multComplex c c),
      i + 1)).1

def rgba (r: u8) (b: u8) (g: u8) (a: u8): i32 =
  (i32.u8 r) | ((i32.u8 b) << 8) | ((i32.u8 g) << 16) | ((i32.u8 a) << 24)

def color (depth: i32) (div: i32): i32 =
  if div >= depth
     then rgba 0 0 0 255
     else
       let quot = f32.i32 div / f32.i32 depth
       let value = u8.f32 (255 * quot)
       in if quot > 0.5 then rgba value value 255 255
                        else rgba 0 0 value 255

def main (screenX: i64) (screenY: i64) (depth: i32)
         (xmin: f32) (ymin: f32) (xmax: f32) (ymax: f32): [screenY][screenX
    ]i32 =
  let sizex = xmax - xmin
  let sizey = ymax - ymin
  in map (\y: [screenX]i32  ->
            map  (\x: i32  ->
                    let c0 = (xmin + (f32.i64 x * sizex) / f32.i64 screenX,
                              ymin + (f32.i64 y * sizey) / f32.i64 screenY)
                    in color depth (divergence depth c0))
                 (iota screenX))
         (iota screenY)
```

Listing 6.3: Futhark code for the Mandelbrot demo.

From there, using the Futhark library is as simple as calling the `entry.main` method. It receives and returns ordinary JavaScript objects. The returned `buf` value is an instance of a `FutharkArray` type for the specific kind of array returned by the `main` entry point. Calling `values` on it lets us retrieve a copy of the result as an ordinary `TypedArray`, in this case a `Int32Array`. Then all that remains is setting that color data into the canvas.

For completeness, the corresponding Futhark program is also included in Listing 6.3. It is very ordinary Futhark, adapted from an example from the compiler's test suite to calculate color values in addition to the divergence values.

7 Conclusion

In this thesis project we have implemented a new backend for the Futhark compiler, targeting JavaScript, WebAssembly and the WebGPU API in the browser. This involved generating shader code to run on the GPU in the WGSL programming language, writing and generating the required host-side code to interface with the WebGPU API in C, and generating a JavaScript interface that is more friendly to use than the compiled C interface. We also implemented initial support for using Futhark's built-in testing and benchmarking tools, automatically controlling a web browser to run the compiled programs.

WebGPU and WGSL in their current forms are more restrictive than the APIs used by the existing Futhark GPU backends, OpenCL, CUDA, and HIP. As a result, we have developed workarounds for a variety of missing features and WebGPU/WGSL-specific restrictions. They are also structurally different from those APIs in several ways, requiring extra complexity in code generation or the host code implementation. Overall we have found WebGPU a viable compilation target, although some of the more advanced high-performance kernels generated by Futhark for other backend cannot yet be implemented efficiently. WebGPU is still a relatively young API and under active development, so it is likely that, over time, some of these limitations can be lifted.

Owing in part to these restrictions and in part to our limited time, the backend we developed is not yet capable of compiling all Futhark programs. Many structurally simple programs work, including basic uses of Futhark's `map`, `reduce`, and `scan` second-order array combinators, and even simple (generalized) histograms. For more complex, potentially nested, combinations of these operators, the Futhark compiler often generates efficient kernels that our backend cannot yet compile successfully.

Thus, more work is required to turn our backend into a reliable option for compiling all Futhark programs. This includes a mixture of relatively straightforward engineering effort on the backend, and potentially some changes to other parts of the compiler to generate simpler, but likely less performant kernels for the WebGPU backend. The future evolution of WebGPU and WGSL will hopefully alleviate the need for such changes, and is in some cases required for reasonably efficient implementation of some Futhark programs. Nevertheless, overall our backend shows the feasibility of making GPU-accelerated high-level functional parallel programming using Futhark available in the browser.

Bibliography

[1] The Dawn & Tint Authors. *Dawn, a WebGPU implementation*. URL: https://dawn.googlesource.com/dawn/ (visited on 2024-05-28).

[2] Alan Baker, Mehmet Oguz Derin, and David Neto. *WebGPU Shading Language, W3C Working Draft*. Draft as of 23 May 2024. May 2024. URL: https://www.w3.org/TR/2024/WD-WGSL-20240523/.

[3] Software Freedom Conservancy. *Selenium*. URL: https://www.selenium.dev (visited on 2024-05-28).

[4] Emscripten Contributors. *Emscripten*. URL: https://emscripten.org/ (visited on 2024-05-28).

[5] Emscripten Contributors. *Emscripten: Asynchronous Code*. URL: https://emscripten.org/docs/porting/asyncify.html (visited on 2024-05-28).

[6] NVIDIA Corporation. *CUDA C++ Programming Guide v12.5*. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/ (visited on 2024-05-30).

[7] "WebGPU native" developers. *webgpu-headers*. URL: https://github.com/webgpu-native/webgpu-headers (visited on 2024-05-28).

[8] The gfx-rs developers. *wgpu*. URL: https://wgpu.rs/ (visited on 2024-05-28).

[9] Khronos OpenCL Working Group. *The OpenCL Specification, Version V3.0.16*. Apr. 2024. URL: https://registry.khronos.org/OpenCL/specs/3.0-unified/html/OpenCL_API.html (visited on 2024-05-30).

[10] Troels Henriksen. "Bounds Checking on GPU". In: *International Journal of Parallel Programming* (Mar. 2021). DOI: 10.1007/s10766-021-00703-4.

[11] Troels Henriksen. "Design and Implementation of the Futhark Programming Language". PhD thesis. Universitetsparken 5, 2100 København: University of Copenhagen, Nov. 2017.

[12] Troels Henriksen et al. "Futhark: Purely Functional GPU-programming with Nested Parallelism and In-place Array Updates". In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2017. Barcelona, Spain: ACM, 2017, pp. 556–571. ISBN: 978-1-4503-4988-8. DOI: 10.1145/3062341.3062354. URL: http://doi.acm.org/10.1145/3062341.3062354.

[13] Dean Jackson and Jeff Gilbert. *WebGL Specification, Editor's Draft*. Feb. 2024. URL: https://registry.khronos.org/webgl/specs/latest/1.0/.

[14] Philip Rajani Lassen. "WebAssembly Backends for Futhark". MA thesis. University of Copenhagen, July 2021.

[15] Philip Munksgaard et al. "Dataset Sensitive Autotuning of Multi-Versioned Code based on Monotonic Properties". In: TFP 2021. 2021.

[16] Kai Ninomiya, Brandon Jones, and Jim Blandy. *WebGPU, W3C Working Draft*. Draft as of 24 May 2024. URL: https://www.w3.org/TR/2024/WD-webgpu-20240524/.

[17] Yoav Weiss. *High Resolution Time, W3C Working Draft*. July 2023. URL: https://www.w3.org/TR/hr-time-3/.

[18] Alon Zakai. *Pause and Resume WebAssembly with Binaryen's Asyncify*. July 2019. URL: `https://kripken.github.io/blog/wasm/2019/07/16/asyncify.html` (visited on 2024-05-28).