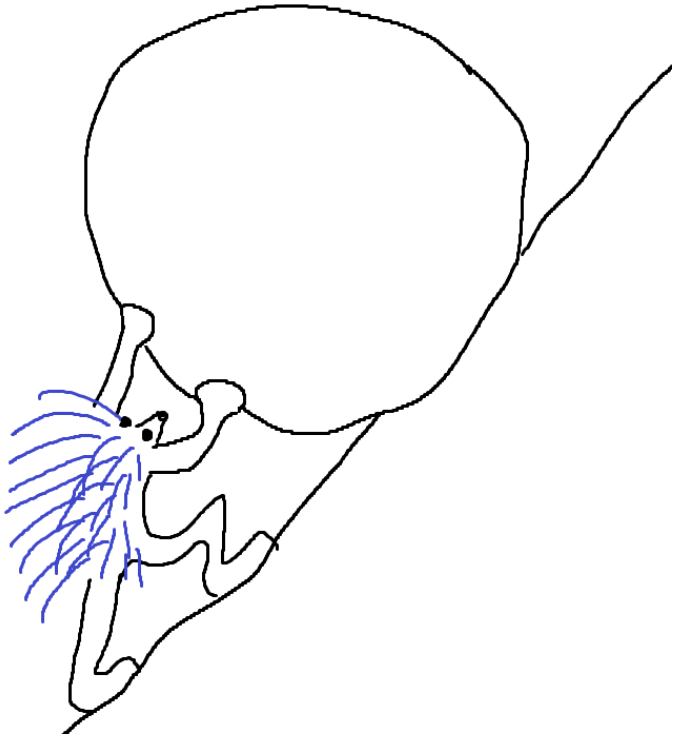
UNIVERSITY OF COPENHAGEN
FACULTY OF SCIENCE



BSc Thesis

Pushing the Boulder

Addressing key limitations of the Futhark WebGPU backend.

Caleb Andreasen
qpv948@alumni.ku.dk

Supervised by Troels Henriksen
Department of Computer Science

July 3, 2025

Abstract

Futhark is a data-parallel functional programming language with a compiler capable of generating efficient GPGPU code. Prior work by Sebastian Paarmann introduced an experimental WebGPU backend, enabling a small subset of Futhark programs to be run in web browsers. This project addresses some of the key remaining limitations of the WebGPU backend.

Using a test-driven development approach, we refine the backend by removing constraints and implementing missing functionality, such as built-in transpose and copy kernels, in-kernel function calls and copies, and support for special floating point values like NaNs and infinities. We show that, while supporting 64-bit atomics is impossible under the current WebGPU specifications, 8- and 16-bit atomics can be emulated despite their lack of native support.

Our testing demonstrates improved functionality across a range of representative programs, confirming the reliability of these improvements. Despite these advancements, major limitations remain, significantly constraining the WebGPU backend's practical usability for running larger high-performance data-parallel computations in web environments.

Contents

1 Introduction

Futhark [6] is a purely functional, data-parallel array programming language. It offers a machine-neutral programming model and an optimizing compiler able to generate both GPGPU code and multithreaded C code. A variety of fully functional GPU backends are supported, allowing Futhark programs to target OpenCL, Cuda and HIP. Previous work by Sebastian Paarmann [11] has recently expanded this list to include the new WebGPU API, allowing Futhark programs to run in a browser while still taking advantage of the GPU. While largely functional, major limitations of the WebGPU backend remain, greatly limiting its practical use. This project seeks to build on top of Paarmann's work, addressing some of the limitations of the WebGPU backend.

In Section 2, we give an introduction to the relevant background on GPU programming, WebGPU, the Futhark compiler and the existing WebGPU backend and its limitations. We also provide necessary background on memory models of relevance to our implementation work on atomics. Section 3 describes the implementation details of adding atomic operations to the WebGPU backend, implementing built-in kernels, and supporting in-kernel function calls and copy operations. We also give a quick overview of other minor bug-fixes and improvements made throughout this project, such as supporting infinite and `NaN` floats. In Section 4, we outline our testing and reevaluate the state of the WebGPU backend after our improvements. We also discuss what limitations remain and the feasibility of implementing them in future work.

2 Background

2.1 Implementation Approach

Since the starting point for the project was an already semi-functional compiler, we opted to take advantage of the huge suite of existing test programs and employ a test-driven development strategy. For each limitation we decided to tackle, we would identify a set of test programs that were failing to compile or returned wrong results and iteratively modify the WebGPU backend until the tests compiled and succeed. Through this process, we would often discover new issues and limitations not discovered in Paarmann's work

2.2 The GPU Programming Model

We begin by giving a high-level introduction the GPGPU programming model. The description provided in this section is based on the programming model defined by OpenCL and CUDA, as these are the APIs the Futhark compiler was originally designed around[1]. As such, much of the existing terminology in the compiler is borrowed from OpenCL and CUDA. Moreover, many of the Futhark compiler's internal assumptions on what capabilities are provided by the compilation target have been inherited from OpenCL. When we later introduce WebGPU in Section 2.3, we will see how it breaks some of these assumptions. For the remainder of this section, we will stick to CUDA terminology.

CUDA and OpenCL follow a heterogeneous programming model where the CPU coordinates execution and offloads parallel workloads to the GPU. The CPU handles sequential logic and manages memory, while the GPU accelerates compute-intensive tasks through massively parallel execution. Unlike CPUs, we cannot control the GPU directly, but instead have to program it indirectly through software layers like the CUDA runtime and GPU drivers. We refer to the CPU as the *host* and the GPU as the *device*. This heterogeneous model assumes that the host and device each maintain their own separate memory spaces referred to as *host memory* and *device memory* respectively [4, p. 17].

Programs run on the device are referred to as *kernels*. In both CUDA and OpenCL, kernels are written as functions in a dialect of C/C++ with some additional language extensions. To ensure predictable execution and efficient parallelism on the GPU, the CUDA/OpenCL C++ dialects impose some constraints on the programmer such as disallowing dynamic memory allocations [4, p. 269] and limiting recursion[2] [4, pp. 83, 437].

When the host dispatches work to the GPU, it specifies how many threads should execute and organizes them into a *grid* of *thread blocks* as illustrated in Figure 1. Each thread block consists of threads that are expected to execute together on a single streaming multiprocessor, and are thus able to synchronize efficiently and share data through *shared memory* [4, p. 12]. Blocks are required to execute independently, allowing them to be scheduled in any order across the available cores [4, p. 13].

Threads within a block can share data through *shared memory*, which is expected to be much faster than the (also much larger) *global memory* visible to all threads across blocks [4, p. 34]. Note that this does not include host threads. The memory is still located on the device, making accesses from the host very slow.

[1] Specifically OpenCL, with CUDA being added later by Jakob Stokholm Bertelsen as part of his BSc project [3].
[2] Kernels cannot be recursive. Recursive function calls are limited, as call stack cannot grow dynamically.

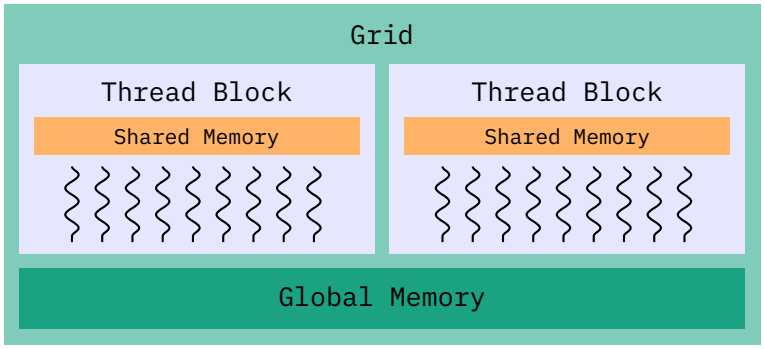


Figure 1: Threads are organized into thread blocks, which are organized in grids. Shared memory can only be accessed by threads within a block, while global memory can be accessed across blocks.

In CUDA and OpenCL, both shared and global memory is accessed through C-style pointers to untyped blocks of memory, allowing for arbitrary casting and pointer arithmetic. While the Futhark compiler restricts its usage of pointer arithmetic to indexing, it generally assumes C-like pointers that can be freely cast and treated as values. As we will see in the next section however, pointers are much more restrictive in WebGPU.

2.3 WebGPU

WebGPU is an API exposing the capabilities of GPU hardware to web browsers. Unlike CUDA or OpenCL, WebGPU is not a native API that interacts directly with the hardware drivers. Instead, WebGPU has been designed as a higher-level API that efficiently maps to modern native graphics APIs like Vulkan, DirectX 12 and Metal [10, §1]. A browser's WebGPU implementation essentially acts as an abstraction layer sitting between web apps and the underlying native GPU API.

While primarily targeted at 3D graphics workloads, WebGPU (unlike its predecessor WebGL) also provides first-class support for general-purpose GPU compute. As a web API, WebGPU has the unique design goal of maximising portability as one of its main focuses. The WebGPU standard should be implementable on top of a variety of native APIs and can thus only expose capabilities supported by all of them. This design focus has naturally led to an API that differs greatly from the CUDA/OpenCL programming model.

The differences already begin at the basic terminology. In WebGPU, kernels are referred to as *shaders* and thread blocks become *workgroups*. Table 1 lists a comparison of basic CUDA and WebGPU terminology. Throughout this report, we will switch freely between these terms depending on the context.

CUDA	WebGPU
Kernel	Shader
Thread	Invocation
Thread block	Workgroup
Global memory	Storage memory
Shared memory	Workgroup memory

Table 1: Comparison of basic CUDA and WebGPU terminology.

Owing to its roots as a graphics API, WebGPU embraces an explicit resource binding model heavily influenced by the design of native graphics APIs like Vulkan and DirectX 12. In WebGPU, shaders declare precise binding groups and indices for every resource it will access (e.g. buffers or textures). The host must then create matching bind groups that map actual GPU resources to these declared slots before dispatching workloads. This stands in stark contrast to the CUDA programming model, where resources are simply passed as kernel parameters. While this simpler model is nice for general-purpose compute workloads targeted by CUDA, the explicit approach taken by graphics APIs is preferable for 3D graphics applications, as it allows the GPU driver to minimize state transitions and resource usage in multi-stage rendering pipelines.

Instead of writing kernels in a C-dialect, WebGPU follows the tradition of graphics APIs like Vulkan and DirectX where shaders are written in a separate shading language like GLSL or HLSL. The shader code is then passed to the WebGPU API at runtime to create create a shader module, which in turn can be used to create a compute pipeline. Once this pipeline and its corresponding bindings have been bound, the host can dispatch a grid of workgroups to run the shader on the GPU. The workgroup dimensions are specified in the shader code, while grid dimensions are specified when dispatching from the host.

Instead of using an existing shading language, WebGPU opted to define it's own shading language called WGSL [2]. As we will see, the capabilities exposed by this language are greatly restricted compared to those of CUDA or OpenCL kernels.

2.3.1 WGSL

The WebGPU Shading Language (WGSL) is a simple statically typed imperative language used to express programs that run on the GPU. WGSL is used for writing shaders that will be executed in both rendering and compute pipelines. For this project, we are only concerned with the latter use case. As a simple example of a compute shader using most WGSL features of relevance for this project, consider the simple memcpy shader listed in Figure 2.

```
override block_size_x: i32;
override block_size_y: i32;
override block_size_z: i32;

struct Parameters {
    dst_offset: i32,
    src_offset: i32,
    n: i32
}

@group(0) @binding(0) var<uniform> args: Parameters;
@group(0) @binding(1) var<storage, read_write> src: array<u32>;
@group(0) @binding(2) var<storage, read_write> dst: array<u32>;
@compute @workgroup_size(block_size_x, block_size_y, block_size_z)
fn memmove(@builtin(global_invocation_id) global_id: vec3<u32>) {
    if (i32(global_id.x) >= args.n) { return; }

    let i: i32 = i32(global_id.x);
    dst[args.dst_offset + i] = src[args.src_offset + i];
}
```

Figure 2: Example of a WGSL compute shader, which performs a memcpy between two buffers

WGSL only supports basic types like booleans, 32-bit signed and unsigned integers, 32- and 16-bit floating point numbers, and user-defined structures. As a shading language, WGSL also has first-class support for composite types for 2, 3 and 4-dimensional vectors and matrices. Array types can either be fixed-size or runtime-sized. Special atomic types are provided for 32-bit integers, with no other types supporting atomic operations. Finally, WGSL has limited support for pointer types. These will be described in detail later.

WGSL has three kinds of value declarations for assigning names to immutable values: `const` declarations specify a name for a value known at shader module creation time [2, §7.2.1]. `override` declarations give a name to a pipeline-overridable constant, the value of which is specified when creating a pipeline [2, §7.2.2]. Finally `let` declarations are used for values that are fixed each time the let statement is executed at runtime [2, §7.2.3].

For mutable values, a variable declaration is used to create a name for a memory location. Unlike value declarations, variable declarations can thus be the basis of a pointer value, since they have an associated memory location [2, §7.1]. Variables also have an associated *address space* specifying its visibility. These are listed in Table 2.

Address Space	Visibility	Scope	Memory Space
`storage`	All threads in dispatch	Module	Global
`uniform`	All threads in dispatch	Module	Global
`workgroup`	Threads in the same workgroup	Module	Shared
`private`	Single thread of dispatch	Module	Local
`function`	Single thread of dispatch	Function	Local

Table 2: List of available address spaces for variable declarations. Variables in all but the `function` address space must be declared in module scope.

WGSL pointers are much more restrictive compared to its C counterpart: WGSL pointers are not storable, meaning they cannot be be assigned to a variable declaration. No arithmetic can be performed on a pointer and casting a pointer to a different type is not possible. Finally, there is no way to allocate or free memory from a heap. These restrictions are in place in order to guarantee that a pointer always points to the base address of a live object [2, p. 6.9.4].

Earlier we mentioned how array types can be either fixed-size or runtime-sized. However, only arrays in the storage address space are allowed to be runtime-sized. This means that `ptr<workgroup,array<i32>>` is not a valid pointer type, while `ptr<storage,array<i32>>` is. As WGSL doesn't allow generic functions, all pointer parameters for a function must have a fully specified pointer type, meaning separate functions must be implemented for working with values in different address spaces. Additionally, a function taking a pointer to an array in shared memory only works for a specific size, since `workgroup` arrays cannot be runtime-sized.

Finally, it should be noted that the limitations of WGSL compared to CUDA kernels do not stem from inherent hardware constraints, but are rather the result of higher-level design decisions aimed at ensuring broad compatibility across various platforms and graphics APIs. Ultimately, on a given device, both WGSL and CUDA kernels are compiled down to native code defined by the same equally capable GPU ISA. How the capabilities of an ISA are exposed to the programmer is an API- and language-level design decision. CUDA and WebGPU/WGSL have simply made different design decisions.

2.4 The Futhark Compiler

A complete description of the Futhark compiler is beyond the scope of this project, but we here give a quick overview. At a high level, the Futhark compiler has a fairly conventional architecture: A frontend parses and type checks programs before transforming them into an intermediate representation. This IR is then processed by a sequence of compiler passes in the middle-end, which ultimately outputs an optimized IR variant. The different IR variants mainly differ in how they express parallelism. For this project, the main IR of relevance is the `GPUMem` variant, which expresses parallelism through flat segmented operations designed to match the semantics of GPU kernels. This is the variant used by the CUDA, OpenCL and WebGPU backends.

In the backend, the immediate representation is translated to an imperative IR called `ImpCode`. Both kernel and host code are expressed as `ImpCode`, although with different extensions, such as a kernel launch operation for the host. In the CUDA/OpenCL backends, device-side `ImpCode` is then translated to their respective C variants after which the host-side `ImpCode` is translated to C code. Instead of generating host code for the GPU API directly, the compiler targets an internal GPU abstraction layer provided by a hand-written C runtime. This layer is shared between all GPU backends, with each GPU API providing its own implementation. The final output of the compiler is thus a single combined C program that can be compiled directly or used as a library.

2.4.1 The WebGPU backend

The WebGPU backend is a bit more involved. Since the internal C runtime provides an implementation of the GPU abstraction layer targeting the native WebGPU API, host-code can be translated to C much like the other GPU backends. However, device-side code has to be compiled to WGSL instead of C. This process involves translating `ImpCode` into an AST matching a subset of WGSL, which is then pretty-printed to actual WGSL code. As we will see in Section 2.4.2, this transformation is not without its pitfalls.

To run the resulting WebGPU C program in the browser, the C code is compiled into WebAssembly using Emscripten. To access the WASM module in the browser, the WebGPU backend also generates a JavaScript wrapper that exposes a nicer interface to the generated WASM. Websites can then run the compiled Futhark programs on the GPU through this JavaScript interface. For a more complete description of the WebGPU backend, we refer to Sebastian Paarmann's thesis [11].

2.4.2 ImpCode and WGSL

As we have already seen in Section 2.3.1, WGSL is in many ways more a limiting compilation target than the established CUDA and OpenCL targets. As a result, despite being a very simple imperative language, `ImpCode` makes some assumptions about the underlying capabilities exposed by the compilation target that do not match up with those provided by WGSL.

An example of this divergence in views is the handling of signedness. In `ImpCode`, integer types have no associated signedness. Instead, signedness is a property of individual operations. This differs from WGSL which uses different types for signed and unsigned values, with the semantics of operations like comparisons then depending on the type. The approach taken by the backend to handle this discrepancy was to always declare variables in WGSL as being signed, and then implement unsigned `ImpCode` operations like `umax_i32` in a WGSL prelude prepended to all generated WGSL kernels. This prelude is also used to emulate primitive types expected by `ImpCode`, such as 8- and 16-bit integers, that are not provided in WGSL.

```
alias i16 = i32;

fn norm_i16(a: i16) -> i32 {
  if (a & 0x8000) != 0 { return a | bitcast<i32>(0xffff0000u); }
  return a & 0x0000ffff;
}

fn add_i16(a: i16, b: i16) -> i16 {
  return norm_i16(a + b);
}

fn read_i16(buffer: ptr<storage, array<atomic<i16>>, read_write>, i: i32) -> i16 {
    let elem_idx = i / 2;       // Index of i32 containing i16 at index i
    let idx_in_elem = i % 2;    // Index of i16 inside the i32

    let v = atomicLoad(&((*buffer)[elem_idx]));
    return norm_i16(v >> bitcast<u32>(idx_in_elem * 16));
}
```

Figure 3: Examples of a helper functions defined in the WGSL prelude used to emulate the otherwise unsupported `i16` datatype. `i16` values are packed in `i32`s

The biggest difference in philosophies is how memory is viewed. In `ImpCode`, a memory variable first has to be declared with a `DeclareMem` statement. This is roughly equivalent to declaring a pointer in C – the variable initially doesn't point to any memory block. Note that unlike a C pointer, `ImpCode` memory variables also carry an associated memory space, such as global or shared memory. Before the declared block can be accessed, it first has to be the target of an `Allocate` or `SetMem` statement. An `Allocate` statement allocates a block of memory and assigns it to the target memory variable. A `SetMem` statement on the other hand, does not perform an allocation, but can instead be thought of as a simple pointer assignment: It simply changes which memory block a name refers to.

This view is fundamentally incompatible with how memory is handled in WGSL, where pointers are explicitly typed and defined as not being storable. In other words, WGSL does not allow reassigning pointers, preventing `SetMem` from being implemented properly. As we will see in the evaluation section (4), this is one of the major limitations preventing the backend from running larger, more complex, real-world Futhark programs.

2.4.3 Limitations of the WebGPU backend

While functional, there are still many features that lack proper support in the WebGPU backend, with the lack of `SetMem` statements mentioned earlier being a big one. Some of the other limitations identified by Paarmann in his thesis [11, pp. 35-36] are listed in Figure 4. We here give a more detailed description of the limitations of concern for this project.

- 64-bit floats and 64-bit division.
- Support for floating-point `NaN` and infinities.
- In-kernel error handling.
- Various `ImpCode` statements like `SetMem`, `Copy` and `DeclareMem` for `ScalarSpace`.
- Atomic operations for types other than 32-bit integers.
- Uniform control flow analysis violations conflicts with atomic types in shared memory.
- Memory fences.
- Various built-in transpose and copy kernels.

Figure 4: Summary of known limitations of the WebGPU backend at the beginning of the project. Adapted from Figure 6.1 of Paarmann's thesis [11]

A relatively simple limitation is the lack of support for floating-point `NaN` and infinities. This is a surprisingly big problem for the backend, as the compiler freely emits special floats for common operations like `f32.minimum`. Programs using these operations will fail to compile or potentially return incorrect results at runtime [11, p. 36]. This is a limitation inherited from WGSL, which deviates from the IEEE-754 floating-point standard. We describe our workarounds in Section 3.5.

Another big limitation is the lack of atomic operations for types other than 32-bit integers. This limitation is carried over from WGSL, which itself only has very limited support for atomics. Their absence causes a lot of Futhark programs to break, since the Futhark compiler relies on atomics for common operations like histogram computation. In Section 3.1, we describe how simulating most of the atomic operations used by Futhark can be accomplished using the limited atomics supported by WGSL. The necessary background on memory models and atomics for understanding the details of our implementation is provided in Section 2.5.

The lack of built-in transpose and copy kernels also causes issues in many Futhark programs. These are hand-written generic kernels that the compiler expects backends to provide. They were only left out of Paarmann's work due to time limitations [11, p. 38], so there is no technical reason for why they cannot be ported to WGSL. We describe their implementation in Section 3.2.

While some of the missing `ImpCode` statements like `SetMem` would require a major refactor of the backend to emulate [11, p. 37], we describe the implementation of in-kernel function call statements in Section 3.3 and `Copy` statements in Section 3.4. Adding support for the copy statement fixed some optimization passes performed by the compiler that previously would trip up the WebGPU backend.

Finally, we briefly briefly describe our implementation of `DeclareMem` statements targeting `ScalarSpace` in section 3.6. This section also contains a brief summary of other small additions, improvements and bug-fixes for the backend that were implemented during this project, but which did not warrant an entire section by themselves. They were nonetheless critical in improving the stability of the WebGPU backend.

2.5 Memory Models

In order to reason about correct multithreaded execution, the semantics of shared memory[3] must be clearly defined through an unambiguous memory model [1, p. 91]. Fundamentally, a memory model just defines the set of values a load operation in a program may return. Unlike for single-threaded execution, there is no immediately obvious answer to this and there may be multiple correct behaviours [9, Ch. 3]. When must a memory operation by one thread be visible to another thread?

The memory model defines an interface between a program and the underlying executing hardware. Compilers rely on the memory model defined by the instruction set architecture (ISA) of a processor to perform optimizations on programs that do not change their behaviour. Likewise, a programmer must be familiar with the memory model defined by their high-level language of choice to write correct multithreaded programs. We here give a quick introduction to some common memory models as well as how to reason about them.

2.5.1 Strong Memory Models

When a program is executed on a single processor, a programmer can safely rely on the fact that the result of the execution will be as if all operations had been executed sequentially in program order. The most natural way to view the execution of a multithreaded program might then be to view multithreaded execution as simply interleaving the steps of the sequentially executing threads. This is the *sequential consistency* (SC) model introduced by Lamport [7].

Consider the following simple example listed in Table 3. We assume all variables are initialized to zero.

Thread 1	Thread 2
`store x 1`	`store y 1`
`load r1 y`	`load r2 x`

Table 3: What values are allowed in `r1` and `r2` after execution?

Executing this program under a SC memory model, the only possible final states for `(r1,r2)` are `(1,0)`, `(0,1)` and `(1,1)`. An outcome of `(0,0)` is not possible, since it would one of the loads to be memory ordered before a store. This is not possible under SC, since such a re-ordering would violate the `store`-then-`load` program order in the example.

In practice however, SC doesn't play nice with hardware optimizations like write buffers and has thus been abandoned by vendors over performance concerns [9, Ch. 4]. Instead, ISAs like x86 adopted the *total store order* (TSO) model, which loosens the constraint that stores be ordered before loads. This has the consequence that `(0,0)` is a valid final state for `(r1,r2)` after executing Table 3 under TSO. If this execution is undesirable, the programmer will have to insert a *memory fence* between the stores and loads.

A memory fence specifies that all memory operations program-ordered before the fence must be ordered before any memory operations program-ordered after the fence. Inserting fences between the stores and loads thus prevents the loads and store from being reordered.

SC and TSO are both known as strong memory models, since they (mostly) respect the per-thread program order of memory operations. However, by relaxing ordering constraints to only the orders specified by programmers, weak memory models seek to further improve performance by enabling more hardware and software optimizations [9, Ch. 5].

[3] In this context, shared memory simply refers to any memory accessed by multiple threads.

2.5.2 Weak Memory Models

Consider the simple example program listed in Table 4 (adapted from [9, Ch. 5]). Line numbers are listed in the first column, and we again assume all variables are initialized to zero. The intention behind this program is clear: Thread 2 should wait for thread 1 to signal that it has updated x and y before they are loaded in thread 2. And this is indeed what happens when executed under SC and TSO: Both require that the x and y stores are ordered before the flag update. Likewise, the flag load in thread 2 must be ordered before the x and y loads.

	Thread 1	Thread 2
1	`store x 1`	`load r1 flag`
2	`store y 1`	`if (r1 != 1) goto 1`
3	`store flag 1`	`load r2 x`
4		`load r3 y`

Table 4: What orderings are needed for correct execution?

But SC and TSO also enforce some unnecessary orderings in this program: The x store must be ordered before the y store and the x load must be ordered before the y load. However, correct execution of this program only requires that the x and y stores are ordered before the flag store, and that the loads are ordered after the flag load. If we weakened our memory model to get rid of these excess ordering requirements, implementations could be freed up to employ more aggressive optimizations, increasing performance [9, Ch. 5].

In a *relaxed memory model*, loads and stores are unordered unless they operate on the same address. If order is needed, a fence has to be used. Disregarding all orderings like this initially seems to break all intuition one might have about the program, but it is the model favoured by GPUs as well as CPU ISAs like ARM and RISC-V. It is now perfectly legal for `r2` and `r3` to end up containing zeros. To ensure correct execution, we have to insert a fence after the `x`, `y` stores and above the `x`, `y` loads[4].

	Thread 1	Thread 2
1	`store x 1`	`load r1 flag`
2	`store y 1`	`if (r1 != 1) goto 1`
3	`fence`	`fence`
4	`store flag 1`	`load r2 x`
5		`load r3 y`

Table 5: Correct implementation of Table 4's program under a relaxed memory model.

In practice, relaxed memory models like the one used in ARM use *release consistency* (RC). Under RC, normal memory operations are still unordered, but we now introduce acquire and release operations as well. For a release operation, we require that any preceding loads/stores must be visible before the release, while an *acquire* operation must be visible before any succeeding loads or stores. Essentially, a release operations introduces a preceding fence while an acquire operation introduces a succeeding fence [9, Ch. 5].

[4] One might think that only adding the fence between stores in thread 1 is sufficient. This is not the case, however. Under relaxed memory ordering, there is no reason why the x and y loads in thread 2 should be ordered after the flag load. A compiler would for example be perfectly within its right to reorder the loads above the flag check.

2.5.3 Memory Models in GPUs

While many of the memory model concepts introduced so far also apply to GPUs, there are significant architectural differences between GPUs and CPUs that slightly complicates the memory model. The main difference is the hierarchical organization of execution units in GPUs. Traditionally, GPUs were narrowly designed for accelerating embarrassingly parallel graphics workloads that only required little data sharing and synchronization. Whereas CPU threads are "equal", the threads of a GPU are grouped into workgroups, which in turn are grouped into kernels. Early GPUs therefore only supported a bulk-synchronous programming model where threads within a workgroup could synchronize with execution barriers, but global memory communication between workgroups of a kernel was disallowed due to the lack of a rigorous memory model [8, p. 2].

To expose this hierarchical organization of threads to the programmer, memory models for GPUs introduced the notion of *scope*. In the Vulkan memory model (which is also the one used by WebGPU), the scope of an atomic operation is defined as the *"sets of shader invocations that must obey the requested ordering and atomicity rules of the operation"* [5, Appendix B]. In other words, which sets of threads have to synchronize with the operation?

The idea of scoping synchronization operations arises from the simple observation that, in most cases, a thread only needs to be synchronized with the neighbouring threads in its workgroup [8, p. 3]. We would be wasting a lot of performance if GPU fences always had to synchronize with all other threads executing on the device. By allowing software to scope its synchronization operations, hardware can avoid unnecessarily propagating synchronization mechanisms beyond the boundaries of the desired scope [8, p. 3].

Scope	Synchronized Invocations
`Device`	All threads executed on the device.
`QueueFamily`	All threads executed by queues in a given queue family.
`Workgroup`	All threads of a workgroup.

Table 6: Selected memory scopes defined by Vulkan [5, §9.25]

In WGSL, atomic operations are scoped based on the address space of the pointer we are accessing. The scopes used by WGSL are defined in the Vulkan spec (see Table 6). For pointers in the `workgroup` address space, the `Workgroup` scope is used, while the `QueueFamily` scope is used for pointers in the `storage` address space. For this project, we are only concerned with kernels executing on the same `compute` queue, so the distinction between `Device` and `QueueFamily` is not important.

3 The WebGPU Futhark Backend

3.1 Atomics

The Futhark compiler makes use of atomic operations on all integer types and supports atomic floating point addition. This complicates translating `ImpCode` to WGSL, as only atomic types for 32-bit signed and unsigned integers are supported on WGSL [2, §6.2.8].

3.1.1 Atomic 8- and 16-bit operations

Since smaller integer types like `i8` and `i16` already are represented by full `i32`s in the WebGPU backend, we can simulate atomic operations on them using by just updating the 8/16 bits of an `atomic<i32>` using a compare-and-swap loop on the full `i32`.

However, while `i8` and `i16`s are represented as `i32`s, the actual memory layout of `i8` and `i16` values differs depending on the address space. For storage buffers (global memory), we want to keep the memory layout of buffers coherent with the layout on the host, since they are often copied directly between host and gpu. As such, `i8` and `i16`s are packed into `i32`s, such that the lower and upper 16 bits of a `i32` are used for two different `i16`s, and similarly for `i8`s.

This means that, when performing our compare-and-swap (CAS) loop on an `i32`, we have to make sure we are only modifying the correct 8 or 16 bits. When the compiler wants to perform an atomic operation on the `i`th 8-bit value of a buffer, we thus have to first index the `i32` buffer at `i / 4` and then index the `i32` value as an array of four `i8`s at `i % 4`. Expressions for obtaining a pointer to the appropriate `i32` and calculating the offset within the `i32` are generated by the compiler and then used as arguments for the actual atomic operation. As an example of what the implementation of these atomic operations looks like in WGSL, see Figure 5 with our implementation of atomic addition of an `i8` value.

```
atomic_add_i8_global(p: ptr<storage, atomic<i32>, read_write>, offset: i32, x: i8) {
    let shift = bitcast<u32>(offset * 8); // i8 is stored in bits [shift : shift+8]
    loop {
        let old = atomicLoad(p);

        // Extract i8 from the i32, perform operation, and shift back up
        let val = i32(add_i8(norm_i8(old >> shift), x)) << shift;
        // Clear the old bits for the i8 in the i32, keep the rest
        let rest = old & ~(0xff << shift);

        // OR the updated value with the remaining unmodified bits and CAS
        if (atomicCompareExchangeWeak(p, old, val | rest).exchanged) {
            return norm_i8(old >> shift);
        }
    }
}
```

Figure 5: WGSL implementation of atomic addition of an `i8` using a CAS loop.

Since shared memory buffers are never copied directly to or from the host, this packing is not done in the workgroup address space, where `i8` and `i16` just take up a full `i32` each[5].

[5]While this wastes a lot of memory on padding, it means that we do not have to mark `i8` and `i16` buffers as atomic if they are never accessed atomically. At the time of Paarmann's thesis, this was necessary for generating `workgroupUniformLoads`, as they did not work on atomic values [11, pp. 22-23]. This was recently changed, however, so they now work with atomic values as well [2, §17.11.4]. See `https://github.com/gpuweb/gpuweb/pull/5141`.

Atomic operations on values in shared memory are thus much simpler to implement, as we do not have to bother maintaining the other bits of the `i32`, but can just swap the entire `i32` directly:

```
atomic_add_i8_shared(p: ptr<workgroup, atomic<i32>>, x: i8) -> i8 {
    var old = atomicLoad(p);
    while (!atomicCompareExchangeWeak(p, old, add_i8(norm_i8(old), x)).exchanged) {
        old = atomicLoad(p);
    }

    return norm_i8(old);
}
```

3.1.2 Floating-point operations

A similar approach can be taken for handling atomic operations on floats, where we can combine `bitcast<f32>`'s with atomic exchanges to simulate atomic operations. This approach was already used as a fall-back option in the CUDA and OpenCL backends to emulate atomic floating point operations on platforms without native support.

```
fn atomic_fadd_f32_shared(p: ptr<workgroup, atomic<i32>>, x: f32) -> f32 {
    var old: f32 = x;
    var ret: f32;
    while (old != 0.0f) {
        ret = bitcast<f32>(atomicExchange(p, 0)) + old;
        old = bitcast<f32>(atomicExchange(p, bitcast<i32>(ret)));
    }
    return ret;
}
```

For 16-bit floating point operations, we again have to pack them into `atomic<i32>` values like we did for `i16`. However, instead of extracting and updating values from an `i32` with bitwise operations, we can simply `bitcast` the `i32` to a `vec2<f16>` and index the `vec2` with the offset.

```
fn atomic_read_f16_global(p: ptr<storage, atomic<i32>, read_write>, offset: i32) {
    return bitcast<vec2<f16>>(atomicLoad(p))[offset];
}
```

To implement the atomic `fadd_16` operation, we pack the `f16` we want to operate on into a `vec2<f16>` at the specified offset and zero the other entry. This value can then be added directly to the old value stored at the specified address, allowing us to implement the `fadd` with the same basic CAS loop used for `i8` and `i16` operations.

```
fn atomic_fadd_f16_global(p: ptr<storage, atomic<i32>, read_write>,
                          offset: i32, x: f16) {
    var val = vec2<f16>(0.0); val[offset] = x;
    var old = bitcast<vec2<f16>>(atomicLoad(p));
    while (!atomicCompareExchangeWeak(p, bitcast<i32>(old), bitcast<i32>(old +
    ↪  val)).exchanged) {
        old = bitcast<vec2<f16>>(atomicLoad(p));
    }

    return old[offset];
}
```

A limitation of our implementation is that we now pad 16-bit floating point values in shared memory like we do for 16-bit integers. This limitation is a result of having to use the same function signature established by the other `atomic_*_shared` functions, which do not include an offset argument for indexing into the `atomic<i32>` value being operated on.

3.1.3 Atomic 64-bit operations

While adding limited support for 64-bit atomic operations in the future has been considered[6], it still looks years off. At the time of writing, simulating atomic operations for 64-bit values in WGSL is not possible given the current memory model and provided atomic operations. As an initial crude attempt at implementing 64-bit atomics, one might consider simply guarding `i64` values with a lock to simulate atomic operations:

```
while (!atomicCompareExchangeWeak(p_flag, 0, 1).exchanged); // Acquire lock
*p_val = add_i64(*p_val, v);                                // Store new i64
atomicStore(p_flag, 0);                                     // Release lock
```

However, this implementation is incorrect. In fact, it is impossible to write a correct spinlock in WGSL. The reason for this is that *all* atomic operations in WGSL use relaxed memory ordering. Quoting the WGSL specification:

> **[WGSL §17.8] Atomic Built-in Functions**
>
> All atomic built-in functions use a relaxed memory ordering. This means synchronization and ordering guarantees only apply among atomic operations acting on the same memory locations. No synchronization or ordering guarantees apply between atomic and non-atomic memory accesses, or between atomic accesses acting on different memory locations.

As such, it is impossible to establish any `happens-before` relationships[7]. If thread A acquires the lock, updates `p_val` and then releases the lock, it is perfectly legal for the write to not be visible to some other thread B that acquires the lock at a later time. In fact, the compiler would be completely within its rights to reorder the first two lines, such that the store operation is `program-ordered`[7] before the compare exchange.

In order to establish the necessary `happens-before` relationships, such that any writes to `p_val` *must* be visible to the next thread that acquires the lock, WGSL would have to expose support for release-acquire memory semantics:

```
while (!atomicCompareExchangeWeak(p_flag, 0, 1, Order::Acquire).exchanged);
*p_val = add_i64(*p_val, v);
atomicStore(p_flag, 0, Order::Release);
```

The release-acquire memory semantics are already well-defined in the Vulkan memory model [5, Appendix B] used by WebGPU: An atomic write with acquire semantics must not be reordered against any read or write that is program-ordered after it, while an atomic write with *release* semantics must not be reordered against any read or write that is program-ordered before it. Together, release and acquire operations can act as synchronization operations between different threads to guarantee that writes that happened *before* the release in one thread are visible *after* the acquire in another thread.

[6] `https://github.com/gpuweb/gpuweb/issues/5071`
[7] As defined in the Vulkan memory model [5, Appendix B], which is also the model followed by WGSL [2, §14.5]

How about using a memory fence then? CUDA provides memory fence functions for enforcing sequentially consistent ordering of memory accesses. For example, the `device`-scoped `__threadfence()` can be used to ensure that no writes to memory made by the calling thread *after* the call to `__threadfence()` are observed by any thread as occurring before any write to all memory made by the calling thread *before* the call to `__threadfence()` [4, §10.5]. Such a fence would allow us to implement the spinlock as follows:

```
1 while (!atomicCompareExchangeWeak(p_flag, 0, 1).exchanged);
2 threadfence();
3 *p_val = add_i64(*p_val, v);
4 threadfence();
5 atomicStore(p_flag, 0);
```

Unfortunately, no such memory fences exist in WGSL. As a last-ditch effort, one might consider using a full memory barrier instead of a fence. WGSL provides the `workgroupBarrier` and `storageBarrier` functions which use `AcquireRelease`[7] memory semantics [2, §14.5.4]. Unfortunately, these are of no use either, since the WGSL also mandates that control barriers only be executed in uniform control flow [2, §15.6.1]. Additionally, all control barriers use workgroup scope [2, §14.5.3], so they would not provide strong enough guarantees for simulating atomics on global memory.

3.1.4 Handling multiple signedness

As mentioned in Section 2.4.2, the WebGPU backend always declares variables as being signed, and then implements functions for performing operations with the proper signedness in the WGSL prelude. However, this was previously not done for atomic operations. Since only 32-bit atomics were supported before, native WGSL atomics like `atomicMax` were just used directly. To ensure the semantics of these atomic operations were correct, the backend would go through all uses of buffers to make sure they were always used at the same signedness and then declare the type of the buffer in that signedness. If multiple signednesses were used, an error would be raised and the Futhark program would fail to compile.

To lift this limitation, we now declare all buffers of atomic integers as signed. While this does mean that we have to emulate 32-bit unsigned atomic max/min operations using a CAS loop (see Figure 6), we consider it an acceptable trade-off, as many programs using unsigned min/max operations previously would completely fail to compile due to the mixed-signedness limitation.

3.2 Built-in Kernels

While most kernels are generated by the compiler custom to the specific program being compiled, some generic kernels for transpositions of multidimensional arrays and copies that require some non-contiguous indexing are instead hand-written and included in all programs.

In total, there are five different built-in kernels, each of which has four variations for operating on 1-, 2-, 4- and 8-byte data. In the CUDA/OpenCL backend, these variations are automatically generated by a big C macro. As WGSL does not have a preprocessor, we instead opted to implement the five kernels in separate WGSL files and then generate the different variations as part of the codegen of the WebGPU backend using basic search and replaces for element types and function names.

```
fn atomic_smax_i32_shared(p: ptr<workgroup, atomic<i32>>, x: i32) -> i32 {
    return atomicMax(p, x);
}

fn atomic_umax_i32_shared(p: ptr<workgroup, atomic<i32>>, x: i32) -> i32 {
    var old = atomicLoad(p);
    while (!atomicCompareExchangeWeak(p, old, umax_i32(old, x)).exchanged) {
        old = atomicLoad(p);
    }

    return old;
}
```

Figure 6: To handle usage of atomic types at multiple signedness, we declare all atomic values as signed, and then emulate unsigned operations with a CAS loop.

As an example of how these kernels were implemented in WGSL, the implementation of the built-in `map_transpose_small` kernel is listed in Figure 7. The parameter struct has been omitted for space reasons, but they should be self-explanatory. 64-bit arguments like buffer offsets are truncated to 32 bits. Occurrences of `NAME` are replaced with `1b`, `2b`, `4b` or `8b` with `ELEM_TYPE` correspondingly being replaced by `i8`, `i16`, `i32` or `i64`.

```
@group(0) @binding(0) var<uniform> args: MapTransposeParameters;
@group(0) @binding(1) var<storage, read_write> dst_mem: array<ELEM_TYPE>;
@group(0) @binding(2) var<storage, read_write> src_mem: array<ELEM_TYPE>;
@compute @workgroup_size(block_size_x, block_size_y, block_size_z)
fn map_transpose_NAME_small(@builtin(global_invocation_id) global_id: vec3<u32>) {
    let dst_offset = args.dst_offset[0];   // truncate i64 arguments to i32
    let src_offset = args.src_offset[0];
    let global_id_0 = i32(global_id.x);

    let our_array_offset = global_id_0 / (args.y_elems * args.x_elems) *
    ↪  args.y_elems * args.x_elems;
    let x_index = (global_id_0 % (args.y_elems * args.x_elems)) / args.y_elems;
    let y_index = global_id_0 % args.y_elems;

    let odata_offset = dst_offset + our_array_offset;
    let idata_offset = src_offset + our_array_offset;
    let index_in = y_index * args.x_elems + x_index;
    let index_out = x_index * args.y_elems + y_index;

    if (global_id_0 < args.x_elems * args.y_elems * args.num_arrays) {
        write_ELEM_TYPE(&dst_mem, odata_offset + index_out,
        ↪  read_ELEM_TYPE(&src_mem, idata_offset + index_in));
    }
}
```

Figure 7: Source code for the built-in `map_transpose_small` kernel.

The `write` and `read` functions called at the end of Figure 7 are part of the WGSL prelude that gets prepended to all WGSL kernels mentioned in Section 2.4.2. Since WGSL has no native support for non-32-bit integers, these helper functions are necessary to correctly access small 8- or 16-bit values from `dst_mem` and `src_mem` that are packed in 32-bit integer buffers.

3.2.1 The shared C runtime

One difference between WGSL and other backends is how pointers to global memory are passed to kernels. In CUDA/OpenCL, we can simply pass pointers to global memory like any other kernel parameter. This is not the case for WGSL, however, where we instead have to bind a storage buffer when launching a kernel. In the existing WebGPU backend, there was an assumption that these bindings always were listed as the last arguments in a kernel launch call on the host. This assumption is only guaranteed in the WebGPU codegen however, and did not exist in the other backends since the order in which pointers are passed to a CUDA/OpenCL kernel doesn't matter.

As such, the shared C runtime for CUDA/OpenCL/WebGPU didn't respect this assumption when launching built-in kernels, causing problems for the WebGPU implementation of the internal GPU abstraction layer. Since the ordering of arguments is of no consequence to the other backends, we made the design decision to enforce this assumption in the shared C runtime to accommodate WebGPU.

Another modification to the WebGPU C runtime was how the built-in kernels are compiled to shader modules. Currently, the WebGPU backend generates a single huge WGSL program containing all the generated kernels which is then compiled into a single shader module in the C runtime. When implementing the built-in kernels, we decided to output each built-in kernel in its own isolated WGSL program instead of appending them all to the main WGSL program. The WebGPU C runtime then generates a separate shader module for each built-in kernel.

3.3 In-kernel function calls

In general, the Futhark compiler aggressively inlines function calls in kernels. As such, their omission in Paarmann's original work doesn't cause any issues when running small programs. When the programs get larger however, or if they, like the example below, use the `#[noinline]` attribute, the compiler will emit function calls in the kernel ImpCode.

```
def f (x: f32) = (2*x, 4*x)
def main = map (\x -> let (a, b) = (#[noinline] f x) in a)
```

When translating ImpCode kernels to WGSL, the backend first runs some passes that generate any declarations that should be added before the compute shader itself (e.g. override constants). To implement function calls, we added a new pass that scans the kernel body for function calls and ensures any unknown functions are generated and declared before the kernel itself. This loosely follows the approach taken by the OpenCL backend.

Unlike WGSL, which only allows functions to return a single value, Futhark allows multiple return values. To handle this discrepancy, we used the same approach as the existing C-backend, where return value destinations are passed as pointer arguments to the function. The `f` function from the Futhark listing above is thus translated into the following WGSL function signature:

```
fn futrts_f_4659(x_5328: f32, prim_5404_out: ptr<function, f32>,
                              prim_5405_out: ptr<function, f32>)
```

Multi-destination ImpCode `Call` statements are then translated as follows:

```
var lifted_lambda_res_5398: f32;
var lifted_lambda_res_5399: f32;
futrts_f_4659(eta_p_5397, (&lifted_lambda_res_5398), (&lifted_lambda_res_5399));
```

Due to time constraints, we have only implemented support for primitive arguments and return values for in-kernel functions. Memory parameters are still not supported.

3.4 In-kernel copies

Another missing ImpCode statement is the `Copy` statement for in-kernel copies. A `Copy` specifies a type, the shape of the copy to perform, and the actual source and destination buffers, along with arguments for the offsets and strides to use when accessing them.

To translate these statements into WGSL, we construct nested loops based on the dimensionality of the shape, iterating over each dimension to compute the corresponding indices in the source and destination buffers. These indices are calculated as linearized addresses using the provided offsets and strides for each buffer. The indices are then used to perform a read from the source buffer and write to the destination buffer. A lightly adapted version with added comments of our implementation is listed in Figure 8.

```haskell
-- | Generate a WGSL.Stmt from an ImpCode Copy with type pt
genCopy pt shape (dst, dst_space) (dst_offset, dst_strides) (src, src_space)
  ↪ (src_offset, src_strides) = do
  -- Generate a WGSL expression for each shape dimension
  shape' <- mapM (genWGSLExp . untyped . unCount) shape

  -- For each iteration, calculate linearized address for source and destination
  -- and generate a WGSL read and write using these indices:
  body <- do
    -- Linearized address calculated by adding offset to dot product of the
    -- shape iterators with their respective strides:
    --  dst_i = dst_offset + i_0 * dst_stride[0] + i_1 * dst_stride[1] + ...
    let dst_i = dst_offset + sum (zipWith (*) is' dst_strides)
        src_i = src_offset + sum (zipWith (*) is' src_strides)
        read' = genReadExp pt src_space src src_i  -- src[src_i]
     in genArrayWrite pt dst_space dst dst_i read' -- dst[dst_i] = read'

  -- Generate nested loops based on the shape dimensionality around the element copy
  pure $ loops (zip iis shape') body
  where
    (zero, one) = (WGSL.VarExp "zero_i64", WGSL.VarExp "one_i64")
    is = map (VName "i") [0 .. length shape - 1]
    is' :: [Count Elements (TExp Int64)]
    is' = map (elements . le64) is
    iis = map nameToIdent is

    -- Given an array of identifiers and loop bounds, generate nested for loops,
    -- with a given WGSL statement as the innermost body:
    -- for (i_0 = 0; i < shape[0]; i++) {
    --   for (i_1 = 0; i < shape[1]; i++) {
    --     ...
    --       body;
    loops :: [(WGSL.Ident, WGSL.Exp)] -> WGSL.Stmt -> WGSL.Stmt
    loops [] body = body
    loops ((i, n) : ins) body =
      WGSL.For i zero
        (wgslCmpOp (CmpUlt Int64) (WGSL.VarExp i) n)
        (WGSL.Assign i $ wgslBinOp (Add Int64 OverflowWrap) (WGSL.VarExp i) one)
        (loops ins body)
```

Figure 8: Annotated implementation of generating a WGSL in-kernel copy.

3.5 Floating-point infinities and NaNs

WGSL floats generally follow the IEEE-754 standard for 32- and 16-bit floats with some small, but rather significant, deviations. The most important of these is how infinities and NaNs are handled. A WGSL implementation may assume that infinities and NaNs are not present during shader execution. If a runtime expression yields an infinity or NaN, "the result will be an indeterminate value of the target type." [2, §15.7.2] As a result, "some functions (e.g. min and max) may not return the expected result due to optimizations about the presence of NaNs and infinities." [2, §15.7.2]

This is problematic for Futhark which explicitly exposes infinities and NaNs in its standard library. Additionally, the compiler uses infinities as neutral elements for common operations like `f32.minimum`. Despite not being mandated by the WGSL spec however, NaNs and infinities seemed to work as expected whenever they were generated at runtime on the platforms we tested[8]. The main problem was thus how to assign a float variable to infinity or NaN when the WGSL spec mandates that "infinities and NaNs generated before shader execution will generate errors" [2, §15.7.2].

To get around this restriction, we bitcast a u32 literal with a bit pattern corresponding to an infinity or NaN. Directly bitcasting such a literal was caught by the WGSL compiler, so we have to wrap it in a helper function.

```
fn f32_inf_helper() -> u32 { return 0x7f800000u; }
fn f32_inf() -> f32 { return bitcast<f32>(f32_inf_helper()); }

fn f16_neg_inf_helper() -> u32 { return 0xfc00u; }
fn f16_neg_inf() -> f16 { return bitcast<vec2<f16>>(f16_neg_inf_helper())[0]; }
```

By adding functions for generating infinities and NaNs to the WGSL prelude, the WGSL codegen can simply assign a float to special value by calling one of these functions.

```
handleSpecialFloats :: T.Text -> Double -> WGSL.Exp
handleSpecialFloats s v
  | isInfinite v, v > 0 = WGSL.CallExp (s <> "_inf") []
  | isInfinite v, v < 0 = WGSL.CallExp (s <> "_neg_inf") []
  | isNaN v = WGSL.CallExp (s <> "_nan") []
  | otherwise = WGSL.FloatExp v
```

The Futhark compiler also expects the runtime of the compilation target to provide `isinf` and `isnan` functions for determining if a given float is infinite or a NaN. To implement this in WGSL, we bitcast the float to an unsigned integer, extract the exponent and mantissa bits, and determine whether it is infinite or a NaN according to the IEEE-754 rules for how the bits of a float are interpreted, as described in [2, §15.7.1]. For example, if a float is infinite, the exponent field will be all 1s while the mantissa is zero.

```
fn futrts_isinf32(x: f32) -> bool {
    let exponent = (bitcast<u32>(x) >> 23) & 0xFF;
    let mantissa = bitcast<u32>(x) & 0x7fffff;

    // If the exponent field is all 1s, and mantissa is zero, x is an infinity.
    return (exponent == 0xff) && (mantissa == 0);
}
```

[8] We tested in Chrome with a discrete NVIDIA 3050ti laptop GPU and an embedded AMD 680M GPU

Likewise, for a `NaN`, the exponent field will be all 1s while the mantissa is non-zero. To extract the bits of a 16-bit float, we again use the trick of packing it in a `vec2<f16>` before bitcasting to an `u32`.

```
fn futrts_isnan16(x: f16) -> bool {
    let bits = bitcast<u32>(vec2<f16>(x, 0.0));
    let exponent = (bits >> 10) & 0x1F;
    let mantissa = bits & 0x3ff;

    // If the exponent field is all 1s, and mantissa is nonzero, x is a NaN.
    return (exponent == 0x1f && mantissa != 0);
}
```

From our limited testing, these implementations work as expected when using infinities and `NaN`s in Futhark code, although we cannot guarantee that they are functional across WGSL implementations on different browsers and GPUs due to the lack of guarantees from the WGSL specification on how special floating point values are handled.

3.6 Other additions and bug-fixes

Throughout the project, we encountered various minor issues, some of which were not discussed in Paarmann's thesis [11]. While fixing these was an important step in improving the robustness of the WebGPU backend, they did not all warrant a section by themselves. We here summarize the issues we encountered, and how they were solved. For a complete list of all contributions to the backend from this project, see Figure 11 in the appendix.

Out-of-date WebGPU runtime At the beginning of the project, programs generated by the WebGPU backend failed to compile, as the shared C runtime for host code had been modified without updating the WebGPU implementation accordingly.

64-bit integer literals If a 64-bit integer literal is specified in `ImpCode`, the WebGPU backend will have to split it into two 32-bit values stored in a `vec2<i32>`. We found that the backend was only extracting the bottom 24 bits of the low and high 32-bits of 64-bit integer literals. This meant that a 64-bit `-1` literal was translated into a `vec2<i32>(0xffffff, 0xffffff)`, which when read as a signed 64-bit integer would be `72057589759737855`.

Mixed 32/64-bit operations Mixing 32- and 64-bit integers in binary operations is not supported in the generated WGSL runtime, but the compiler was attempting to generate them for histogram constructs. This was resolved by making the compiler sign-extend the 32-bit integer to 64-bits before operating on the them.

Multiline comments The WebGPU backend would sometimes generate multiline comments in WGSL code. However, the pretty-printer for outputting these comments to WGSL did not properly handle multiline comments, causing only the first line to be commented which resulted in shader compilation errors.

```
-- Before: Multi-line comments would spill out below the "//" line
pretty (Comment c) = "//" <+> pretty c
-- After: Each line of a comment gets "//" prepended.
pretty (Comment c) = vsep (map ("//" <+>) (pretty <$> T.lines c))
```

Negation The WebGPU backend was erroneously translating unary negations to the logical negation operator `!` for float and integer types. For floating-point types, we instead want to output a minus sign, while we for integer types have to call `neg_*` helper functions to handle negations of 8-, 16- and 64-bit integers not natively supported in WGSL.

Hanging tests While testing, we found that a lot of tests would hang when run. This was caused by the default max message size of the WebSocket opened by the webgpu test runner being very small, breaking tests taking anything beyond trivially-sized arrays as inputs. This was solved by increasing the max message size to 1 GiB.

Out-of-memory crashes when testing As our test inputs increased in size, however, we started running into out-of-memory crashes. The WebGPU test runner would previously read binary input data from a file, encode it to b64 string and send send it over a WebSocket. The JavaScript wrapper would then decode the string back to binary data. For large input data, this decoding step resulted in out-of-memory crashes that froze the browser. Instead of sending input data as b64 encoded strings, we now send a path to the binary test input file, allowing the JavaScript wrapper to fetch it directly without having to decode anything.

DeclareMem statements in ScalarSpace As mentioned in Section 2.4.3, the `DeclareMem` statement for `ScalarSpace` had not been implemented yet. In the Futhark compiler, `ScalarSpace` refers to a statically sized array of some primitive type in private memory on the GPU. To translate this into WGSL, we declare a statically sized array of the given type. The `ScalarSpace` type specifies the shape of the array as a list of constants. These are multiplied together into a big constant WGSL expression that is then used as the size parameter for the array.

```
1  -- | The ScalarSpace DeclareMem case in the main genWGSLStm function
2  genWGSLStm (DeclareMem name (ScalarSpace vs pt)) =
3    pure $ WGSL.DeclareVar
4             (nameToIdent name)
5             (WGSL.Array (wgslPrimType pt) (Just $ wgslProduct vs))
```

GPU copies within buffers The shared C runtime provides a `gpu_memcpy` function for copying blocks of memory around in device memory. To implement this function in WebGPU, the `copyBufferToBuffer` function was used. Unlike its CUDA and OpenCL equivalents, however, the WebGPU function does not allow the source and destination buffers to be the same [10, §13.4]. As this condition was not checked in the WebGPU backend's `gpu_memcpy` implementation, Futhark programs would simply crash if the compiler attempted to move blocks of memory around within a buffer with a `gpu_memcpy`.

This issue was another example of the disconnect between capabilities assumed by the existing compiler, and what WebGPU actually provides. As a temporary solution, we modified `gpu_memcpy` to allocate a temporary GPU buffer if the source and destination buffers are identical and then call `copyBufferToBuffer` twice. As an alternative solution that would avoid this allocation, we considered implementing a built-in `memmove` kernel that could be launched to perform the move instead. However, we did not have time to test out this approach.

4 Evaluation

4.1 Testing

The Futhark compiler comes with a large suite of test programs. To evaluate the robustness of the WebGPU backend, we ran some selected groups of tests that focus on different aspects of the compiler.

The `primitive` suite contains tests for primitive types, operations on them, and some built-in functions. The tests themselves are very simple, but they are useful for evaluating our coverage of basic built-in functions and types. To test codegen for more complicated compiler transformations, we also run the `noinline`, `intragroup` and `hist` test suites, which should be representative of the more challenging compiler transformations.

The `noinline` suite tests test our implementation of in-kernel function calls. The `soacs` suite tests that we can compile basic second-order array combinators like maps, scans and reduces correctly. The `hist` suite contains tests generating various histogram constructions, while `intragroup` tests further stress-test more complex compiler transformations.

Running these test suites, we get the results listed in Table 7. We list the total number of test cases, along with the number of passing and failing cases. We have split failing test cases into fail and error columns in order to distinguish between tests that compile, but exhibit incorrect behaviour (fail), and those who fail to compile (error).

Test Suite	Count	Pass	Fail	Error
`primitive`	489	359	11	119
`soacs`	81	57	11	13
`hist`	106	89	5	12
`intragroup`	74	57	5	12
`noinline`	15	11	2	2

Table 7: Test results for the Futhark WebGPU backend at the end of the project. Number of test cases (i.e. not test programs) are listed, with most programs having 2-10 test cases. Cases that fail to compile with a reported error are listed in the error column, while tests that compile but exhibit incorrect behaviour are listed in the fail column.

Overall we find that, while most tests pass, a large number of tests currently fail with a compiler error when using the WebGPU backend. This is especially prevalent in the `primitive` suite, where a lot of tests use unsupported features like 64-bit floats or 64-bit atomics. For some of the more complicated tests from the other suites, the lack of support for `SetMem` statements also causes issues. It should be noted, however, that the number of tests failing to compile is likely heavily inflated. The `primitive` suite especially contains lots of test programs with different cases for 16-, 32- and 64-bit floats. However, since the 64-bit case doesn't compile, all three cases of the program will be counted as failing to compile. We simply have not had enough time to properly filter out these erroneous test failures.

All suites also have a handful of tests that compile, but either return the wrong result or, more commonly, fail at runtime due to a WebGPU assertion or JavaScript exception. We now go over what exactly causes these failures.

4.1.1 Failure causes

If we exclude tests that fail to compile, there are only three test programs from the `primitive` suite that fail. All of these failures are caused by missing primitive operations that haven't been implemented yet. These are listed in Table 8.

Test	Failure mode	Reason
`mad_hi.fut`	32/64-bit test cases fail.	32/64-bit `mad_hi` not supported.
`mul_hi.fut`	32/64-bit test cases fail.	32/64-bit `mul_hi` not supported.
`u64_division.fut`	All test cases fail.	64-bit division not properly supported.

Table 8: Remaining failing `primitive` tests (excluding tests that fail to compile).

Failure causes for the remaining failing tests of the other suites are listed in Table 9. This list includes both runtime failures and compile-time errors. We find that all 11 of the failing `soacs` test cases are caused by an exception being raised in the JavaScript code generated by Emscripten. While we have not had time to properly investigate this issue, it seems to related to Emscripten's handling of async function calls in WASM, resulting in in a function expecting a `i64` parameter being called without any arguments.

Failure cause	Suite	Affected tests
Buffer used at different types	`hist`	`horizontal-fusion`
Dispatched grid too large	`intragroup`	`big0`
Memory parameters in GPU function	`noinline`	`noinline3`
No in-kernel error handling	`noinline`	`noinline2 & noinline4`
JavaScript BigInt conversion exception	`soacs`	All failing tests
	`hist`	`large & large2d`

Table 9: Failure causes of note for remaining failing tests.

The `intragroup/big0.fut` test generates host code that attempts to dispatch a number of workgroups that exceeds the maximum compute workgroups allowed per grid dimension by WebGPU. This is not caught as a compile error, but instead causes the test to crash at runtime. We also ran into this issue when running larger programs (see Table 10 of the following section) or attempting to benchmark large datasets.

Some tests also fail to compile with errors caused by limitation not discussed so far. The backend for example currently generates a compile error if a single memory buffer is used at different types. This causes the `hist/horizontal-fusion.fut` test to fail to compile.

While we have implemented in-kernel function calls, device functions still do no support memory parameters, causing the `noinline/noinline3.fut` test to fail to compile. The remaining failures from the `noinline` suite are caused by the lack of in-kernel error handling, as two of the `noinline` test cases expect the output to be a divide-by-zero error reported from the kernel. This feature is still not supported in the WebGPU backend.

4.1.2 Running larger programs

To test our implementation on some larger programs that more closely resemble real-world use-cases of Futhark, we attempted to run some of the programs from the `futhark-benchmarks` repository[9]. Examples of programs from this suite include implementations of algorithms like kmeans and pagerank, as well as programs like a mandelbrot renderer, kmeans clustering and a small fluid simulation. The results of running all the programs from the `accelerate` and `finpar` suites are listed in Table 10

Program	Compiles	Functional	Notes
`fluid`	yes	yes	Passes tests.
`hashcat`	yes	yes	Passes tests.
`kmeans`	yes	yes	Passes tests.
`mandelbrot`	yes	partially	Large tests crash: Grid size limit.
`tunnel`	yes	partially	Large tests crash: Grid size limit.
`LocVolCalib`	yes	partially	Large tests crash: Grid size limit.
`canny`	yes	no	Wrong results.
`nbody-bh`	yes	no	Number of storage buffers exceeds 10.
`pagerank`	yes	no	Number of storage buffers exceeds 10.
`crystal`	no	no	Buffer used at multiple types.
`fft`	no	no	Generates `SetMem` statements.
`nbody`	no	no	Generates `SetMem` statements.
`smoothlife`	no	no	Uses 64-bit atomics.

Table 10: Results of running larger `futhark-benchmark` programs.

When attempting to run these larger programs however, we ran into a new critical limitation of the WebGPU backend affecting the `nbody-bh` and `pagerank` programs. All WebGPU-enabled devices specify a set of limits, constraining its capabilities. These limits are validated by all API calls, and will cause an assertion if violated. The `maxStorageBuffersPerShaderStage` limit here proved to be problematic for the `nbody-bh` and `pagerank` test programs. By default, this limit only allows 8 storage buffers to be bound in a given kernel dispatch [10, §3.6.2]. While this default represents the minimum required limit for a WebGPU-compliant device, most devices will, in practice, support a higher threshold, which can be requested during WebGPU device creation.

However, while the device itself may support it, the actual WebGPU implementation might not. We found that on Dawn, the WebGPU implementation for Chromium, the highest limit supported is 10 storage buffers per shader stage. This is a somewhat arbitrary limit unique to Chromium[10]. On our devices, for example, Firefox allows requesting a limit of up to 64 storage buffers.

4.2 Benchmarking

To benchmark our implementation, we ran all the functional programs from Table 10 through `futhark-bench` and compared with timings with the CUDA backend. All results are listed in Table 11. We here find that the Futhark backend generally runs ∼4x slower than the CUDA backend.

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

[10] This restriction has existed since October 2023 and seems unlikely to change anytime soon, as raising it would require a more extensive refactor of the Dawn Metal backend: `https://issues.chromium.org/issues/366151398`

However, WebGPU notably beats the CUDA backend for large datasets in the `fluid` and `LocVolCalib` benchmarks, while still being slower or only matching CUDA for small inputs. These 2-6x performance gains for some programs were very surprising, but they remained consistent across multiple benchmark runs. We have not had time properly investigate these results.

		WebGPU		CUDA	
Benchmark	Case	Mean	95% CI	Mean	95% CI
LocVolCalib	small	405.10	[398.85, 410.03]	441.56	[439.66, 443.98]
	medium	451.40	[449.01, 453.69]	893.68	[890.47, 897.66]
`fluid`	medium	45.18	[40.93, 49.47]	9.54	[8.62, 10.48]
	large	349.09	[343.53, 358.35]	2204.43	[2199.84, 2207.83]
`hashcat`	rockyou	44.56	[43.15, 46.25]	12.36	[12.26, 12.47]
`kmeans`	trivial	28.68	[24.98, 33.63]	1.74	[1.65, 1.85]
	50000	310.86	[301.81, 319.79]	35.74	[32.74, 41.11]
	200000	394.17	[383.31, 409.18]	52.42	[49.04, 57.36]
`mandelbrot`	800	6.39	[5.83, 7.31]	0.79	[0.77, 0.80]
	1000	9.92	[6.97, 13.97]	1.17	[1.15, 1.18]
	2000	16.28	[15.41, 18.09]	3.11	[3.07, 3.17]
	4000	40.23	[38.33, 44.54]	11.16	[11.10, 11.22]
`tunnel`	800	7.43	[6.17, 8.95]	1.14	[1.13, 1.16]
	1000	11.37	[10.57, 13.03]	1.65	[1.63, 1.68]
	2000	27.26	[25.28, 30.10]	6.37	[6.29, 6.45]
	4000	87.04	[84.28, 90.93]	24.93	[24.20, 25.15]

Table 11: Benchmark results for selected `futhark-benchmarks` programs. The arithmetic mean runtime and 95% confidence intervals across 10 runs for each test case are listed. All timings are in milliseconds. Benchmarked on an NVIDIA 3050ti laptop GPU. WebGPU running in Chrome 137. CUDA running in WSL.

We also ran a microbenchmark targeting the built-in transpose kernels. By varying the shape of the input array, the compiler will generate calls to our different built-in transpose kernels. When using a small 4-by-4 array as input for example, the `map_transpose_small` kernel is launched. For the remaining kernels, we use inputs with 10 million entries corresponding to datasets of about 10, 20, 40 and 80 MiB for 1-, 2-, 4-, and 8-byte element sizes respectively. The source program for benchmarking 4-byte transpose kernels is listed in Figure 9. Identical input shapes and sizes were used for other element sizes.

```
1 -- entry: transpose_i32
2 -- random input { [3162][3162]i32 } auto output        -- default
3 -- random input { [5000000][2]i32 } auto output        -- low_width
4 -- random input { [2][5000000]i32 } auto output        -- low_height
5 -- random input { [4][4]i32 }       auto output        -- small
6 entry transpose_i32 (xs: [][]i32) = transpose
```

Figure 9: Excerpt from the Futhark program used for benchmarking built-in transpose kernels.

Running these tests on the WebGPU and CUDA backends with `futhark-bench`, we get the results listed in Table 12. From these results, we can draw a few conclusions. Looking at the reported runtimes for the `small` kernel, we see that WebGPU is an order of magnitude slower than CUDA, suggesting that the generated WebGPU programs suffer from much higher overhead than their CUDA equivalents. This mirrors the results found in Paarmann's thesis [11, p. 40]

When scaling up to larger datasets, however, this difference shrinks significantly, as seen in the `default`, `low_width` and `low_height` tests. Here, WebGPU is in some cases almost equivalent with CUDA when using 4-byte element sizes. For smaller 1- and 2-byte elements however, CUDA is significantly faster. This is likely due to the lack of native support for these smaller data types in WebGPU, forcing the WebGPU backend to emulate them with 32-bit atomics. This also explains why the 1- and 2-byte transposition kernels have near-identical runtimes, where whereas CUDA sees an almost 2x performance decrease when moving from 1- to 2-byte elements.

		WebGPU		CUDA	
Kernel	Type	Mean	95% CI	Mean	95% CI
`default`	1b	16.54	[15.86, 17.51]	4.51	[4.48, 4.55]
	2b	16.76	[15.89, 18.84]	6.28	[6.24, 6.32]
	4b	10.33	[9.95, 10.76]	10.51	[10.44, 10.60]
	8b	15.32	[13.84, 17.53]	22.85	[22.34, 23.30]
`low_width`	1b	13.34	[11.80, 16.57]	2.24	[2.23, 2.25]
	2b	14.38	[12.57, 16.91]	4.95	[4.93, 4.97]
	4b	13.69	[11.95, 16.38]	8.60	[8.56, 8.67]
	8b	11.99	[10.84, 13.82]	15.66	[15.50, 15.91]
`low_height`	1b	13.34	[11.90, 15.57]	2.43	[2.42, 2.45]
	2b	13.09	[11.74, 15.30]	4.95	[4.04, 4.07]
	4b	13.04	[11.49, 15.48]	7.73	[7.71, 7.78]
	8b	11.11	[10.31, 12.82]	15.40	[15.34, 15.48]
`small`	1b	8.46	[6.00, 13.96]	0.20	[0.19, 0.21]
	2b	7.93	[5.93, 11.01]	0.19	[0.18, 0.19]
	4b	7.25	[6.32, 9.00]	0.20	[0.19, 0.20]
	8b	7.38	[6.18, 9.92]	0.19	[0.18, 0.20]

Table 12: Benchmark results for the built-in `map_transpose` kernels. The arithmetic mean runtime and 95% confidence intervals across 10 runs for each test case are listed. All timings are in milliseconds. Benchmarked on an NVIDIA 3050ti laptop GPU. WebGPU running in Chrome 137. CUDA running in WSL.

Surprisingly, we also see that the WebGPU backend beats CUDA's runtime when using 8-byte elements. Like the previous results from Table 11 however, we have not had time to properly investigate the cause of this performance difference.

4.3 Remaining limitations

Finally, we summarize the remaining limitations discussed in this section Figure 10. This can be viewed as an updated version of the list from Figure 6.1 of Paarmann's thesis [11, p. 36].

- 64-bit floats.
- 64-bit atomic operations.
- The `SetMem` statement in `ImpCode`.
- Some primitive operations like 64-bit division and 32/64-bit `mul_hi` and `mad_hi`.
- Some large programs violate the `maxStorageBuffersPerShaderStage` limit.
- Some large programs violate the `maxComputeWorkgroupsPerDimension` limit.
- Memory fences.
- Small types still waste a lot of padding in shared memory.
- In-kernel error handling.
- Buffer re-use between multiple types causes a compilation error.
- Some programs raise JavaScript BigInt conversion exceptions.
- Memory parameters in GPU functions.

Figure 10: Summary of current limitations of the WebGPU backend.

4.4 Future Work

While the WebGPU backend at this point is robust enough to compile and run non-trivial Futhark programs, more work is needed to bring it to parity with the other existing backends. Beyond implementing missing primitive operations like 64-bit versions of `mul_hi`, `mad_hi` and `division`, future work would entail continuing to lift compiler restrictions specific to the WebGPU backend.

An example of such a restriction, is the backends inability to handle buffer reuses with multiple types. The Futhark compiler will sometimes re-use buffers within a kernel as a memory-saving optimization. If this results in a buffer being accessed at multiple types, the WebGPU backend will generate a compile error. Future work could look into disabling this optimization for the WebGPU backend if the buffer re-use cannot be done with the same type.

A lot of memory is currently being wasted on padding in shared memory when using 8- and 16-bit integers on the WebGPU backend. Part of Paarmann's reasoning for introducing this limitation, was the fact that the built-in `workgroupUniformLoad` WGSL function did not allow accessing atomic types. This restriction was recently lifted (April 2025)[11] however, and the WGSL spec now lists an overloaded version for pointers to atomic types [2, §17.11.4]. Future work could investigate if this could be exploited to allow packing small integer types in shared memory.

The most problematic restriction still enforced by the WebGPU compiler is the lack of support for `SetMem` statements. Investigating this problem was outside the scope of this project, but Paarmann's original thesis suggested a potential workaround involving using an index to track what buffer a name is referring to at runtime. The compiler would then generate code to access the correct variable at runtime depending on the index value [11, p. 37]. How best to solve this remains an open question.

[11] See `https://github.com/gpuweb/gpuweb/issues/5124`

Another big limitation of running Futhark programs on WebGPU is the max storage buffer count per kernel limit. While this is a problem with Chrome's WebGPU implementation, a potential workaround could be to make the Futhark compiler coalesce storage buffers into a few large buffers, and then associating each name referring to a storage buffer with an offset. The code generation for all buffer accesses would then have to be modified to take this offset into account.

Finally, future work could look into properly supporting different web browsers. For this project, we only concerned ourselves with getting Futhark programs to run in the Chrome browser. While WebGPU at the time of writing still isn't enabled by default in other browsers, Firefox has a functional implementation in Firefox Nightly builds,[12] and Apple recently (June 9) announced that WebGPU would be coming to their next release of Safari.[13] We briefly attempted to run some Futhark programs on Firefox, but found that their WGSL compiler has reserved the `i8`, `i16` and `i64` keywords that the Futhark WGSL runtime uses as type aliases.

5 Conclusion

In this project, we have continued work on the experimental WebGPU backend for the Futhark compiler, addressing key limitations and bringing it closer to feature parity with the existing backends. Our contributions include implementing support for atomic operations, built-in transpose and copy kernels, as well as implementing in-kernel copies and function calls. A complete list of our contributions can be found in Figure 11 of the appendix.

We show that with our improvements, the backend now passes the majority of compiler test cases evaluated in this project. Likewise, we find that the WebGPU backend is capable of compiling and correctly executing some of the more complex benchmarking programs – none of which compiled before our contributions. Major limitations still remain, however, and continued future work is needed if WebGPU is to reach parity with the existing OpenCL, CUDA and HIP backends.

At the start of this project, the WebGPU backend had fallen out of date with the rest of the Futhark compiler, rendering it unable to generate any functional programs. An important contribution of this project has thus been to simply keep the effort to add a fully-functional WebGPU backend alive. We hope that future work will continue this effort such that WebGPU eventually can become a viable compilation target for Futhark on par with the likes of OpenCL and CUDA.

[12] See `https://github.com/gpuweb/gpuweb/wiki/Implementation-Status`
[13] See `https://webkit.org/blog/16993/`

References

[1] Sarita V. Adve and Hans-J. Boehm. “Memory models: a case for rethinking parallel languages and hardware”. In: *Commun. ACM* 53.8 (Aug. 2010), pp. 90–101. DOI: 10.1145/1787234.1787255.

[2] Alan Baker, Mehmet Oguz Derin, and David Neto. *WebGPU Shading Language, W3C Candidate Recommendation Draft*. Draft as of 22 March 2025. URL: https://www.w3.org/TR/2025/CRD-WGSL-20250322/.

[3] Jakob Stokholm Bertelsen. “Implementing a CUDA Backend for Futhark”. BSc Thesis. University of Copenhagen, 2019. URL: https://futhark-lang.org/student-projects/jakob-bsc-thesis.pdf.

[4] NVIDIA Corporation. *CUDA C++ Programming Guide*. Release 12.8. URL: https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.

[5] Khronos Group. *Vulkan Specification*. Version 1.4.311. URL: https://registry.khronos.org/vulkan/specs/latest/html/vkspec.html/.

[6] Troels Henriksen. “Design and Implementation of the Futhark Programming Language”. PhD thesis. University of Copenhagen, 2017.

[7] Leslie Lamport. “How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs”. In: *IEEE Transactions on Computers* C-28.9 (1979), pp. 690–691. DOI: 10.1109/TC.1979.1675439.

[8] Daniel Lustig, Sameer Sahasrabuddhe, and Olivier Giroux. “A Formal Analysis of the NVIDIA PTX Memory Consistency Model”. In: *Proceedings of ASPLOS ’19*. ACM, 2019, pp. 257–270. DOI: 10.1145/3297858.3304043.

[9] Vijay Nagarajan, Daniel J. Sorin, Mark D. Hill, and David A. Wood. *A Primer on Memory Consistency and Cache Coherence*. 2nd ed. Springer Cham, 2020.

[10] Kai Ninomiya, Dzmitry Malyshau, and Justin Fan. *WebGPU, W3C Candidate Recommendation Draft*. Draft as of 30 April 2025. URL: https://www.w3.org/TR/2025/CRD-webgpu-20250430/.

[11] Sebastian Paarmann. “A WebGPU backend for Futhark”. MSc Thesis. University of Copenhagen, 2024. URL: https://futhark-lang.org/student-projects/sebastian-msc-thesis.pdf.

Appendix

Summary of Contributions

- Implemented atomic operations for non-i32 types.
- Implemented built-in transpose and copy kernels
- Implemented in-kernel copies.
- Implemented in-kernel function calls.
- Implemented the `DeclareMem` statement for `ScalarSpace`.
- Implemented support for special floating point infinities and `NaN`s.
- Removed limitation on mixing atomic operations with different signedness.
- Added workaround for performing `gpu_memcpy` within WebGPU buffers.
- Added support for requesting the max supported device limits when requesting device.
- Fixed a bug with negation of floats and integers.
- Fixed a bug with how 64-bit integer literals were generated.
- Fixed a bug with multiline comments.
- Fixed out-of-memory crashes and hangs when testing with large inputs and outputs.
- Fixed shader compilation errors for histograms, caused by mixing 32- and 64-bit integers.
- Updated the WebGPU C runtime to match recent changes to the shared runtime.
- Standardized parameter ordering of built-in kernels in the shared C runtime.

Figure 11: Summary of bug-fixes, improvements and added capabilities to the WebGPU backend contributed by this project.

Declaration of using generative AI tools

☐ I/we have used generative AI as an aid/tool.

☒ I/we have *not* used generative AI as an aid/tool.

List which GAI tools you have used and include the link to the platform:

Not applicable.

Describe how generative AI has been used in the exam paper:

Not applicable.