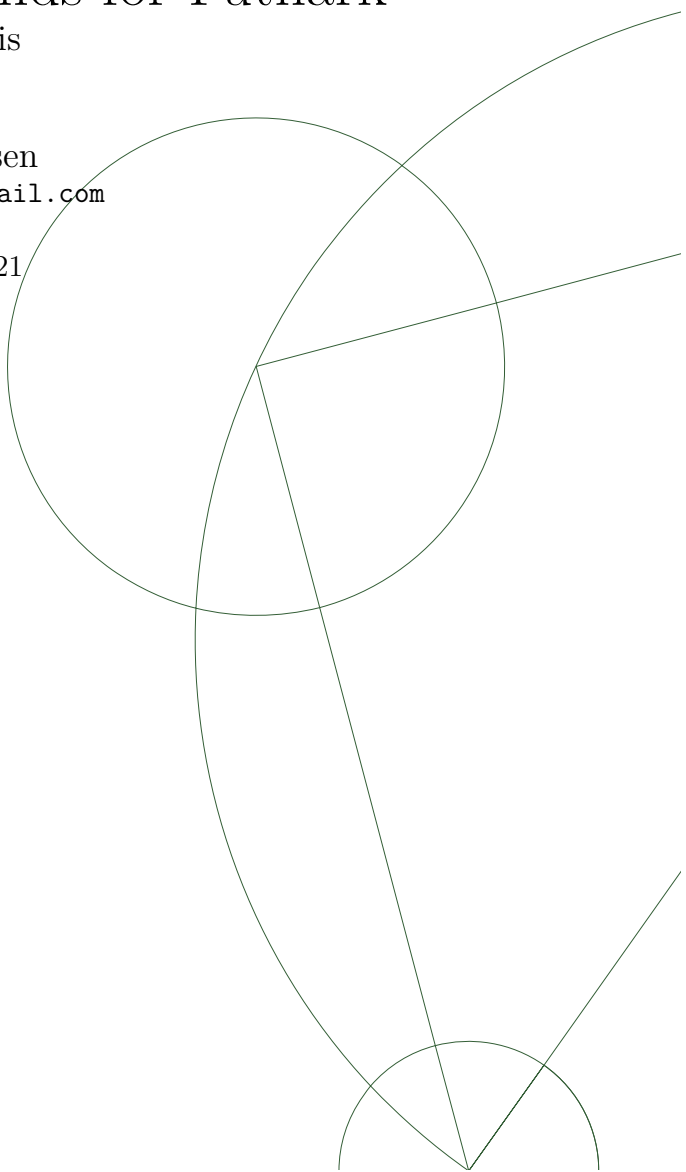
FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

WebAssembly Backends for Futhark

Msc Thesis

Philip Lassen
philiplassen@gmail.com

June 29, 2021



Abstract

Futhark is a high-performance purely functional data-parallel array programming language targeting parallel compute hardware. Futhark has backends for several compute architectures and this thesis adds browsers by targeting WebAssembly and threaded WebAssembly. These are browser technologies which map better to the underlying hardware of devices, including multicore CPUs.

A JavaScript API is developed for easily calling compiled Futhark WebAssembly libraries in the browser. The implementation and generated WebAssembly code is benchmarked for both browsers and Node.js, against the Futhark sequential C and multicore C backends. The sequential WebAssembly performs close to sequential C speeds. The parallel execution of threaded WebAssembly speeds up some example programs by a factor equal to the number of physical CPU cores.

Contents

Abstract **i**

Contents **iii**

1 Introduction **1**

2 Background **3**
2.1 Programming Languages in the Browser 3
2.2 Parallel Programming in the Browser 4
2.3 Futhark 5

3 WebAssembly **7**
3.1 WebAssembly Module Structure 7
3.2 Memory 8
3.3 WebAssembly and JavaScript Interaction 10
3.4 Emscripten 10

4 API Design **15**
4.1 Comparing Futhark APIs 15
4.1.1 C API 16
4.1.2 Python API 17
4.1.3 API Comparison 19
4.2 JavaScript API 19
4.3 Memory Management 23
4.4 Summary 23

5 WebAssembly Backend **25**
5.1 WebAssembly Code Generation 25
5.2 Library Implementation 27
5.2.1 Interacting with C functions 28
5.2.2 FutharkArray 30
5.2.3 Opaques 33
5.2.4 Error Handling 34
5.2.5 Remaining details 35

5.3 Compiler Pipeline 36
5.3.1 Emscripten Compiler Flags 37
5.4 Application 38
5.5 Benchmarking 40
5.6 Testing 41

6 Parallel Execution in the Browser 43
6.1 Web Workers 43
6.2 Shared Memory and Atomics 45
6.3 Threaded WebAssembly 48

7 WebAssembly Multicore Backend 51
7.1 Implementation Structure 51
7.2 Implementation Details 52
7.2.1 Plumbing 52
7.2.2 Multicore C code changes 52
7.2.3 API Change 52
7.2.4 Emscripten Invocation 53
7.2.5 Running in browser with HTTP 53
7.2.6 Applications 54
7.3 Benchmark 57
7.4 Summary 58

8 Conclusion 59

A Source Code 61
A.1 GenericWASM.hs 61
A.2 mandelbrot.fut 67
A.3 raytracer.fut 68

Bibliography 73

CHAPTER 1

Introduction

The last two decades have seen a proliferation of consumer devices—laptops, tablets, phones—with parallel computing capabilities with GPUs and multicore CPUs. A large portion of the software running on these devices runs in the browser.

For decades, JavaScript has been the standard tool for programming in the browser. JavaScript is ubiquitous but it suffers from inefficient execution. Moreover, there is an increasing interest in compiling other programming languages to run in the browser and JavaScript is not optimal as a compilation target in terms of code size and execution speed. Attempts to solve these issues have led to the development of first asm.js and then WebAssembly, which acts as a kind of assembly language for the browser. WebAssembly is compact and executes at near native speeds and is now supported as a compilation target for many major programming languages, including C, C++, and Rust. However, WebAssembly is single threaded and doesn't utilize parallel computing capabilites of the underlying hardware.

Development into utilizing parallelism is ongoing with efforts being made both in the context of GPUs and multicore CPUs. One is WebGPU, a proposed web standard and JavaScript API for calling GPUs in the browser. Another is threaded WebAssembly, an experimental extension to WebAssembly that supports parallel execution on multicore CPUs using Web Workers and provides an interesting compilation target for parallel programming languages to run in the browser.

Futhark is a high-performance purely functional data-parallel array programming language targeting parallel compute hardware, primarily GPUs. More recently, the Futhark compiler has gained an additional backend targeting multicore CPUs.

This thesis develops backends for compiling Futhark to WebAssembly and threaded WebAssembly for efficient, parallel execution in web browsers, together with a convenient and efficient JavaScript API for Futhark interoperation with browser applications. In this way this thesis contributes towards

extending high level and high performance parallel programming to modern consumer devices.

Objectives

The objectives of this thesis are to:

- Survey technologies for high performance and parallel programming in the browser.
- Design an API for interoperation between Futhark and browser applications.
- Implement Futhark backends that generate libraries that run efficiently and in parallel in browsers.
- Evaluate the performance of these backends.

Thesis Structure

This thesis is organized with the following structure:

- Chapter 2, Background: Overview of the related work that this thesis builds on.
- Chapter 3, WebAssembly: Explanation and analysis of WebAssembly as a programming language and as a target language for the browser.
- Chapter 4, API Design: We develop an API for calling Futhark WebAssembly libraries from JavaScript.
- Chapter 5, WebAssembly Backend: Description of the implementation of the WebAssembly backend. As well as a overview of the performance bench-marked against the native C backend.
- Chapter 6, Parallel Execution in the Browser: Analysis of the facilities and paradigms for parallel programming in the browser through JavaScript and WebAssembly.
- Chapter 7, WebAssembly Multicore Backend: Description of the implementation of the multicore WebAssembly backend. As well as an overview of the performance benchmarked against the native multicore C, and WebAssembly backend developed in Chapter 5.
- Chapter 8, Conclusion: Summary of the implementations and performance of the backends and API developed. As well as a brief discussion of future developments for Futhark targeting parallel compute in the browser.

CHAPTER 2

Background

This chapter describes relevant background for the work in this thesis, namely browser programming facilities and the Futhark programming language.

2.1 Programming Languages in the Browser

In the early days of web browsers, web pages would render differently across browsers as web APIs were not standardized. Different browsers supported different programming languages, e.g. Java applets and VBScript, creating headaches for programmers who wanted their websites to render identically across browsers. The first popular language for the browser was JavaScript, but the implementations across browsers differed. Eventually the big vendors converged on JavaScript releasing a standardized version called ECMAScript.

Huge investments have been made to increase the execution speed of JavaScript in the browser. All the major browser vendors have optimized the performance of their JavaScript engines, in particular V8 in Chrome and SpyderMonkey in Mozilla. Many approaches have been taken to make JavaScript faster but they have all been fundamentally limited by the language design.

One of the approaches taken by the browser vendors was to define a subset of the language asm.js and a convention for type hints, which were designed for efficient execution by leveraging types and compiler tricks to allow ahead-of-time compilation. It was intended as a target language for compilation of statically typed programming languages. Emscripten [19] was developed to be a C/C++ to asm.js compiler.

Google also introduced Google Native Client (NaCl) [18] as a way to bridge the speed gap between running code in the browser, and natively. NaCl is a sandbox for running compiled C and C++ code in the browser efficiently and securely, independent of the user's operating system [18]. However it struggled to gain adoption due to its lack of portability as it was only supported by Chromium based browsers.

The major browser vendors collaboratively designed WebAssembly (Wasm) [4] to more comprehensively address the limitations of JavaScript as a target language for the web. It is a portable low level byte code, designed for compact representation, efficient compilation, and near native execution speeds. WebAssembly is gaining adoption and has been used for a variety of applications especially as a target for compilation from C, C++ and Rust.

Google Earth [9] is an example of a major application that is adopting WebAssembly. Google Earth renders a 3D representation of Earth based primarily on satellite imagery. It started out as a desktop application, but in 2013 was ported to the web. It originally only ran in Chrome, as it was built on NaCl. The developers tried to build it with asm.js, but found the binary sizes of compiling over a million lines of code with Emscripten to be infeasibly large. However with the creation of WebAssembly they were able to make a high performance cross browser implementation of Google Earth due to the speed and small binary sizes of WebAssembly [12].

Another example is TensorFlow [10], an open source machine learning library, originally written in C++. Due to the large eco-system of JavaScript developers and its ability to run on the browser, the developers introduced TensorFlow.js [14]. They have multiple backends including a WebAssembly backend, which is 10-30x faster than their plain JavaScript backend [2]. What is interesting about TensorFlow.js is that it shows how WebAssembly can be used to create high performance libraries that can be called from the web.

One of the technologies that has greatly helped the adoption of WebAssembly as a target language is the LLVM [8] compiler tool chain. Writing a full compiler from scratch that supports multiple targets is a huge undertaking. In order to have high performance backends for different target architectures such x86 and ARM requires knowledge of many of the low level details of each respective target. An alternative approach is for the compiler frontend of the source language to take the source code and translate it to the LLVM internal representation (IR). The LLVM compiler tool-chain can then generate high performance code on all the most common computer architectures. Many of the biggest languages are currently built with or have compiler implementations using LLVM. LLVM compiles from its IR to WebAssembly and therefore languages that generate LLVM IR have an easy path to WebAssembly code generation. Emscripten uses LLVM and can generate WebAssembly in addition to asm.js. It is a widely used compiler for generating WebAssembly.

2.2 Parallel Programming in the Browser

While WebAssembly has progressed the state of the art of single threaded computation speed in browsers another avenue for execution speed is parallelism. Browsers have facilities for parallel programming. Javascript supports two different paradigms with web workers. Message passing enables parallel

programming without shared memory. SharedArrayBuffer and atomics enable shared memory multithreading with thread synchronization. There is a threaded WebAssembly proposal that adds atomic operations to the language, and adds support for SharedArrayBuffers while relying on JavaScript's web workers to create and join threads. Emscripten can currently compile C with POSIX threads to threaded WebAssembly. The Chrome and Firefox browsers along with Node.js have experimental support for threaded WebAssembly.

2.3 Futhark

Futhark [6] [5] is a data parallel programming languages that can generate high performance parallel code for both the CPU and GPU . Writing GPU code and multicore CPU code is difficult as there are many low level details required to make correct and optimal implementations. Futhark is a high level functional programming language, which aims to do the heavy lifting for the user.

Futhark programs are generally written with Second-Order Array Combinators (SOACs), which are similar to the filter, map, and reduce functions commonly found in many functional programming languages. These functions can be optimized to efficient parallel code. These combinators are expressive and be combined to encode code complex programs. To see this in action below is a Futhark implementation of matrix multiply:

```
let matmul [n][p][m] (xss: [n][p]f64) (yss: [p][m]f64): [n][m]f64 =
    let dotprod xs ys = reduce (+) 0 (map2 (*) xs ys)
    in map (\xs -> map (dotprod xs) (transpose yss)) xss
```

Here `dotprod xs ys` computes the dot product of two vectors `xs` and `ys` by computing the pairwise products `zs = map2 (*) xs ys` and then summing the products with `reduce (+) 0 zs`. In the last line the innermost `map` in `map (dotprod xs) (transpose yss)` computes a row vector of the product matrix and the outermost `map` generates all the rows.

This serves as an illustration of how a relatively involved operation can be written using SOACs. Not only is the implementation short, it's also fast.

This thesis is not going to explain the Futhark language in further detail. Only very short Futhark functions will be used in examples and do not require deeper understanding of the language.

Currently the futhark compiler has C backends generating Cuda, OpenCL, and sequential C code. Recently a multicore C backend was added that generates parallel code using POSIX threads (pthreads) [15]. Futhark also has two Python backends, one sequential and one using PyOpenCL. All the backends can be compiled to libraries, making it possible to call Futhark from C or Python applications. For this this thesis we build off of the sequential C backend and the multicore C backend.

CHAPTER 3

WebAssembly

This chapter gives an overview of the design and structure of the WebAssembly programming language. We illustrate the instruction set with two simple hand written WebAssembly modules. We show how functions and memory work, and how to call a module from JavaScript. Most importantly we show how to generate WebAssembly and JavaScript glue code with Emscripten and how to work with Emscripten's JavaScript API.

3.1 WebAssembly Module Structure

This section describes some of the lower level details of the WebAssembly programming language. They illustrate some of WebAssembly's characteristics but they are not critical for understanding the rest of the thesis.

A WebAssembly file is commonly referred to as a module, and given a *.wasm* file extension. WebAssembly also defines a text format that serves to be a human readable version of the underlying binary format, much in the same way assembly provides a human readable format for machine code. The `wat2wasm` program from The WebAssembly Binary Toolkit[1] compiles the textual format to a binary module.

WebAssembly modules are segmented into sections. The segmentation of these sections is done so that loading a WebAssembly file is as efficient as possible. The sections are structured such that the byte code can be compiled in a single pass, and in parallel. Furthermore the code can be parsed and compiled before the complete WebAssembly file has been downloaded, reducing the instantiation time of a WebAssembly module.

WebAssembly supports the 4 number types of 32-bit and 64-bit integers, `i32` and `i64`, and floats, `f32` and `f64`. These don't map cleanly to JavaScript's number types but are useful for supporting number types for languages like C/C++ and Rust, the languages it aims to be a target language for.

[1] https://github.com/WebAssembly/wabt

A WebAssembly function has the following structure.

```
( func <signature> <locals> <body> )
```

The signature gives the function name, parameter types, and return types. The locals are the local variables that will be used in the execution of the function, and the body is the actual implementation of the function. A simple WebAssembly example is the add function.

```
(module
  (export "add" (func $add))
  (func $add (param $a i32) (param $b i32) (result i32)
    local.get $a
    local.get $b
    i32.add
  )
)
```

The function signature of add specifies the two arguements a and b as `i32` and specifies the return type result as `i32`. There are no locals. The function body has three instructions. The instructions `local.get $a` and `local.get $b` push the two arguments to onto the stack. The instruction `i32.add` pops the two elements off the stack and pushes their sum. The function returns the number on the stack.

The body of the function could be replaced with:

```
(i32.add (local.get $a) (local.get $b))
```

That is, WebAssembly allows the programmer to use a notation where arguments to instructions are passed as parameters instead of manually being placed on the stack.

3.2 Memory

A notion of memory is needed for writing more complex programs. In a language like C, it is common practice to use pointers to locations in memory, or for writing an array of values. Memory from the perspective of WebAssembly is just an array of bytes that can be read from and written to. WebAssembly has two essential functions for interacting with this array, namely the `load.i32` and `store.i32`, for reading and writing to the array of bytes respectively.

As a motivating example, the following C code gives a simple implementation of a place prefix sum. (Again, this example is just for illustration and not needed to understand the rest of the thesis.)

```
void prefix_sum(int* arr, int size) {
  for (i = 1; i < size; i++) {
    arr[i] += arr[i-1];
  }
}
```

The following code is a WebAssembly implementation of in-place prefix sum.

```
 1 (module
 2   (import "env" "memory" (memory $memory 1))
 3   (export "prefixSum" (func $prefixSum))
 4   (func $prefixSum (param $size i32)
 5     (local $offs i32)
 6     (local $acc i32)
 7     (local $last i32)
 8     (local.set $offs (i32.const 4))
 9     (local.set $last (i32.mul (local.get $size) (i32.const 4)))
10     (local.set $acc (i32.load (i32.const 0)))
11     loop $forloop
12       (local.set $acc (i32.add (local.get $acc)
                          (i32.load (local.get $offs))))
13       (i32.store (local.get $offs) (local.get $acc))
14       (local.set $offs (i32.add (local.get $offs) (i32.const 4)))
15       (br_if $forloop (i32.ne (local.get $offs)
                                 (local.get $last)))
16     end $forloop
17   )
18 )
```

For the implementation, a local variable accumulator is set to the first value of the array and an offset is set to 0. A loop is then entered, where the element at offset in the array is loaded with `i32.load` and added to the accumulator, `$acc`. The result is stored with `i32.store`. The offset is increased by 4 bytes, and then compared to the local variable `$last`. If it is not equal the loop goes back to the loop on line 12, and repeats.

The most important details of the function implementation for understanding WebAssembly's interaction with memory are the load and store operations, which use byte offsets to address memory. The memory is never explicitly referenced in the function because WebAssembly modules only have one declaration of memory in the memory section, making the array of memory implicit. Memory can either be imported from JavaScript, in which case the memory is created in JavaScript and passed to WebAssembly on instantiation, or, alternatively, the memory can be exported from WebAssembly, in which case the memory is created in WebAssembly on instantiation and can be accessed in JavaScript afterwards. The memory is imported in line 2:

```
(import "env" "memory" (memory $memory 1))
```

The ending, 1, sets the size of the memory heap to be 1 page of memory. For WebAssembly 1 page corresponds to 64 kilobytes. Memory is always set to an integer number of pages.

3.3 WebAssembly and JavaScript Interaction

Listing 1 shows how to load, instantiate, and call the *add.wasm* module from JavaScript in a web page.

```
<html>
  <head>
  <script>
  fetch('add.wasm')
  .then(r => r.arrayBuffer() )
  .then(r => WebAssembly.instantiate(r, { }))
  .then(asm => console.log("4 + 6 =", asm.instance.exports.add(4,6)));
   </script>
   </head>
   <body></body>
</html>
```

Listing 1: Invocation of *add.wasm* in web page

By design a WebAssembly module only interfaces with its environment through function inputs and outputs, and through memory. The most clear cut example of this is the lack of access to the DOM. WebAssembly does not have direct access to Web APIs. However it can call imported JavaScript functions. This facility allows it to interact with Web APIs. This simple design feature is powerful. It is this feature that enables threaded WebAssembly as it can leverage JavaScript's ability to spawn web workers. This will be discussed in greater detail in Chapter 6.

3.4 Emscripten

This section describes how to generate WebAssembly and JavaScript glue code with Emscripten and how to work with Emscripten's JavaScript API. This material is critical for understanding the backend implementation in chapter 5.

As described in chapter 2, the popular Emscripten toolchain compiles C/C++ to a WebAssembly module. It also generates JavaScript "glue code", a phrase we use with the specific meaning of code that Emscripten generates to encapsulate module instantiation and access through Emscripten's JavaScript API. Technically the library user can load the WebAssembly module directly as in listing 1. However this is impractical for a couple of reasons. The module

is constructed by Emscripten to import a series of system library functions to access files and allocate memory etc. Emscripten's glue code supplies all these at instantiation time and hides this from the library user.

For many use cases, compiling code to a library is desirable. This way a programmer can write source code in C/C++ that contains functions for compute intensive parts of their application. They can then generate WebAssembly modules that can be called from JavaScript, and offload the computation to WebAssembly.

It is relatively straightforward to call simple C functions from JavaScript when they are compiled with Emscripten. We just need to explicitly export the functions at compile time.

If we have a simple C file *add.c* that contains an add function.

```
int add(int a, int b) { return a + b; }
```

We compile with Emscripten to a library while making sure to export the function.

```
emcc add.c -o add.js -s MODULARIZE -s EXPORTED_FUNCTIONS=[_add]
```

We also use the MODULARIZE flag, to make the WebAssembly Module easier to import. This is helpful because WebAssembly is loaded asynchronously. With the MODULARIZE flag we are able to get the module as a promise. The following code runs in Node.js:

```
var load_module = require('./add.js');
load_module().then((instance) => {
  console.log(instance._add(4, 6));
});
```

The `load_module` is a factory function. Once the WebAssembly Module is loaded we run the code in the callback. The actual logic for running the C function is just the one line:

```
  console.log(instance._add(4, 6));
```

The example library can also be loaded as a script in a web page and invoked from JavaScript in the browser. This will be shown later in the Mandelbrot example in Section 5.4.

Library functions compiled with Emscripten are easy to use when they take integer arguments and have integer return types. However lots of C/C++ works with pointers. Emscripten models that with locations in a single memory region attached to the WebAssembly module. This memory region is called the heap. Emscripten offers multiple views into the heap with a typed array for each primitive element type, namely those supported by JavaScript typed arrays. See Table 3.1. These typed array all share the same underlying ArrayBuffer memory.

Heap view	JavaScript type
HEAP8	Int8Array
HEAP16	Int16Array
HEAP32	Int32Array
HEAP64	BigInt64Array
HEAPU8	Int8Array
HEAPU16	Uint16Array
HEAPU32	Uint32Array
HEAPU64	BigUint64Array
HEAPF32	Float32Array
HEAPF64	Float64Array

Table 3.1: Emscripten's heap views and their JavaScript types

Typed arrays will play an important role in this thesis, both for interacting with the Emscripten heap and for array parameters to compiled Futhark functions. Typed arrays are useful because they represent number arrays compactly and efficiently, and the ability for typed arrays to act as different views of the same underlying ArrayBuffer can be used to avoid memory copies for parameter passing in many cases.

To illustrate how to pass pointer arguments, consider the simple C string function in listing 2 which takes two strings and return their concatenation into allocated memory that the caller must free.

```c
#include <string.h>
#include <stdlib.h>
char* concat(const char* a, const char* b) {
  int alen = strlen(a);
  int blen = strlen(b);
  char* res = malloc(alen + blen + 1);
  strcpy(res, a);
  strcpy(res + alen, b);
  return res;
}
```

Listing 2: C concat function

To use this function the caller will need to use the malloc and free system library functions, so we export those alongside concat when we compile to a library with Emscripten:

```
emcc concat.c -o concat.js -s MODULARIZE \
  -s EXPORTED_FUNCTIONS=[_malloc,_free,_concat]
```

The Node.js program in listing 3 calls concat.

```
var load_module = require("./concat.js");
load_module().then((instance) => {
  var hello = "hello ";
  var alen = hello.length;
  var a = instance._malloc(alen + 1);
  new TextEncoder().encodeInto(hello, instance.HEAPU8.subarray(a, a + alen));
  instance.HEAPU8[a + alen] = 0;
  var world = "world!";
  var blen = world.length;
  var b = instance._malloc(blen + 1);
  new TextEncoder().encodeInto(world, instance.HEAPU8.subarray(b, b + blen));
  instance.HEAPU8[b + blen] = 0;
  var c = instance._concat(a, b);
  var cend = instance.HEAPU8.indexOf(0, c);
  console.log(new TextDecoder().decode(instance.HEAPU8.subarray(c, cend)));
  instance._free(c);
  instance._free(b);
  instance._free(a);
})
```

Listing 3: Node.js call to Emscripten compiled concat function

The pointers a and b passed to `_concat` are locations in the Emscripten heap, so the caller must first copy the strings to the heap for `_concat` to access them. To find a place in the heap to place them, we call `malloc` twice and then use the `HEAPU8` heap view to write the strings into the allocated memory. We zero terminate the a string with:

```
instance.HEAPU8[a + alen] = 0;
```

and write into the first `alen` bytes through:

```
instance.HEAPU8.subarray(a, a + alen)
```

It is another Uint8Array which is a view into the subarray of the heap that holds the first alen bytes, up to and excluding the terminating zero. The returned pointer, `c`, again points into the heap and we find the heap location of the terminating zero with:

```
var cend = instance.HEAPU8.indexOf(0, c);
```

and then we get a view of the concatenated string with:

```
instance.HEAPU8.subarray(c, cend)
```

Finally, all the allocated strings are freed with calls to _free.

Emscripten's function and heap API and compiler flags will play important roles in the implementation of the WebAssembly backends for Futhark in chapter 5 and 7.

CHAPTER 4

API Design

The WebAssembly backends that we are going to develop will compile Futhark functions to libraries for use in the browser. This section designs a JavaScript API for programs in the browser to call the compiled library functions.

In order to make Futhark a practical language for writing libraries that can be called in the browser it is important that it has a simple and efficient API. As discussed in chapter 3 code that is compiled to WebAssembly modules typically comes packaged with a JavaScript file. This file contains the library functions that are exposed to the programmer, where internally these functions handle the interaction with WebAssembly. This allows users of the library to be oblivious of the fact that WebAssembly is being used under the hood, much in the same way that some Python developers are oblivious that numpy is running compiled C under the covers.

A successful API for calling Futhark in the browser will have the following properties

1. It should be convenient, seamlessly integrating with the most commonly used JavaScript number and array data types.

2. The API should be efficient both with respect to memory usage and runtime speed.

3. Finally the API should minimize boiler plate code.

4.1 Comparing Futhark APIs

It is insightful to look at the APIs of other languages so that we can copy the elements of the APIs that are effective and avoid the parts that are cumbersome or inefficient. These decisions are constrained by the limitations and capabilities of the target language, which for us is JavaScript and WebAssembly.

An additional reason why it is important to observe the C API, is that the WebAssembly backend builds off the sequential C backend. For this reason our design choices for the API are limited to how we can wrap the C API function calls behind JavaScript classes and functions.

The exposition will be example based. The first example is the minimal program in listing 4 that takes a 32 bit integer and returns the successor.

```
entry increment (a : i32) = a + 1
```

Listing 4: Futhark increment function, *increment.fut*

The second example in listing 5 works with multidimensional array inputs and outputs as well as a scalar input:

```
entry scale (scalar : f32) (matrix : [][]f32) =
  map (map (scalar *)) matrix
```

Listing 5: Futhark scale function, *scale.fut*

The scale function multiplies each element of a matrix by a scalar, and returns the scaled matrix.

4.1.1 C API

When *increment.fut* is compiled as a C library, the files *increment.c* and *increment.h* are generated. The implementation and header files respectively. Listing 6 is a C program that interfaces with the generated library.

```c
#include <stdio.h>
#include "increment.h"

int main() {
  // Initialize config and context
  struct futhark_context_config *cfg = futhark_context_config_new();
  struct futhark_context *ctx = futhark_context_new(cfg);

  int32_t res;
  futhark_entry_increment(ctx, &res, 42);
  printf("%d\n", (int) res);

  futhark_context_free(ctx);
  futhark_context_config_free(cfg);
}
```

Listing 6: C code for interacting with the C API of the compiled program *increment.fut*

At the top level the API works off of a context and a configuration, `futhark_context`, and `futhark_context_config`. The configuration stores

choices like debugging. The context manages global information and bookkeeping. Including logging and profiling, and a mutex to guarantee thread safety when used from multiple threads. The API has functions to create these at the beginning and free them at the end.

```
futhark_context_config_new
futhark_context_new
futhark_context_free
futhark_context_config_free
```

Futhark's primitive types (bool, integers and floats) are represented by corresponding C types.

For arrays futhark generates a C type for every type that appears as an argument or result type in any entry point function in the library. It also generates functions to create, access, and free arrays. For example for the futhark array type `[][]f32`, the generated header file *scale.h* declares a type `futhark_f32_2d` and functions `futhark_new_f32_2d` and `futhark_free_f32_2d` for creating and freeing arrays of that type. Moreover the API generates functions `futhark_shape_f32_2d` and `futhark_values_f32_2d` for getting the shape (dimensions) and values of the futhark array respectively.

For each entry point function in the library a C function is generated, which takes the futhark context, an output parameter for each type in the result tuple, as well as an input parameter for each argument. The function signature for the generated C function for *scale.fut* can be seen below.

```
int futhark_entry_scale(
  struct futhark_context *ctx,
  struct futhark_f32_2d **out0,
  const float in0, const struct futhark_f32_2d *in1);
```

Listing 7 is a C program that calls `futhark_entry_scale`.

Memory management is manual. Both the input array created with `futhark_new_f32_2d` and the output array returned from `futhark_entry_scale` are freed manually.

4.1.2 Python API

When *scale.fut* is compiled as a Python libray it generates a single Python file *scale.py*. It contains a class that can be used to interact with the Python methods.

```c
#include <stdio.h>
#include <stdlib.h>
#include "scale.h"

int main() {
  float scalar = 0.5;
  float arr_2d[12] = {0, 1, 2, 3,
                      4, 5, 6, 7,
                      8, 9, 10, 11};

  // Initialize config and context
  struct futhark_context_config *cfg = futhark_context_config_new();
  struct futhark_context *ctx = futhark_context_new(cfg);

  // Turn array_2d into futhark type
  struct futhark_f32_2d *fut_arr_2d = futhark_new_f32_2d(ctx, arr_2d, 3, 4);

  // Initialize futhark result array
  struct futhark_f32_2d *res;

  futhark_entry_scale(ctx, &res, scalar, fut_arr_2d);

  const int64_t* shape = futhark_shape_f32_2d(ctx, res);

  float res_arr[shape[0] * shape[1]];
  futhark_values_f32_2d(ctx, res, res_arr);
  for (int i = 0; i < shape[0]; i++) {
    for (int j = 0; j < shape[1]; j++)
      printf("%f ", res_arr[i * shape[1] + j]);
    printf("\n");
  }

  futhark_free_f32_2d(ctx, res);
  futhark_free_f32_2d(ctx, fut_arr_2d);

  futhark_context_free(ctx);
  futhark_context_config_free(cfg);
}
```

Listing 7: C code for interacting with the C API of the compiled program *scale.fut*

```python
import numpy as np
import scale

scalar = 0.5
matrix = [[0, 1, 2, 3],
          [4, 5, 6, 7],
          [8, 9, 10, 11]]

np_matrix = np.array(matrix).astype("float32")

scale_class = scale.scale()
result = scale_class.scale(scalar, np_matrix)
print(result)
```

Listing 8: 64 bit multiplication with 128 bit casting

In order to invoke the python library we import the file. Then we instantiate a `scale_class` by calling the `scale()` method. Finally we invoke the method by calling the entry point function with a scalar input and a 2-

dimensional numpy array. All primitive Futhark types conveniently map to primitive numpy types. Numpy ndarray are made up of these primitive types so passing them in as arguments in the API provides sufficient information. We make sure that the numpy array has values of the correct type by calling the `astype` function of numpy with `f32`, the numpy equivalent of the futhark primitive `f32`. The entry point function logically returns a numpy ndarray whose elements have type `float32`. Interestingly numpy scalars are returned by the Python API if the return time is a scalar. However the entry point function can both accept scalars as normal Python numbers or as numpy scalars.

4.1.3 API Comparison

One of the features of the Python API that make it simpler is that it wraps the context and configuration in a class. For the C backend the context needs to be passed to every function call that interacts with Futhark specific functions. The other big advantage of the Python API is that it utilizes the numpy library which provides convenient classes for representing multidimensional arrays. C does not have a clean representation of multidimensional arrays. This makes the C API more verbose.

4.2 JavaScript API

When designing the JavaScript API we consider both usability and performance.

JavaScript has one standard number, which is a 64 bit floating point number. This is typically used to encode all numbers. All the Futhark types `u8`, `u16`, `u32`, `i8`, `i16`, `i32`, `f32`, `f64`, and `bool` except for `u64` and `i64` can be encoded by a 64 bit float. Fortunately JavaScript recently introduced `BigInt` in ES2020[1] to address this shortcoming of the language. With this there is a way to represent every futhark primitive type with either standard JavaScript `number`, `boolean`, or `BigInt`.

One problem with JavaScript is that it doesn't have a standard package with widespread adoption for scientific computing that gives an efficient encoding of n-dimensional arrays. The standard JavaScript array are more reminiscent of lists and have similar semantics to the list type of Python. A problem is that there is no way to efficiently validate the shape for n-dimensional arrays and the types of its elements.

[1] ES2020 is the 2020 version of the JavaScript language specification [7]

TypedArrays

The approach that many scientific JavaScript libraries take is to accept typed arrays as arguments, as well as arguments for specifying the shape and dimensions of the typed arrays. Typed arrays are briefly discussed in chapter 3, but as a reminder typed arrays are standard JavaScript types. There are 10 different typed arrays types, one for each of the 10 common numeric types. These happen to correspond to Futhark's scalar types, with the exception of booleans. Typed arrays have a number of advantages:

- Type Enforcement : typed arrays only store the type that is specified by their constructor.
- Memory Efficiency : typed arrays can store types with less than 64 bit precision more efficiently as they don't need to use 64bit float to also store high order bits. As an example a typed array of 8-bit integers takes an eighth of the space compared to a regular array storing numbers.
- Ubiquity : Typed arrays are present in all standard JavaScript runtimes, such as Node.js, and Chrome. They don't need to be imported and are often the return types of scientific libraries such as for image or graphics processing

In order to maintain the most flexibility with our API design we choose to both have a way to work with the API through simple JavaScript Arrays, as well as through typed arrays. Typed arrays will have a little extra baggage to carry along shapes.

Futhark primitive types for all types except `i64`, `u64` and `bool` are represented by the standard JavaScript number type. `i64` and `u64` are represented by the `BigInt` number type. `bool` is represented by the standard JavaScript `boolean` type. The type conversion between Futhark types and JavaScript types can be seen in listing 9.

All array values share the same structure API. They can be passed in as n-dimensional arrays of their associated JavaScript type. Note that these arrays must be regular, it is on the caller to enforce this. Alternatively we provide a FutharkArray class which can be instantiated from typed arrays. FutharkArray instances are also accepted by entry point functions. Users who have a focus on (space and speed) efficiency should elect to use this over regular JavaScript arrays. Users using a more efficient representation of their data with typed arrays aren't forced to convert into a less efficient representation.

Finally there are Opaque types, which don't have a clean mapping from Futhark to JavaScript. Instead they are represented with a FutharkOpaque class. This class doesn't have meaning outside of the context of being passed from the output of one entry point to the input to another entry point with the same type.

Futhark Type	JavaScript Type
i8	number
i16	number
i32	number
i64	BigInt
u8	number
u16	number
u32	number
u64	BigInt
f32	number
f64	number
bool	boolean
[]i32	FutharkArray
[][]f64	FutharkArray
opaque	FutharkOpaque

Listing 9: Type conversion table from Futhark types to primitive JavaScript types

FutharkContext

FutharkContext is a class that contains information about the context and configuration from the C API. It has methods for invoking the Futhark entry points and creating FutharkArrays on the WebAssembly heap.

It is instantiated with the empty constructor.

```
var fc = new FutharkContext();
```

`FutharkContext` has methods for creating FutharkArrays, one method for each futhark array type that appears as an argument to any entry point function. Each method has the following signature

```
new_<type>_<n>d(typedArray, dim_1, ..., dim_n)
```

Working off of the example *scale.fut*, to create FutharkArray for a matrix of `[][]f32`, we would make the following call.

```
var fc = new FutharkContext();
var typedArray = new Float32Array([0.5, 0.4, 0.3, 0.2, 0.1, 0.0]);
var futharkArray = fc.new_f32_2d(typedArray, 3, 2);
```

Entry Points

Each entry point function in the compiled module has a method on the FutharkContext. Scalar parameters are either `BigInt` numbers, regular JavaScript `numbers`, or booleans. Array parameters can be either FutharkArrays or, for convenience, JavaScript Arrays.

The method returns an array if the futhark return type is a tuple. Otherwise it returns a single value.

Invoking the entry points of *scale.fut* looks something like this:

```
var fc = new FutharkContext();
var typed_array = new Float32Array([0.5, 0.4, 0.3, 0.2, 0.1, 0.0]);
var fut_arr = fc.new_f32_2d(typed_array, 3, 2);
var fut_arr_result = fc.scale(0.5, fut_arr);
console.log(fut_arr_result.toTypedArray(), fut_arr_result.shape());
```

As stated earlier entry points also allow JavaScript Arrays as inputs, provided they have the correct types and dimensions and are regular. So an alternative approach with JavaScript arrays would look like this:

```
var fc = new FutharkContext();
var matrix = [[0.5, 0.4],
              [0.3, 0.2],
              [0.1, 0.0]];
var fut_arr_result = fc.scale(0.5, matrix);
console.log(fut_arr_result.toArray());
```

The second implementation is simpler but also less efficient as regular JavaScript arrays require multiple levels of type conversion under the hood.

The `fut_arr_result` in both the examples is a FutharkArray. FutharkArray has methods for getting the underlying data out into basic JavaScript as either a TypedArray or JavaScript Array.

We have omitted calls to free. Memory management will be discussed in the next section.

FutharkArray

The FutharkArray allows us to get the underlying data from the WebAssembly Heap to JavaScript. It provides the following methods:

- **toArray()** : Returns a JavaScript array with the correct dimensions.
- **toTypedArray()** : Returns a flat typed array of the underlying data.
- **shape()** : Gives the shape of the FutharkArray as an array of `BigInt`.
- **futharkType()** : Returns the futhark type of the elements as a string, i.e. `bool`, `u32`, `i8`, etc.

FutharkOpaques

For handling the opaque types we introduce the FutharkOpaque class. This class has no methods, other than free, which will be discussed in the next section. The opaque class does not have any meaning to the user outside of the FutharkContext it is defined in. It is only useful for cases where an entry point function on the context returns an opaque type and another entry point function on the context accepts the opaque as input. For this reason there are no additional methods on the class.

4.3 Memory Management

Unfortunately WebAssembly does not have garbage collection. This means that the programmer is responsible for doing this manually. The JavaScript API introduced 3 classes, FutharkContext, FutharkArray, and FutharkOpaque. Each has a free method, which the API user must call manually to free underlying memory on the Emscripten memory heap.

With this last information about memory management we have the full picture of the JavaScript API. Our final look at the *scale.fut* example gives us:

```
var fc = new FutharkContext();
var matrix = [[0.5, 0.4],
              [0.3, 0.2],
              [0.1, 0.0]];
var fut_arr_result = fc.scale(0.5, matrix);
console.log(fut_arr_result.toArray());
fut_arr_result.free();
fc.free();
```

4.4 Summary

Our API is relatively concise but we do make some tradeoffs between conciseness and efficiency. In particular we return arrays from entry points as FutharkArrays rather than nested JavaScript arrays. We also provide an additional factory method to create FutharkArrays from flat typed arrays. These design choices favored efficiency at the expense of some complexity. The justification is that an efficient and complex API can be made simpler, whereas a simple and inefficient API cannot be made efficient.

One drawback with our API design is that the caller has to free memory, by calling free methods on the FutharkContext, FutharkArray, and FutharkOpaque objects. However this is a pain point of working with WebAssembly as it does not provide garbage collection.

CHAPTER 5

WebAssembly Backend

This chapter describes the implementation of an additional Futhark backend that generates WebAssembly and JavaScript code such that Futhark programs can be compiled to libraries that can be run in the browser. The implementation is benchmarked in the Chrome browser, and in Node.js and against the C backend.

The backend implementation takes outset in the existing futhark Sequential C backend and passes the generated C code to the Emscripten compiler for the final compiler pass. Emscripten generates a WebAssembly module along with JavaScript glue code. We generate additional JavaScript code to wrap our API around Emscripten's JavaScript API.

To illustrate the backend implementation working in practice we use the WebAssembly backend to efficiently generate graphics for the Mandelbrot set in a web page.

Finally we benchmark the Futhark backend to quantify its performance. The benchmarks come from the futhark benchmark suite[1], which are standardized benchmarks to test the performance of Futhark backends against industry standards. The benchmarks show the WebAssembly, both when run in the Browser and run on Node.js locally, performs between 13% to 67% slower than the generated C running locally.

5.1 WebAssembly Code Generation

We will now describe the implementation of the WebAssembly backend for Futhark.

We considered three choices for generating WebAssembly:

- Generating WebAssembly directly: This is the least attractive of all the options. Writing low level code is time consuming and also inefficient, in

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

the sense that it would be a huge investment to capture the optimizations expected of a modern compiler.

- LLVM IR: This is the most standard approach among compilers. However since Futhark doesn't already generate LLVM IR utilizing this approach would effectively require a redesign of the compiler architecture. This would be an interesting project, with possible positive implications in more places than just a WebAssembly backend. However this would be quite a large task, as the Futhark frontend wasn't designed with this specific intermediate representation in mind.

- Emscripten: Emscripten can simply take C source code generated from the Futhark compiler. This approach would require the fewest modifications to the compiler. Futhark also aims to emit high performance C code, and Emscripten aims to translate C to high performance WebAssembly code. Connecting the two compilers together is the approach that is most logical.

We chose to go with the Emscripten approach. We started by running Futhark's generated C code through Emscripten and surfaced a handful of issues that we describe in the following.

Futhark's C backend generates over 2200 lines of C code when compiling even a minimal function like the increment function from listing 4. This is because it contains logic for option parsing, logic for parsing input and formatting output, and library functions for mathematical operations.

One issue was in platform specific code used for timing but on further inspection it turned out that this code was a relic and not actually used anywhere in the compiler, and therefore could just be deleted.

Listing 10 gives the other function implementation that Emscripten couldn't compile.

```
1 static uint64_t futrts_mul_hi64(uint64_t a, uint64_t b)
2 {
3     __uint128_t aa = a;
4     __uint128_t bb = b;
5
6     return aa * bb >> 64;
7 }
```

Listing 10: 64 bit multiplication with 128 bit casting

The issue is that Emscripten and WebAssembly don't support `uint128` types. Instead an alternate implementation for calculating the high order bits of 64 bit multiplication that doesn't cast to `uint128` numbers is used [2].

[2]The solution to this problem was found at `https://stackoverflow.com/a/28904636`

```
1  static uint64_t futrts_mul_hi64(uint64_t x, uint64_t y)
2  {
3    uint64_t a = x >> 32, b = x & 0xffffffff;
4    uint64_t c = y >> 32, d = y & 0xffffffff;
5    uint64_t ac = a * c;
6    uint64_t bc = b * c;
7    uint64_t ad = a * d;
8    uint64_t bd = b * d;
9    uint64_t mid34 = (bd >> 32) + (bc & 0xffffffff) + (ad & 0xffffffff);
10   uint64_t upper64 = ac + (bc >> 32) + (ad >> 32) + (mid34 >> 32);
11   return upper64;
12 }
```

Listing 11: 64 bit multiplication without 128 bit casting

With the small modifications described to get the generated C to a state where it can be compiled with the Emscripten compiler, Futhark programs can be compiled to WebAssembly and run with Node.js as executables. Note that this is specifically for executables and not C libraries. However we are primarily concerned with getting Futhark running in the browser as a library such that web developers can design high performance libraries in Futhark and then call them from JavaScript in the browser.

5.2 Library Implementation

In order use the JavaScript and WebAssembly code that is generated by Emscripten as a library, the C library functions need to be exported during the Emscripten compilation process. The library functions that need to be exported are the ones that the Futhark C backends emit into the C header file when the futhark program is compiled as a library. Again this whole process works almost out of the box when connecting Futhark's C library code generator with the appropriate Emscripten command. However an issue is that `int64` types are hard to work with when they are provided argument types to functions. numbers in JavaScript are 64-bit floating-point values. This means that all 32 bit numbers can be represented in JavaScript, but not all 64 bit numbers can be represented with full precision. By default Emscripten solves this by passing two arguments to JavaScript functions, one for the low order 32 bits, and another for the high order 32 bits. Instead we take an alternative approach offered by Emscripten with the `WASM_BIGINT` compiler flag. This gets 64-bit integers working in C with the caveat that when the function is called from JavaScript, the argument must be provided as a JavaScript BigInt.

With these fixes a programmer familiar with the semantics of the Emscripten heap has a workable WebAssembly library. Emscripten generates two files when compiling C code. One is a WebAssembly module, which contains the instructions for computing the functions described in the C source code. The other file is JavaScript glue code, which is used to take care of

loading the WebAssembly file, as well as handling any JavaScript calls that the WebAssembly module might use.

The WebAssembly module has methods for each function that appears in our .c file, plus some system library functions like malloc and free. To call these functions from JavaScript we must tell Emscripten with the `-s EXPORTED_FUNCTIONS` compiler flag to export them. The method names are all prefixed with an underscore.

5.2.1 Interacting with C functions

Given all this we can now work with Futhark from JavaScript and do the equivalent of listing 6. Listing 6 showed how to call Futhark's generated C library from the *increment.fut* example. After compiling the C library to WebAssembly with Emscripten, the library can be called with the JavaScript equivalent shown in listing 12.

```
1  var cfg = _futhark_context_config_new();
2  var ctx = _futhark_context_new(cfg);
3
4  var input = 42;
5  var out_ptr = _malloc(4);
6  _futhark_entry_increment(ctx, out_ptr, input);
7  var result = HEAP32[out_ptr >> 2];
8  _free(out_ptr);
9
10 console.log(result);
11
12 _futhark_context_free(ctx);
13 _futhark_context_config_free(cfg);
```

Listing 12: Working with raw Emscripten for the increment function

This is quite cumbersome for a function that simply takes a single integer argument and returns a single integer. The program is basically a one to one match of the C equivalent from listing 6, including passing around pointers, which is unsafe and not idiomatic in JavaScript. The allocation and freeing of `out_ptr` in lines 5 and 8 are even worse than the equivalent C. They are needed because the output 32-bit integer parameter can only write to the Emscripten heap and therefore the caller must explicitly allocate and free the 4 bytes, and copy out the result from the heap using the HEAP32 heap view. This isn't the case in C because of the shared address space between the caller and the library function, allowing the caller to pass a pointer to a stack allocated result variable.

The picture gets even worse when the Futhark entry points accept and return array arguments. These necessitate copying array contents to and from the Emscripten heap, exacerbating the C like manual memory management which is a poor fit in JavaScript.

We now illustrate how to work with the more complex *scale.fut* from listing 7. As a reminder the futhark function takes a 2d array of floats and a scalar float as input and returns a 2d array of floats that is element wise scaled by the scalar.

```
var typed_array = new Float32Array([0.5, 0.4, 0.3, 0.2, 0.1, 0.0]);

var cfg = _futhark_context_config_new();
var ctx = _futhark_context_new(cfg);

var copy = copyToHeap(typedArray);
var in_ptr = _futhark_new_f32_2d(ctx, copy, 3n, 2n);
Module._free(copy);

var out_ptr = Module._malloc(4);
_futhark_entry_scale(ctx, out_ptr, 0.5, in_ptr)
_futhark_free_f32_2d(ctx, in_ptr);

var ptr = HEAP32[out_ptr >> 2];
Module._free(out_ptr);

var shape_ptr = _futhark_shape_f32_2d(ctx, ptr) >> 3;
var shape = Array.from(HEAP64.subarray(shape_ptr, shape_ptr + 2));

var values_ptr = _futhark_values_raw_f32_2d(ctx, ptr) >> 2;
var length = Number(shape[0] * shape[1]);
var values = HEAPF32.subarray(values_ptr, values_ptr + length);

console.log(values, shape);
_futhark_free_f32_2d(ctx, ptr);

_futhark_context_free(ctx);
_futhark_context_config_free(cfg);
```

Listing 13: Working with raw Emscripten

The JavaScript is as tedious as the original C code with respect to manual memory management and, like in the previous example, needs to allocate and free memory on the Emscripten heap to retrieve the `out_ptr` output parameter. In this example the output parameter is an array pointer. Pointers default to 32-bit signed integers in WebAssembly.

Yet another complication is error handling. In listings 12 and 13 we omitted checking the return error code from the entry point functions and extracting the error message.

These pain points were motivating reasons for the design of the API. Working with the raw functions from Emscripten is only accessible to programmers with a strong understanding of memory in WebAssembly. This isn't reasonable as inter-operation between Futhark and JavaScript is a determining factor in whether Futhark is a good language for writing code to run in the browser. For Futhark to be practical the implementation details of the raw pointers to the heap should hidden by a layer of abstraction. This is precisely what the API designed in Chapter 4 accomplished.

In the following we will illustrate with examples how our Javascript API

wraps the WebAssembly module and JavaScript glue code generated by Emscripten.

The simplest case is when all entry point inputs and outputs are scalar as in the increment function. In this case the following FutharkContext class meets the specification of the API.

```
class FutharkContext {
  constructor() {
    this.cfg = _futhark_context_config_new();
    this.ctx = _futhark_context_new(this.cfg);
  }
  free() {
    _futhark_context_free(this.ctx);
    _futhark_context_config_free(this.cfg);
  }
  increment(in0) {
    var out0 = _malloc(4);
    _futhark_entry_increment(this.ctx, out0, in0);
    var result0 = HEAP32[out0 >> 2];
    _free(out0);
    return result0;
  }
}
```

Listing 14: FutharkContext class for the futhark increment function

The class has two fields `cfg` and `ctx`, that point to the config and context and are created in the constructor by calling functions from the C API. Observe how the increment methods successfully hides all the tedious details needed to call the underlying exported WebAssembly function `_futhark_entry_increment`. The reason we can do this is because it has access to the `ctx` class field. Without a class this was something that the programmer was forced to pass around. This is why the API was designed to access Futhark entry point functions as methods on the FutharkContext class. The other important benefit of the wrapper is that the results are simply returned instead of the cumbersome heap allocation and copying required to call the WebAssembly function.

5.2.2 FutharkArray

The FutharkArray class represents any array regardless of dimension and element type, somewhat like Python ndarrays. Listing 15 below shows our implementation.

```
class FutharkArray {
  constructor(ctx, ptr, type_name, dim, heap, fshape, fvalues, ffree) {
    this.ctx = ctx;
    this.ptr = ptr;
    this.type_name = type_name;
    this.dim = dim;
    this.heap = heap;
    this.fshape = fshape;
    this.fvalues = fvalues;
    this.ffree = ffree;
  }
  futharkType() { return this.type_name; }
  free() { this.ffree(this.ctx, this.ptr); }
  shape() {
    var s = this.fshape(this.ctx, this.ptr) >> 3;
    return Array.from(HEAP64.subarray(s, s + this.dim));
  }
  toTypedArray(dims = this.shape()) {
    console.assert(dims.length === this.dim);
    var length = Number(dims.reduce((a, b) => a * b));
    var v = this.fvalues(this.ctx, this.ptr) / this.heap.BYTES_PER_ELEMENT;
    return this.heap.subarray(v, v + length);
  }
  toArray() {
    var dims = this.shape();
    var ta = this.toTypedArray(dims);
    return (function nest(offs, ds) {
      var d0 = Number(ds[0]);
      if (ds.length === 1) {
        return Array.from(ta.subarray(offs, offs + d0));
      } else {
        var d1 = Number(ds[1]);
        return Array.from(Array(d0), (x,i) => nest(offs + i * d1, ds.slice(1)));
      }
    })(0, dims);
  }
}
```

Listing 15: Class FutharkArray

An instance is constructed with not only a pointer to the underlying Futhark array on the Emscripten heap, but also type information and the functions for manipulating the Futhark array itself. While this design choice leads to many fields and constructor arguments it has the advantage that we do not need to generate a class per array type.

The constructor and all the fields are effectively hidden from the API user in that they are never needed to use the API. In other languages these fields and methods would be private or otherwise hidden from the user. Unfortunately JavaScript doesn't provide information hiding facilities.

The fshape and fvalues functions are exported WebAssembly functions to access the shape and contents of the underlying Futhark array in the Emscripten heap. Similarly, the ffree function is the function to free the array. In the case of 2 dimensional `f32` arrays, as in the *scale.fut* example, these functions are `_futhark_shape_f32_2d`, `_futhark_values_raw_f32_2d`,

`_futhark_free_f32_2d`, as can be seen in line 17 in Listing 16 below, where the FutharkArray is constructed.

The heap argument in the constructor is the heap view corresponding to the array element type e.g `HEAPF32` corresponds to `f32`. The full mapping of the heap views to Futhark types can be seen in table 3.1. The `toTypedArray` method uses the heap view to return a heap subarray which is a view into the heap (no underlying memory is copied).

The *toArray* function calls *toTypedArray* to get the view of the underlying values. We construct a nested JavaScript array from the view, recursively.

```
1  class FutharkContext {
2    constructor() { ... }
3    free() { ... }
4    new_f32_2d_from_jsarray(array2d) {
5      return this.new_f32_2d(array2d.flat(), array2d.length, array2d[0].length);
6    }
7    new_f32_2d(array, d0, d1) {
8      console.assert(array.length === d0*d1);
9      var copy = _malloc(array.length << 2);
10     HEAPF32.set(array, copy >> 2);
11     var ptr = _futhark_new_f32_2d(this.ctx, copy, BigInt(d0), BigInt(d1));
12     _free(copy);
13     return this.new_f32_2d_from_ptr(ptr);
14   }
15   new_f32_2d_from_ptr(ptr) {
16     return new FutharkArray(this.ctx, ptr, 'f32', 2, HEAPF32,
17       _futhark_shape_f32_2d, _futhark_values_raw_f32_2d, _futhark_free_f32_2d);
18   }
19   scale(in0, in1) {
20     var out0 = _malloc(4);
21     var to_free = [];
22     if (in1 instanceof Array) {
23       in1 = this.new_f32_2d_from_jsarray(in1);
24       to_free.push(in1);
25     }
26     futhark_entry_scale(this.ctx, out0, in0, in1.ptr);
27     var result0 = this.new_f32_2d_from_ptr(HEAP32[out0 >> 2]);
28     _free(out0);
29     to_free.forEach(f => f.free());
30     return result0;
31   }
32 }
```

Listing 16: Class FutharkContext

Listing 16 gives the generated code for FutharkContext for the scale example. There are three new functions for creating FutharkArrays. The first two, `new_f32_2d_from_jsarray` and `new_f32_2d`, are those described in the API design in chapter 4 for creating FutharkArrays from JavaScript nested arrays and flat arrays. The third, `new_f32_2d_from_ptr` should be considered private and is not meant to be invoked by the API user. It creates a FutharkArray from a pointer to an underlying Futhark array on the Emscripten heap. The

`new_f32_2d_from_ptr` method is also the only place where we invoke the FutharkArray constructor and supply all its many arguments. It is used by the two API methods and also by the entry point method wrapper to wrap result arrays returned by the WebAssembly endpoint function.

The `new_f32_2d` method calls the underlying WebAssembly function `_futhark_new_f32_2d` to create an underlying Futhark array on the Emscripten heap. To pass the array input it must be copied to the Emscripten heap because `_futhark_new_f32_2d` expects a heap pointer. To do this we must first allocate space (in bytes, therefore the array length is multiplied by 4, the `f32` element byte size) on the heap with `_malloc`, which returns a location in the heap. Then the `HEAPF32` heap view is used to write into that location. Since `HEAPF32` addresses the heap in 4 byte `f32` chunks, the location needs to be divided by 4. After the call the allocated copy of the input is freed with `_free`.

The entry point method `scale` takes an array as its second argument and returns an array. For the input array, we designed the API to allow either passing in an already constructed FutharkArray or a JavaScript array. The first is useful when you have the FutharkArray from the output of an entry point, e.g. if you call scale twice:

```
fc.scale(2.0, fc.scale(0.5, [[1,2],[3,4]]));
```

It is also useful, to avoid unnecessary object construction, if you already have multidimensional data represented in a flat array or typed array and you know the shape. Then you can bundle this information efficiently in a FutharkArray with `new_f32_2d` before calling scale:

```
// array is a flat array representing a 2x2 matrix
var fa = fc.new_f32_2d(array, 2, 2);
fc.scale(0.5, fa);
```

To support calling with a JavaScript nested array argument, `scale` first tests whether the array argument is a JavaScript Array (not typed array). If it is it calls `new_f32_2d_from_jsarray` to create the FutharkArray and the underlying Futhark array on the Emscripten heap needed for the call to the WebAssembly entry point function. The API user never sees this FutharkArray and thus cannot free it and therefore this must be handled internally and we use `to_free` to remember that.

5.2.3 Opaques

As discussed in the API section we would like to wrap opaque inputs and return types so that users of the library aren't working with pointers. The only other important consideration for opaques in terms of implementation is that we also need a method for freeing their memory when they are no longer used. Listing 17 contains the implementation of the FutharkOpaque class.

```
class FutharkOpaque {
  constructor(ctx, ptr, ffree) {
    this.ctx = ctx; this.ptr = ptr; this.ffree = ffree;
  }
  free() { this.ffree(this.ctx, this.ptr); }
}
```

Listing 17: Class FutharkOpaque

Like FutharkArray, the API user does not need to call the FutharkOpaque constructor. It is only called by our internal entry point method wrapper code, namely to wrap opaque return values from the WebAssembly entry point function. The only method that is of importance to the API user is `free`.

5.2.4 Error Handling

Up until now we have assumed that the entry point functions have run without issue. Issues can arise when the arguments passed into an entry point function don't have matching dimensions.

Lets take a look at a simple implementation of dot product.

```
entry dotprod [n] (xs: [n]i32) (ys: [n]i32): i32 =
  reduce (+) 0 (map2 (*) xs ys)
```

Listing 18: A Simple dot product Implementation

The `n` in the type signature of the method tells the compiler that the sizes of xs and ys are the same. If we were to compile this into a C library and call the entry point function with two futhark arrays of different lengths, the function would return an error code of 1. At this stage we can find the error message by calling `futhark_context_get_error()`. The function takes a pointer to a futhark context as input and returns a string with the error message.

In JavaScript it is common practice to throw an exception to exit a function that encounters an error. To implement this we first add a method `get_error()` to the FutharkContext class:

```
get_error() {
  var ptr = _futhark_context_get_error(this.ctx);
  var end = HEAP8.indexOf(0, ptr);
  var str = String.fromCharCode(...HEAP8.subarray(ptr, end));
  _free(ptr);
  return str;
}
```

Listing 19: Helper function for reading error message off the Context

This function conveniently gets a view of the subarray from `ptr` to the terminating zero character, and converts it into a JavaScript string and returns said string. The C API requires the caller to free the error string. We can then combine `get_error` with our calling of the WebAssembly entry point function. Listing 20 shows this for the scale.fut library:

```
1   if (_futhark_entry_scale(this.ctx, out, in0, in1.ptr) > 0) {
2     _free(out0);
3     to_free.forEach(f => f.free());
4     throw this.get_error();
5   }
```

Listing 20: Error handling for scale

If the result is greater than 0 we know we have encountered an error, in which case we also free all the memory that has been allocated in the current run. And finally we throw the error by calling the `get_error` function we defined in listing 19. This is all that is required to get the Futhark error reporting facilities provided in C ported over to JavaScript.

5.2.5 Remaining details

Futhark entry points can return a single value or multiple values. For our implementation if there is a single value we just return the value and if there are multiple return values, we return an array of values.

The code generation from the compiler varies slightly from the code snippets presented. This is because the code generation needs to work for an arbitrarily number of input arguments and return values. For ease of explanation we tried to omit the aspects of the code generation that weren't specifically insightful. Even though our code is generated by the compiler, we put an emphasis on having a readable implementation. The big browser vendors provide tools for developers to interactively debug their code in the browser. For this reason it is practical to produce legible code.

For illustration, consider the following function (an artificial example to exercise different scalar, array, and opaque types):

```
entry tuple (b: bool) (f: f64) (u: u16) (i: i64) (m: [][]f32)
  = (b, f, (u, i), m[0])
```

The generated JavaScript code to wrap the WebAssembly entry point function in a FutharkContext entry point method is shown in Listing 21. The code is somewhat readable, given the inherently complexity of the tasks it accomplishes.

```
tuple(in0, in1, in2, in3, in4) {
  var out = [1, 8, 4, 4].map(n => _malloc(n));
  var to_free = [];
  var do_free = () => { out.forEach(_free); to_free.forEach(f => f.free()); };
  if (in4 instanceof Array) {
    in4 = this.new_f32_2d_from_jsarray(in4); to_free.push(in4);
  }
  if (_futhark_entry_tuple(this.ctx, ...out, in0, in1, in2, in3, in4.ptr) > 0) {
    do_free();
    throw this.get_error();
  }
  var result0 = HEAP8[out[0] >> 0]!==0;
  var result1 = HEAPF64[out[1] >> 3];
  var result2 = new FutharkOpaque(this.ctx, HEAP32[out[2] >> 2],
                                  _futhark_free_opaque_8021f38c);
  var result3 = this.new_f32_1d_from_ptr(HEAP32[out[3] >> 2]);
  do_free();
  return [result0, result1, result2, result3];
}
```

Listing 21: tuple entry point method

5.3 Compiler Pipeline

When putting all these pieces together and adding the modifications in the compiler an important implementation observation is that the API only depends on 3 things. The name of Futhark's C library functions, these functions' return types, and these functions' arguments types. This greatly simplifies the implementation. The function names, return types, and argument types are needed for generating the runtime classes, and exporting the functions to the emcc command. With this approach we only depend on Futhark's C API, which is a part of the compiler that is most stable and therefore will rarely have to be adapted to deal with code changes in the intermediate representation of the compiler.

Emscripten provides facilities for combining the JavaScript glue code with a library with the `--post-fix` and `--js-library` flags. This is convenient as it reduces the number of files that are produced from running the `futhark wasm` command.

Figure 5.1 illustrates the logical flow of the compilation of the WebAssembly backend.

The source program is turned into Futhark's intermediate representation, and then into C source code. Our JavaScript API classes that the WebAssembly backend generates only depend on the function names, argument types, and return types, which can be taken from the intermediate representation. At this stage both the C source code and the API are joined together with the EMCC command, generating glue code with the attached API classes, and a WebAssembly module.

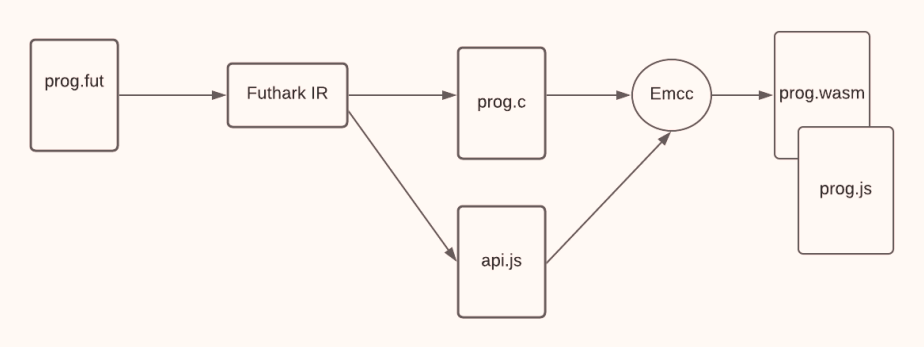


Figure 5.1: Sequential WebAssembly backend compiler pipeline

```
EMCFLAGS="-s INITIAL_MEMORY=$((16777216 * 4))" futhark wasm --lib prog.fut
```

Listing 22: Example futhark wasm compile command

5.3.1 Emscripten Compiler Flags

The Emscripten compiler provides a large number of flags that effect the usage and the performance of the generated WebAssembly code. The backend implementation uses only a few flags. The WebAssembly backend implementation includes a facility for the user to specify further compiler flags to pass emcc, via the EMCFLAGS environment variable. This way the users of the WebAssembly backend can add the flags that are best suited for their applications. As discussed previously in order for 64-bit integers to be handled correctly, the backend needs to use the `WASM_BIGINT` flag. The other flag that it defaults to is the O3 optimization level. Though this obfuscates the resulting code, it comes with sizeable performance improvements over the other optimization levels. Most browsers will have means to automatically pretty print the obfuscated code.

The most important flag for users of the WebAssembly backend to consider is the `INITIAL_MEMORY` flag. The memory flag in Emscripten defaults to 16 megabytes, but with performance heavy computation it is likely that this limit will be exceeded. In these cases the user should manually set the memory based on their needs. It is recommended to use less than 2 gigabytes as only recently have virtual machines started to allow more than 2 gigabytes of heap space. This means that using more memory than 2 gigabytes will likely not be portable across browsers and with different WebAssembly engines.

Listing 22 shows an example of calling the backend with an additional compiler flag to set the memory limit higher. The additional compiler flags to Emscripten are passed through the environment variable EMCFLAGS. In this case the memory is increased to 64 megabytes.

5.4 Application

With the WebAssembly backend implemented as described above, it can now be seen in action. One of the applications for high performance computing in the browser is graphics. Figure 5.2 illustrates a Futhark program running in the browser for visualizing the Mandelbrot set.

The full implementation of *mandelbrot.fut* and can be found in the appendix A. Taking a look at the function signature:

```
let main (screenX: i64) (screenY: i64)
         (depth: i32) (xmin: f32)
         (ymin: f32) (xmax: f32)
         (ymax: f32): [screenY][screenX]i32 =
```

We note that the entry point function takes 7 input arguments. The first two arguments `screenX` and `screenY` specify the dimensions of the output.

We firstly compile the *mandelbrot.fut* file as a library:

```
futhark wasm --lib mandelbrot.fut
```

This generates the files *mandelbrot.js* and *mandelbrot.wasm*. At this stage we can write a HTML file to call the JavaScript API.

Listing 23 is a web page that takes 7 inputs from the user, one for each of the arguments in the *mandelbrot.fut* entry point function. The user can then click a button, which will then run the Mandelbrot computation and visualization by calling the Futhark function.

In order to run the code in the browser, we need to launch a web server. This can be done from Python with

```
python -m http.server
```

After running the server, we can go the web page in any modern browser. Put in inputs and then press the button. Once the mandelbrot computation is complete the result will render in the page.

Figure 5.2 shows the visualization once the function is done executing. The example illustrates how Futhark code can be compiled into a WebAssembly module that can be called in the browser. The WebAssembly module offload the computationally heavy workload from the less efficient JavaScript execution engine. The only Futhark specific code in listing 23 are lines 27-30:

```
var instance = await createFutharkModule();
var fc = new instance.FutharkContext();
var result = fc.main(screenX, screenY, depth, xmin, ymin, xmax, ymax);
var vals = result.toTypedArray();
```

The first line loads the WebAssembly module. The second line instantiates the context. The third line runs the entry point function. And finally the fourth

```
<!doctype html>
<html>
  <label for="mandelbrot">Numbers for mandelbrot set!</label><br>
  <input type="text" value="800" id="screenX" name="screenX"><br>
  <input type="text" value="600" id="screenY" name="screenY"><br>
  <input type="text" value="255" id="depth" name="depth"><br>
  <input type="text" value="-2.23" id="xmin" name="xmin"><br>
  <input type="text" value="-1.15" id="ymin" name="ymin"><br>
  <input type="text" value=".83" id="xmax" name="xmax"><br>
  <input type="text" value="1.15" id="ymax" name="ymax"><br>
  <button id="action" onclick="myFunction()">Run Mandelbrot!</button>
  <p>
  <canvas id="canvas"></canvas>
  </p>
  <script src="mandelbrot.js"></script>
  <script>
    async function myFunction() {
      // Get variables from input fields
      var screenX = BigInt(parseInt(document.getElementById("screenX").value));
      var screenY = BigInt(parseInt(document.getElementById("screenY").value));
      var depth = parseInt(document.getElementById("depth").value);
      var xmin = parseFloat(document.getElementById("xmin").value);
      var ymin = parseFloat(document.getElementById("ymin").value);
      var xmax = parseFloat(document.getElementById("xmax").value);
      var ymax = parseFloat(document.getElementById("ymax").value);
      // Call Futhark
      var instance = await createFutharkModule();
      var fc = new instance.FutharkContext();
      var result = fc.main(screenX, screenY, depth, xmin, ymin, xmax, ymax);
      var vals = result.toTypedArray();
      // Set pixels for canvas
      var data = new Uint8ClampedArray(vals.length * 4);
      for (var i = 0; i < vals.length; i++) {
        data[4*i+0] = (vals[i] & 0xFF0000) >> 16;
        data[4*i+1] = (vals[i] & 0xFF00) >> 8
        data[4*i+2] = (vals[i] & 0xFF)
        data[4*i+3] = 255;
      }
      result.free();
      fc.free();
      // Make canvas and ctx
      var canvas = document.getElementById('canvas');
      canvas.width = Number(screenX);
      canvas.height = Number(screenY);
      var ctx = canvas.getContext('2d');
      var imgdata = new ImageData(data, Number(screenX), Number(screenY));
      ctx.putImageData(imgdata, 0, 0);
    }
  </script>
</html>
```

Listing 23: HTML file for calling *mandelbrot.js*

line converts the result FutharkArray to a standard JavaScript typed array. Observe that the user of this library doesn't need to be aware that there is any WebAssembly under the covers.

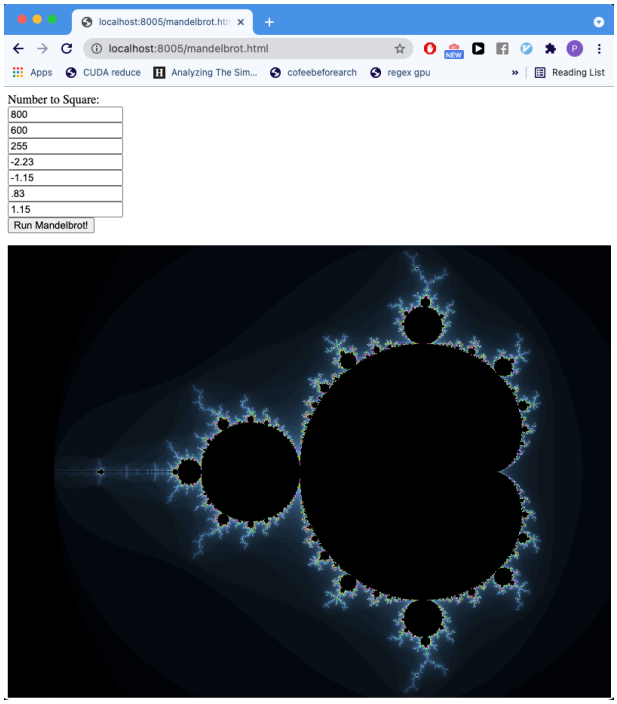


Figure 5.2: Caption

5.5 Benchmarking

The WebAssembly backend is benchmarked against the C backend to see how competitive the execution speed is. The WebAssembly backend is both benchmarked running in the browser with Chrome as well as running with Node.js.

- Chrome: benchmarking WebAssembly in Chrome gives details on how WebAssembly performs in the browser, which is likely where the backend will be deployed in practice.

- Node: benchmarking with Node.js gives details into how the WebAssembly preforms when run as a backend language. This is interesting as it gives details into the WebAssembly's performance for use cases outside of the browser.

The Futhark programs that are benchmarked come from the Futhark benchmark suite, which are used for standardized benchmarks to test the performance of all Futhark backends against industry standards. What is interesting to look at is how the WebAssembly performs relative to the C backend it is built on top of.

Suite	Dataset	Size	C	WebAssembly	Chrome
Accelerate	Tunnel	1000	735 ms	1,219 ms	1217
		2000	2,942 ms	4,889 ms	4,827 ms
		4000	11,762 ms	19,693 ms	19,302 ms

Table 5.1: Caption

The Tunnel Benchmark shows that the WebAssembly backend takes approximately 65% performance penalty for running the benchmarks with WebAssembly relative to the sequential C backend. This performance penalty is consistent as the dataset increases, which means that it is not a constant overhead of launching or instantiating WebAssembly. WebAssembly when run in Chrome and in Node.js have nearly identical performance, with the difference never exceeding 2 percentage points.

Suite	Dataset	Size	C	WebAssembly	Chrome
Accelerate	Mandelbrot	1000	131 ms	149 ms	155 ms
		2000	530 ms	602 ms	601 ms
		4000	2,117 ms	2,403 ms	2446 ms

Table 5.2: Caption

For the mandelbrot benchmarks the WebAssembly backend has more competitive execution speeds. The WebAssembly backend is consistently 13-14% slower across all the sizes. Similarly to the tunnel benchmark, the relative performance difference of the C backend and the WebAssembly does not change with respect to the dataset sizes.

5.6 Testing

We validate our implementation against the Futhark test suite. It consists of over 1500 test programs. We also use Node.js to run the tests locally. This is because testing WebAssembly in the browser requires a fair deal of extra machinery, and we cannot tap into Futhark's existing test framework.

We have both validated our implementation when compiled as a server and as an executable. Before the Futhark test infrastructure changed, we validated our backend when compiled as an executable. This required small tweaks to handle UTF-8 decoding issues with Node.js and Emscripten. After the testing infrastructure changed to test Futhark programs in server mode. This required implementing the Futhark server protocol in JavaScript, as well a functionality for reading and writing data from Futhark's binary format.

CHAPTER 6

Parallel Execution in the Browser

Browsers have facilities for parallel programming. Javascript supports two different paradigms with web workers. Message passing enables parallel programming without shared memory. SharedArrayBuffer and atomics enable shared memory multithreading with thread synchronization. There is a threaded WebAssembly proposal that adds atomic operations to the language, and adds support for SharedArrayBuffers while relying on JavaScript's web workers to create and join threads. This chapter introduces all these concepts and illustrates them with examples.

6.1 Web Workers

Parallelism with JavaScript in browsers is achieved through web workers. Web workers are extra threads of execution beyond the main thread. The threads interact via message passing. Typically messages are passed through the postMessage and onmessage. postMessage is used to send a message between threads and onmessage works as an event handler to receive messages from threads.

Web workers are relatively heavyweight, and should not be created in large numbers. They are expected to be long lived and have both high start and high per instance memory cost [17].

The following example computes the Riemann integral of sine over an interval from 0. The interval is broken up into subintervals which are computed by separate workers.

```
1  var num_workers = 4;
2  var result = 0;
3  var counter = 0;
4  workers = [];
5
6  for (var i = 0; i < num_workers; i++) {
7    workers.push(new Worker('worker.js'));
8  }
9
10 for (var i = 0; i < num_workers; i++) {
11   workers[i].onmessage = function (event) {
12     counter += 1;
13     result += event.data;
14     if (counter == num_workers) console.log(result);
15   }
16 }
17
18 for (var i = 0; i < num_workers; i++) {
19   workers[i].postMessage(i);
20 }
```

Listing 24: Main file that calls workers which handle the computation of Riemann integral

The example code in Listing 24 spawns 4 worker threads in lines 5-7. It sends each thread a message with their respective index in lines 18-20. It asynchronously waits for messages from each of the worker threads with their partial result and prints the final result when all the threads have sent a message in lines 10-16.

```
1  var GRANULARITY = 1000000000;
2  var NUM_THREADS = 4;
3  var interval = 3.14;
4
5  onmessage = function (event) {
6    var index = event.data;
7    var bottom = index * (interval / NUM_THREADS);
8    var upper = bottom + (interval / NUM_THREADS);
9    var sum = 0;
10   for (var i = 0; i < GRANULARITY;  i++) {
11     var x = bottom + (upper - bottom) / GRANULARITY * i;
12     sum += Math.sin(x);
13   }
14   var res = sum / GRANULARITY;
15   postMessage(res);
16 }
```

Listing 25: Worker thread logic for computing Riemann integral

The code in Listing 25 contains the implementation of the worker threads. Once the thread receives a message from the main thread with their index, they compute the partial Riemann integral over their respective quartile of the interval adding the value of sine(x) as many times as specified by granularity. Figure 6.1 shows the execution time of the code against a different number of web workers.

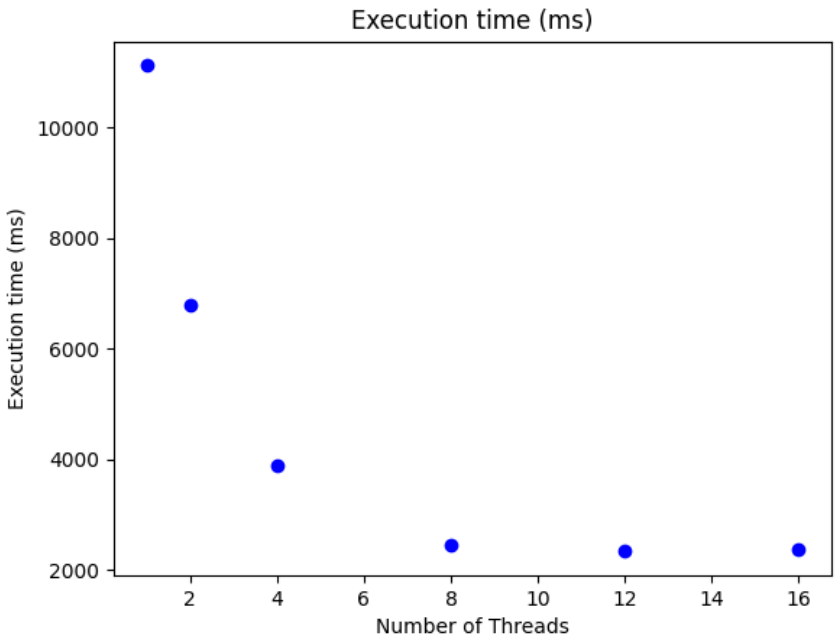


Figure 6.1: Execution time of Riemann integration for different thread counts. Run on a Macbook Pro with 2,2 GHz 6-Core Intel Core i7

The code execution time in 6.1 is for the Reimann integral computed with one billion sample values. For one worker thread the execution time was 11.11 seconds. For two threads the execution time was 6.8 seconds, which is nearly double as fast. However as the number of threads increases the increase in execution speed tapers off. Going from 8 to 12 cores only yields a marginal increase in execution speed going from 2.45 second to 2.39. Once more than twelve worker threads are exceeded, there is no longer a marginal increase in speedup. This can be attributed to the physical limitation of threads on the hardware the code was executed on. The number of logical cores on the computer that executed this code was 12, meaning that any additional web worker launched after the initial 12 must wait on the thread pool for another to finish before it can be activated. In which case the overhead of launching it is only detrimental to the complete execution time of the program

6.2 Shared Memory and Atomics

Web workers with message passing have some similarities in how parallelism is executed with the Erlang programming language. Both of which use message passing to coordinate parallel execution. However many other programming

```
var num_workers = 9;
var n = 100;
var signal = new SharedArrayBuffer(2 * 4);
var elements = new SharedArrayBuffer(n*4);
var arr = new Int32Array(elements);
for (var i = 0; i < n; i++) {
 arr[i] = i;
}
var prefix_sum = new SharedArrayBuffer(n * 4);
workers = [];

for (var i = 0; i < num_workers; i++) {
  workers.push(new Worker('prefix.js'));
}
var counter = 0;
for (var i = 0; i < num_workers; i++) {
  workers[i].onmessage = function (event) {
    counter += 1;
    if (counter === num_workers) {
      console.log(new Int32Array(prefix_sum));
    }
  }
}
for (var i = 0; i < num_workers; i++) {
  workers[i].postMessage({index : i,
                          num_workers,
                          signal,
                          elements,
                          prefix_sum
                          });
}
```

Listing 26: Main file that calls workers which compute prefix sum using shared memory and atomics in parallel

languages and libraries also support and utilize shared memory. An example of this is C/C++ and POSIX threads. Shared memory maps closely to modern multicore hardware, and is faster for workloads that cannot efficiently partition memory for mutually exclusive access from different threads of execution. However it comes at the cost of a new set of bugs in the shape of data races, which is why languages such as Erlang and Futhark itself abstracts the construct away from the programmer. JavaScript also offers shared memory through SharedArrayBuffers. A SharedArrayBuffer points to a piece of linear memory. The SharedArrayBuffer can be passed to multiple web workers who can access the memory in parallel.

In principle safe access to shared memory can be coordinated with message passing, but it's far more efficient for fine grained synchronization to use atomic operations, which again map efficiently to the underlying hardware. Atomic operations make sure that predictable values are written and read, that operations are finished before the next operation starts and that operations are not interrupted [11]. The Atomics package in JavaScript contains functions for performing atomic operations on SharedArrayBuffers. The

Atomics package also includes wait and notify functions, like linux futex, wait, and wake.

To illustrate shared memory and atomics the following example is an implementation of prefix sum. It is a fundamental parallel algorithm which is

```
onmessage = function (event) {
  var i = event.data.index;
  var num_workers = event.data.num_workers;
  var signal = new Int32Array(event.data.signal);
  var elements = new Int32Array(event.data.elements);
  var prefix_sum = new Int32Array(event.data.prefix_sum);
  var n = elements.length;
  var partitions = num_workers + 1;
  var stride = n / partitions;

  var prefix_sum_partition = function (idx, offset) {
    prefix_sum[idx * stride] = elements[idx * stride] + offset;
    for (var j = idx * stride + 1; j < (idx+1) * stride; j++) {
      prefix_sum[j] = prefix_sum[j-1] + elements[j];
    }
  }

  // First pass of algorithm
  prefix_sum_partition(i, 0);

  // Synchronization logic between workers
  if (i === 0) {
    // wait for other num_workers to finish
    var finished = 0;
    while (finished < num_workers - 1) {
      Atomics.wait(signal, 0, finished);
      finished = signal[0];
    }
    // Calculate the cumulative sums of the partitions
    var x = 0;
    for (var j = 1; j < partitions; j++) {
      x += prefix_sum[j * stride - 1];
      prefix_sum[j * stride] = x;
    }
    // notify other num_workers to restart
    Atomics.store(signal, 1, 1);
    Atomics.notify(signal, 1, num_workers-1);
  } else {
    // notify worker 0 that we are finished
    Atomics.add(signal, 0, 1);
    Atomics.notify(signal, 0, 1);
    // wait for worker 0
    while (Atomics.wait(signal, 1, 0) != "not-equal") {}
  }

  // Second pass of algorithm
  prefix_sum_partition(i+1, prefix_sum[(i+1) * stride]);

  postMessage("Done!");
}
```

Listing 27: Worker file for computing the prefix sum using shared memory and atomics.

used as a building block for many other parallel algorithms. The example implements a shared memory 2 pass algorithm [1].

The code in Listing 26 spawns 9 worker threads. It sends a message to each of the threads with parameters, some of which are shared array buffers. This allows each of the threads to have access to shared memory. When each thread has sent a message to indicate completion, the final result of prefix sum is logged to the console.

The code in Listing 27 handles the actual execution of prefix sum. In the first pass of the algorithm, each thread calculates the prefix sum of their partition in the array, by calling the function `prefix_sum_partition`. Each thread signals that they are done with their work in the first pass by using the Atomics add, and notify functions. The first thread detects signal has accumulated a response of `num_workers - 1`, by using the Atomics function wait. At this point the the first thread calculates the cumulative sums of partitions. At this stage it notifies the other other threads using the Atomics store and notify. At this stage all threads calculate the final `prefix_sum`, using the precomputed values. On completion each thread sends a message back to the the main file using the postMessage to indicate they are done.

6.3 Threaded WebAssembly

There is a proposal to extend the WebAssembly specification with support for threads, namely by leveraging web workers, shared memory, and atomics. Chrome and Firefox and Node.js all have experimental support for threaded WebAssembly. Emscripten supports compilation of C/C++ with pthreads to threaded WebAssembly.

Threaded WebAssembly uses web workers to create and join threads. It doesn't natively invoke web workers but instead handles this by calling out to JavaScript. Shared memory is accomplished by integrating SharedArrayBuffer with WebAssembly's paged memory model. WebAssembly is extended with atomic operation instructions. Putting it concisely the additions of supporting shared array buffers in WebAssembly and adding atomic operations in WebAssembly was all that was needed to facilitate threaded WebAssembly.

A key observation is that WebAssembly does not natively allow for spawning of threads. This is actually taken care of by the runtime or compiler. Specifically for Emscripten, compiling C code written with pthreads will generate three files. It will generate a WebAssembly file, and and two Javascript files. One for the main glue code and other for worker glue code. The glue code takes care of loading the WebAssembly module, populating the memory with the required values, and integrating with the host system as the C code would expect. The C function pthread_create is translated to Javascript and not WebAssembly. It launches a Javascript Worker, passing it a shared array buffer and the wasm module that it should run. The WebAssembly simply

needs the shared array buffer and atomics to synchronize.

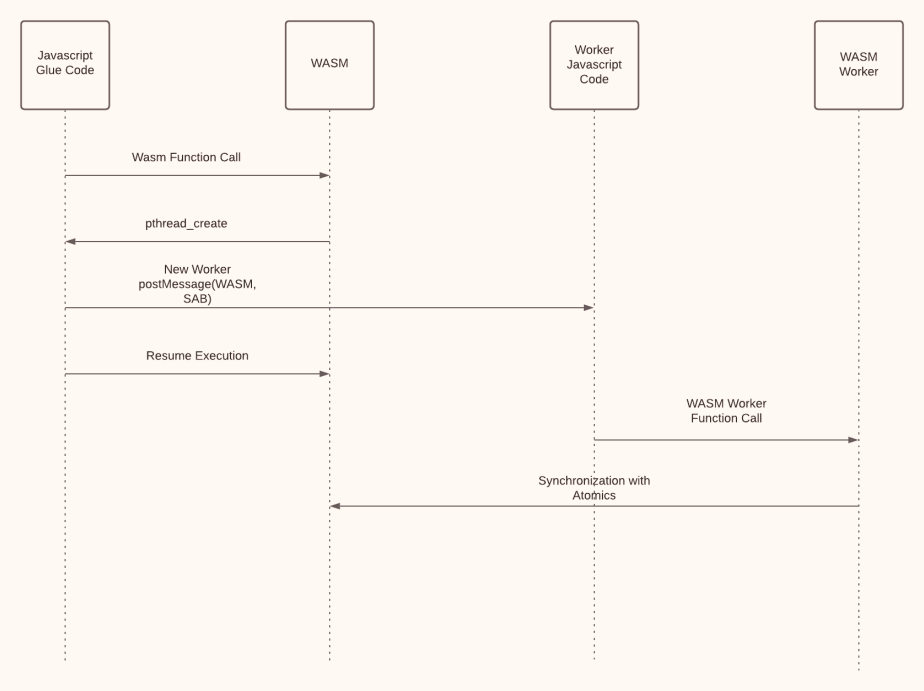


Figure 6.2: UML diagram showing flow of execution of threaded programs in WebAssembly

The UML diagram in figure 6.2 demonstrates the execution flow of a parallel function written with pthreads in C, compiled by Emscripten, and run in Javascript and WebAssembly. The javascript glue code calls the parallel WebAssembly function, which then calls an external Javascript function that is used to emulate the functionality of `pthread_create`. This function launches a new worker, sending it a message with the WebAssembly module, and a shared array buffer, and resumes execution. This JavaScript worker code then instantiates the WebAssembly module, calling the designated WebAssembly module function. And then with these multiple WebAssembly modules running in parallel, they use atomic instructions native to WebAssembly to facilitate synchronization, as specified in the program.

CHAPTER 7

WebAssembly Multicore Backend

This chapter details the extensions that are added to the Futhark compiler to support a multicore WebAssembly backend. It also benchmarks the generated WebAssembly-multicore code against the sequential WebAssembly backend. It also compares the multicore C backend against the WebAssembly-multicore backend running in the browser.

Fortunately only small adaptions had to be made to the WebAssembly backend developed earlier, to get it running with Multicore. The Futhark compiler has a backend that generates both Sequential C code as well as a backend that generates multicore C code using POSIX threads. As discussed Emscripten can translate multicore C code that uses POSIX threads to multicore WebAssembly that can run in parallel in the browser. The JavaScript API developed in chapter 4 stays most unchanged, with only one small modification for the WebAssembly-multicore backend.

Though this thesis adds 2 backends to the 6 backends already present in the Futhark compiler (4 C backends, and 2 Python backends), the added complexity is relatively modest because of the high degree of reuse of code between the two WebAssembly backends.

7.1 Implementation Structure

Below we discuss how to add a new futhark backend that can be invoked from the command line with `futhark wasm-multicore`. It is structured very similarly to the plain WebAssembly backend described chapter 4. Instead of calling Emscripten on the Sequential C, we apply it to Futhark's multicore C backend. We utilize the JavaScript and WebAssembly runtime code written in chapter 4, for the backend and add the necessary Emscripten compiler flags required to enable Multicore WebAssembly. Figure 7.1 illustrates the structure of the WebAssembly multicore implementation.

One of the key differences that can be seen in the figure 7.1 is that the wasm-multicore backend produces 2 JavaScript files and 1 WebAssembly file

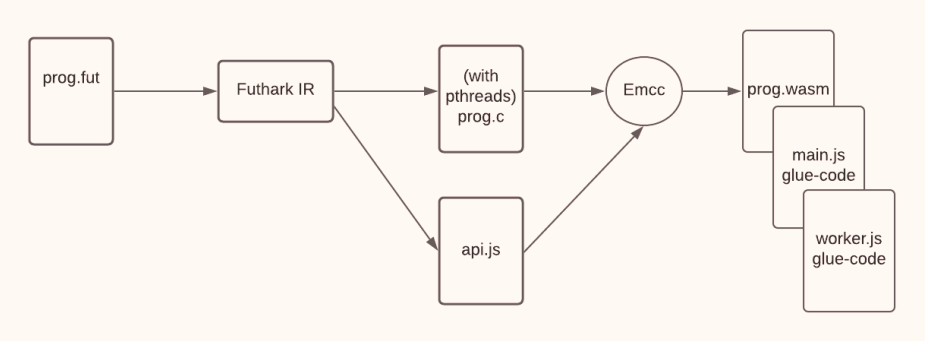


Figure 7.1: WebAssembly-multicore compilation

as opposed to the Sequential Wasm backend which generates 1 JavaScript file and 1 WebAssembly file. The second JavaScript file is the web worker glue code.

7.2 Implementation Details

Here we discuss the series of steps that were needed complete the implementation.

7.2.1 Plumbing

We simply combine the multicore C compiler with the JavaScript code generation discussed in chapter 4, by calling the respective functions that contain the meat of the logic, which have already been developed. The last thing of note is that the flag `-pthread` is passed to `runEMCC`. This flag lets Emcc know that our C code contains pthreads, and thereby compiles it correctly.

7.2.2 Multicore C code changes

The pthread C code generated by the multicore C backend used platform specific implementations in the generated function `getrusage_thread` and `num_processors`. For `getrusage_thread` it was possible to reuse the linux implementation. For `num_processors` it was necessary to add an additional implementation for the Emscripten platform. This was simply done by including *emscripten/threading.h* and calling the function `emscripten_num_logical_cores`.

7.2.3 API Change

The multicore C backend exports one important additional function for setting the number of threads the library will use:

```
void futhark_context_config_set_num_threads(
     struct futhark_context_config *cfg, int n);
```

`futhark_context_config_set_num_threads` does this by taking threads and configuration as argument and then number of threads that the library will use on the configuration. This information we would like to have when running our library. We adapt our API to pass an an optional argument to the FutharkContext constructor:

```
class FutharkContext {
    constructor(num_threads) {
      this.cfg = _futhark_context_config_new();
      if (num_threads) _futhark_context_config_set_num_threads(this.cfg, num_threads);
      this.ctx = _futhark_context_new(this.cfg);
    }

}
```

Listing 28: FutharkContext with optional argument

In the constructor in listing 28 we simply set the number threads to be the argument given in the FutharkClass constructor. If no argument is given, it defaults to `emscripten_num_logical_cores`. Surprisingly this was the only adaption we needed to make in the API described in chapter 4.

7.2.4 Emscripten Invocation

The generated multicore C code uses the POSIX function `pthread_create`. Emscripten aims to follow the POSIX standard closely, but in some places has slightly different behaviour. This is the case for `pthread_create`, which is a function that is used in the generated multicore C code.

When pthread_create() is called, if we need to create a new Web Worker, then that requires returning the main event loop. That is, you cannot call `pthread_create` and then keep running code synchronously that expects the worker to start running - it will only run after you return to the event loop [13]. In order to work around the API differences, the compiler flag `PTHREAD_POOL_SIZE=<integer>` needs to be passed to the Emscripten compiler. This effectively creates the web workers before the main thread is called, in which case `create_pthread` can just use an already spawned web worker.

7.2.5 Running in browser with HTTP

Running threaded WebAssembly in the browser is slightly more involved than running standard WebAssembly. This is because threaded WebAssembly uses SharedArrayBuffers. SharedArrayBuffers introduce a few security vulnerabilities. For this reason browsers require us to set to HTTP headers when we run our web server. We need to set the flags as follows:

- Cross-Origin-Embedder-Policy (COEP) : require-corp
- Cross-Origin-Opener-Policy (COOP) : same-origin

The following listing 29 file can be used to run a web server with COEP and COOP headers set:

We can run this Python file, to run threaded WebAssembly programs locally.

```
#!/usr/bin/env python3
from http import server # Python 3

class MyHTTPRequestHandler(server.SimpleHTTPRequestHandler):
    def end_headers(self):
        self.send_my_headers()
        server.SimpleHTTPRequestHandler.end_headers(self)

    def send_my_headers(self):
        self.send_header("Cross-Origin-Embedder-Policy", "require-corp")
        self.send_header("Cross-Origin-Opener-Policy", "same-origin")

if __name__ == '__main__':
    server.test(HandlerClass=MyHTTPRequestHandler)
```

Listing 29: Python server implementation for setting COEP and COOP HTTP Headers

7.2.6 Applications

In order to see the WebAssembly multicore code in action, we will use ray tracing as a motivating example. Ray tracing is a technique in graphics, which is used to simulate how rays of light bounce off objects. Ray tracing can be used to create realistic, high definition pictures. Its a common technique used in rendering and movies.

We are mainly concerned with the arguments and return type of the entry point function. The entry point for *raytracer.fut* is defined as follows:

```
let main (nx: i64) (ny: i64)
         (ns: i32) (nobj: i32)
         : [ny][nx]argb.colour
```

The function takes **nx** and **ny** specifying the number of pixels we want our ray tracer to cover. This **ns** specifies the number of samples we want per pixel. Lastly nobj specifies the number of reflections per ray. The return type is a grid of `u32` numbers.

We can can compile the WebAssembly library with the following command

```
EMCFLAGS="-s INITIAL_MEMORY=2147418112 -s PTHREAD_POOL_SIZE=12" \
                        futhark wasm-multicore --lib raytracer.fut
```

```
<!doctype html>
<html>
  <button id="action" onclick="myFunction()">Run RayTracer!</button>
  <p>
  <canvas id="canvas"></canvas>
  </p>
  <script src="raytracer.js"></script>
  <script>
    async function myFunction() {
      var screenX = 400n
      var screenY = 400n
      var ns = 50;
      var nobj = 5;
      var instance = await createFutharkModule();
      var fc = new instance.FutharkContext(12);
      // Call Futhark
      var result = fc.main(screenX, screenY, ns, nobj);
      // Get javascript values from futhark
      var vals = result.toTypedArray();
      // Set pixels for canvas
      var data = new Uint8ClampedArray(vals.length * 4);
      for (var i = 0; i < vals.length; i++) {
        data[4*i+0] = (vals[i] & 0xFF0000) >> 16;
        data[4*i+1] = (vals[i] & 0xFF00) >> 8
        data[4*i+2] = (vals[i] & 0xFF)
        data[4*i+3] = 255;
      }
      // Make canvas and ctx
      var canvas = document.getElementById('canvas');
      canvas.width = Number(screenX);
      canvas.height = Number(screenY);
      var ctx = canvas.getContext('2d');
      var imgdata = new ImageData(data, Number(screenX), Number(screenY));
      ctx.putImageData(imgdata, 0, 0);
      result.free();
      fc.free();

    }
  </script>
</html>
```

Listing 30: HTML file for creating ray trace visualization

With this command we are giving WebAssembly access to approximately 2 gigabytes and setting the number of threads to be run simaltaniusly to 12. The result of this command will be three files: *raytracer.js*, *raytracer.worker.js*, and *raytracer.wasm*. Listing 30 shows an example HTML page that calls the compiled Futhark library.

The web page has a button, which runs a function. This function loads the Futhark library, and instantiates FutharkContext. It then calls the Futhark entry point with the arguments 400, 400, 50, and 5 for **nx**, **ny**, **ns**, and **nobj** respectively. It proceeds to display the result on the page. With this HTML file, compiled Futhark library, and the Python server file from 29 we can run the code.

We simply run the Python file (it launches a web server with the appropriate configuration), and then go to `localhost:8000`. Figure 7.2 shows the state of the web page after the button has been clicked and it has finished executing.

Figure 7.2: Ray Tracer Web Page

On average 40 percent of visitors to a web page leave if the page hasn't loaded in 3 seconds [3]. Ray tracing in the browser in the browser gives a 6 times speed up over running the sequential WASM. The web page took 2.955 seconds to render the ray tracing when running on a Macbook Pro with 2,2 GHz Intel Core i7 with 6 physical cores and 12 logical cores. In comparison running the Sequential WebAssembly on the same 400 by 400 pixels, 50 sample points and 5 object reflections took 17.74 seconds.

7.3 Benchmark

We start by benchmarking against the familiar *mandelbrot.fut* program from chapter 5. This time we benchmark the multicore WebAssembly in both Chrome and Node.js against the multicore C backend. We use 12 threads for each of the benchmark runs. The results can be seen in table 7.1. Again these experiments are performed on a Macbook Pro with 2,2 GHz Intel Core i7 with 6 physical cores and 12 logical cores.

Suite	Dataset	Size	mc C	mc Node.js	mc Chrome
Accelerate	Mandelbrot	1000	14.7 ms	39 ms	27 ms
		2000	53 ms	103 ms	104 ms
		4000	211 ms	412 ms	432 ms

Table 7.1: Mandelbrot benchmarks

Interestingly running in WebAssembly backend in Node.js and Chrome takes about double as long as the multicore C backend for the mandelbrot data. On further inspection when comparing the data to the data from the original WebAssembly backend implementation, we only have a 6 times speed up. Where the multicore has a 12 times speed up. The table 7.2 is the same table as 5.2 from Chapter 5, copied here for convenience.

Suite	Dataset	Size	seq C	seq WebAssembly	seq Chrome
Accelerate	Mandelbrot	1000	131 ms	149 ms	155 ms
		2000	530 ms	602 ms	601 ms
		4000	2,117 ms	2,403 ms	2446 ms

Table 7.2: Benchmark results for sequential backends on mandelbrot dataset

When running the the raytracing example in 7.2 there was similarly only a 6 times speed up even though ray tracing can be perfectly parallelized. This motivates us to see the performance change of the multicore WebAssembly backend with respect to the number of web workers being used.

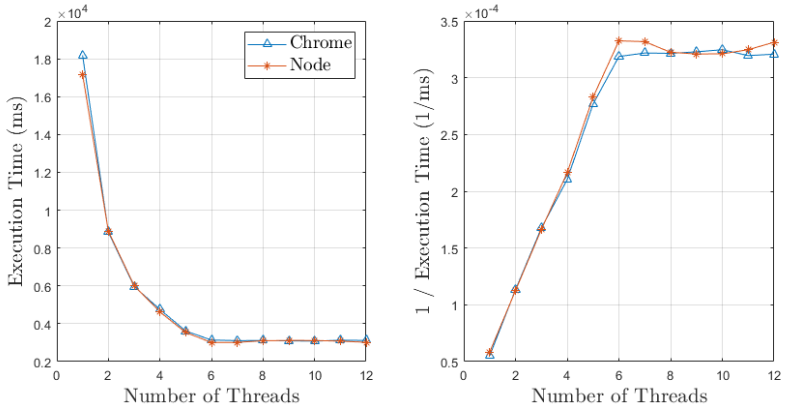


Figure 7.3: Benchmark results of execution time against the number of threads

The listing 7.3 plots the execution speed of *raytracer.fut* with different numbers of threads. The right figure plots the number of threads against the reciprocal of the execution speed. The linear growth up to six threads shows perfect parallelization. However from 6 to 12 threads the plot is flat. This indicates that no more parallelization is exploited after 6 threads. Interestingly 6 is the number of physical CPU cores on the laptop used for benchmarking. Evidently hyper threading didn't improve execution speed. The same behaviour was observed in [16]. It's important to note that both Node.js and Chrome experienced the same affect. This is not surprising as both are implemented on top of the same V8 engine from Google.

7.4 Summary

In this chapter we implemented the Futhark multicore WebAssembly backend. We only needed to make small tweaks to the existing API, to allow us to set the number of threads for the context. We then use the backend to render ray tracing, experiencing a 6 times speed up over the sequential WebAssembly backend. Finally we benchmark our backend and observe that the parallelization is bounded by the number of physical CPU cores and not logical cores.

CHAPTER 8

Conclusion

This thesis has presented the design and implementation of two new backends for Futhark targeting WebAssembly and threaded WebAssembly, enabling efficient execution of sequential and parallel code in the browser. New WebAssembly and threaded WebAssembly backends were developed for the Futhark compiler to compile Futhark functions to WebAssembly modules that run efficiently in the browser. Furthermore, to call the WebAssembly modules from the browser, a simple and efficient JavaScript API was developed and illustrated with examples. The backends and API were implemented in Haskell as additions to Futhark's open source compiler and were benchmarked to prove their efficiency and the performance gains that can be obtained with multithreading on multicore computers. The utility of compiling Futhark to WebAssembly was tested with Mandelbrot and ray tracing, two working visualization programs that generate and display images in the browser.

The JavaScript API hides implementation details from the API user. It does not expose any WebAssembly specific details. It can therefore be used for future Futhark backends targeting different browser technologies such as WebGPU. We also designed the API such that both the sequential and multicore WebAssembly backend were able to have large code reuse. Furthermore the backend only depended on the C API of the Futhark C backend, which is designed to be one of the most stable parts of the Futhark compiler. In that way the code for the WebAssembly backends will be robust to future evolution of the Futhark compiler.

We validate our implementation against the Futhark test suite. It consists of over 1500 test programs. We also use Node.js to run the tests locally. This is because testing WebAssembly in the browser requires a fair deal of extra machinery, and we would not be able to tap into Futhark's existing test framework. We add a Futhark server protocol implementation for JavaScript and functionality for reading and writing Futhark's binary format, thereby allowing us to run the Futhark testing framework.

We benchmarked the Futhark backends to quantify their performance. The

benchmarks come from the futhark benchmark suite, which are standardized benchmarks to test the performance of Futhark backends against industry standards. The benchmarks show the WebAssembly, both when run in the Browser and run on Node.js locally, performs between 13% to 67% slower than the generated C running locally. These results show that the sequential WebAssembly backend can perform at near-native speeds in the browser.

Comparing the threaded WebAssembly backend to the sequential WebAssembly backend, we found up to a 6 times speedup when using a CPU with 6 physical cores on tasks that can be perfectly parallelized. We also learned that we are bound by the number of physical cores, and not logical cores due to the underlying implementation of the V8 engine used by Node.js and Chrome.

In summary Futhark now runs in the browser with efficient single threaded and parallel execution and a simple and efficient JavaScript API. This enables high level and high performance parallel programming in the browser.

APPENDIX A

Source Code

A.1 GenericWASM.hs

```haskell
{-# LANGUAGE QuasiQuotes #-}
{-# LANGUAGE TemplateHaskell #-}

module Futhark.CodeGen.Backends.GenericWASM
  (
    runServer,
    GC.CParts (..),
    GC.asLibrary,
    GC.asExecutable,
    GC.asServer,
    JSEntryPoint (..),
    emccExportNames,
    javascriptWrapper,
    extToString
  )
where

import Data.FileEmbed
import Data.List (intercalate, nub)
import qualified Data.Text as T
import qualified Futhark.CodeGen.Backends.GenericC as GC
import Futhark.CodeGen.Backends.SimpleRep (opaqueName)
import qualified Futhark.CodeGen.ImpCode.Sequential as Imp
import Futhark.IR.Primitive
import NeatInterpolation (text)

extToString :: Imp.ExternalValue -> String
extToString (Imp.TransparentValue (Imp.ArrayValue vn _ pt s dimSize))
  = concat (replicate (length dimSize) "[]") ++ extToString
  (Imp.TransparentValue (Imp.ScalarValue pt s vn))
extToString (Imp.TransparentValue (Imp.ScalarValue (FloatType Float32)
  _ _)) = "f32"
extToString (Imp.TransparentValue (Imp.ScalarValue (FloatType Float64)
  _ _)) = "f64"
extToString (Imp.TransparentValue (Imp.ScalarValue (IntType Int8)
  Imp.TypeDirect _)) = "i8"
extToString (Imp.TransparentValue (Imp.ScalarValue (IntType Int16)
  Imp.TypeDirect _)) = "i16"
extToString (Imp.TransparentValue (Imp.ScalarValue (IntType Int32)
  Imp.TypeDirect _)) = "i32"
extToString (Imp.TransparentValue (Imp.ScalarValue (IntType Int64)
  Imp.TypeDirect _)) = "i64"
```

```
35  extToString (Imp.TransparentValue (Imp.ScalarValue (IntType Int8)
    ↪  Imp.TypeUnsigned _)) = "u8"
36  extToString (Imp.TransparentValue (Imp.ScalarValue (IntType Int16)
    ↪  Imp.TypeUnsigned _)) = "u16"
37  extToString (Imp.TransparentValue (Imp.ScalarValue (IntType Int32)
    ↪  Imp.TypeUnsigned _)) = "u32"
38  extToString (Imp.TransparentValue (Imp.ScalarValue (IntType Int64)
    ↪  Imp.TypeUnsigned _)) = "u64"
39  extToString (Imp.TransparentValue (Imp.ScalarValue Bool _ _)) = "bool"
40  extToString (Imp.OpaqueValue oname vds) = opaqueName oname vds
41  extToString _ = "Not Reached"
42
43  type EntryPointTyp = String
44
45  data JSEntryPoint = JSEntryPoint
46    { name :: String,
47      parameters :: [EntryPointTyp],
48      ret :: [EntryPointTyp]
49    }
50
51  emccExportNames :: [JSEntryPoint] -> [String]
52  emccExportNames jses =
53    map (\jse -> "'_futhark_entry_" ++ name jse ++ "'") jses
54      ++ map (\arg -> "'" ++ gfn "new" arg ++ "'") arrays
55      ++ map (\arg -> "'" ++ gfn "free" arg ++ "'") arrays
56      ++ map (\arg -> "'" ++ gfn "shape" arg ++ "'") arrays
57      ++ map (\arg -> "'" ++ gfn "values_raw" arg ++ "'") arrays
58      ++ map (\arg -> "'" ++ gfn "values" arg ++ "'") arrays
59      ++ map (\arg -> "'" ++ "_futhark_free_" ++ arg ++ "'") opaques
60      ++ ["_futhark_context_config_new", "_futhark_context_config_free",
61          "_futhark_context_new", "_futhark_context_free",
62          "_futhark_context_get_error"]
63    where
64      arrays = filter isArray typs
65      opaques = filter isOpaque typs
66      typs = nub $ concatMap (\jse -> parameters jse ++ ret jse) jses
67      gfn typ str = "_futhark_" ++ typ ++ "_" ++ baseType str ++ "_" ++
        ↪  show (dim str) ++ "d"
68
69  javascriptWrapper :: [JSEntryPoint] -> String
70  javascriptWrapper entryPoints =
71    unlines [
72      jsServer,
73      jsValues,
74      classFutharkOpaque,
75      classFutharkArray,
76      classFutharkContext entryPoints
77    ]
78
79  jsServer :: String
80  jsServer = $(embedStringFile "rts/javascript/server.js")
81
82  jsValues :: String
83  jsValues = $(embedStringFile "rts/javascript/values.js")
84
85  classFutharkOpaque :: String
86  classFutharkOpaque =
87    T.unpack
88      [text|
89    class FutharkOpaque {
90      constructor(ctx, ptr, ffree) { this.ctx = ctx; this.ptr = ptr;
        ↪  this.ffree = ffree; }
91      free() { this.ffree(this.ctx, this.ptr); }
92    }
93    Module['FutharkOpaque'] = FutharkOpaque;
94    |]
95
96  classFutharkArray :: String
```

```
97  classFutharkArray =
98    T.unpack
99      [text|
100   class FutharkArray {
101     constructor(ctx, ptr, type_name, dim, heap, fshape, fvalues,
        ↪  ffree) {
102       this.ctx = ctx;
103       this.ptr = ptr;
104       this.type_name = type_name;
105       this.dim = dim;
106       this.heap = heap;
107       this.fshape = fshape;
108       this.fvalues = fvalues;
109       this.ffree = ffree;
110     }
111     futharkType() { return this.type_name; }
112     free() { this.ffree(this.ctx, this.ptr); }
113     shape() {
114       var s = this.fshape(this.ctx, this.ptr) >> 3;
115       return Array.from(HEAP64.subarray(s, s + this.dim));
116     }
117     toTypedArray(dims = this.shape()) {
118       console.assert(dims.length === this.dim, "dim=%s,dims=%s",
          ↪  this.dim, dims.toString());
119       var length = Number(dims.reduce((a, b) => a * b));
120       var v = this.fvalues(this.ctx, this.ptr) /
          ↪  this.heap.BYTES_PER_ELEMENT;
121       return this.heap.subarray(v, v + length);
122     }
123     toArray() {
124       var dims = this.shape();
125       var ta = this.toTypedArray(dims);
126       return (function nest(offs, ds) {
127         var d0 = Number(ds[0]);
128         if (ds.length === 1) {
129           return Array.from(ta.subarray(offs, offs + d0));
130         } else {
131           var d1 = Number(ds[1]);
132           return Array.from(Array(d0), (x,i) => nest(offs + i * d1,
              ↪  ds.slice(1)));
133         }
134       })(0, dims);
135     }
136   }
137   Module['FutharkArray'] = FutharkArray;
138   |]
139
140 classFutharkContext :: [JSEntryPoint] -> String
141 classFutharkContext entryPoints =
142   unlines [
143     classDef,
144     constructor entryPoints,
145     getFreeFun,
146     getEntryPointsFun,
147     getErrorFun,
148     unlines $ map toFutharkArray arrays,
149     unlines $ map jsWrapEntryPoint entryPoints,
150     endClassDef,
151     "Module['FutharkContext'] = FutharkContext;"
152   ]
153   where
154     arrays = filter isArray typs
155     typs = nub $ concatMap (\jse -> parameters jse ++ ret jse)
        ↪  entryPoints
156
157 classDef :: String
158 classDef = "class FutharkContext {"
```

```
159
160 endClassDef :: String
161 endClassDef = "}"
162
163 constructor :: [JSEntryPoint] -> String
164 constructor jses =
165   T.unpack
166     [text|
167   constructor() {
168     this.cfg = _futhark_context_config_new();
169     this.ctx = _futhark_context_new(this.cfg);
170     this.entry_points = {
171       ${entries}
172     };
173   }
174   |]
175   where
176     entries = T.pack $ intercalate "," $ map dicEntry jses
177
178 getFreeFun :: String
179 getFreeFun =
180   T.unpack
181     [text|
182   free() {
183     _futhark_context_free(this.ctx);
184     _futhark_context_config_free(this.cfg);
185   }
186   |]
187
188 getEntryPointsFun :: String
189 getEntryPointsFun =
190   T.unpack
191     [text|
192   get_entry_points() {
193     return this.entry_points;
194   }
195   |]
196
197 getErrorFun :: String
198 getErrorFun =
199   T.unpack
200     [text|
201   get_error() {
202     var ptr = _futhark_context_get_error(this.ctx);
203     var len = HEAP8.subarray(ptr).indexOf(0);
204     var str = String.fromCharCode(...HEAP8.subarray(ptr, ptr + len));
205     _free(ptr);
206     return str;
207   }
208   |]
209
210 dicEntry :: JSEntryPoint -> String
211 dicEntry jse =
212   T.unpack
213     [text|
214   '${ename}' : [${params}, ${rets}]
215   |]
216   where
217     ename = T.pack $ name jse
218     params = T.pack $ show $ parameters jse
219     rets = T.pack $ show $ ret jse
220
221 jsWrapEntryPoint :: JSEntryPoint -> String
222 jsWrapEntryPoint jse =
223   unlines
224     [ func_name ++ "(" ++ inparams ++ ") {",
225       "  var out = [" ++ outparams ++ "].map(n => _malloc(n));",
226       "  var to_free = [];",
```

```
227         "  var do_free = () => { out.forEach(_free); to_free.forEach(f
            ↪  => f.free()); };",
228         paramsToPtr,
229         "  if (_futhark_entry_" ++ func_name ++ "(this.ctx, ...out, " ++
            ↪  ins ++ ") > 0) {",
230         "    do_free();",
231         "    throw this.get_error();",
232         "  }",
233         results,
234         "  do_free();",
235         "  return " ++ res ++ ";",
236         "}"
237       ]
238     where
239       func_name = name jse
240
241       alp = [0 .. length (parameters jse) - 1]
242       inparams = intercalate ", " ["in" ++ show i | i <- alp]
243       ins = intercalate ", " [maybeDerefence ("in" ++ show i) $
            ↪  parameters jse !! i | i <- alp]
244       paramsToPtr = unlines $ filter ("" /=) [arrayPointer ("in" ++ show
            ↪  i) $ parameters jse !! i | i <- alp]
245
246       alr = [0 .. length (ret jse) - 1]
247       outparams = intercalate ", " [show $ typeSize $ ret jse !! i | i
            ↪  <- alr]
248       results = unlines [makeResult i $ ret jse !! i | i <- alr]
249       res_array = intercalate ", " ["result" ++ show i | i <- alr]
250       res = if length (ret jse) == 1 then "result0" else ("[" ++
            ↪  res_array ++ "]")
251
252   maybeDerefence :: String -> String -> String
253   maybeDerefence arg typ =
254     if isScalar typ then arg else (arg ++ ".ptr")
255
256   arrayPointer :: String -> String -> String
257   arrayPointer arg typ =
258     if isArray typ
259     then "  if (" ++ arg ++ " instanceof Array) { " ++ reassign ++ ";
            ↪  to_free.push(" ++ arg ++ "); }"
260     else ""
261     where
262       reassign = arg ++ " = this.new_" ++ signature ++ "_from_jsarray("
            ↪  ++ arg ++ ")"
263       signature = baseType typ  ++ "_" ++ show (dim typ) ++ "d"
264
265   makeResult :: Int -> String -> String
266   makeResult i typ =
267     "  var result" ++ show i ++ " = " ++
268       if isArray typ
269       then "this.new_" ++ signature ++ "_from_ptr(" ++ readout ++ ");"
270       else
271         if isOpaque typ
272         then "new FutharkOpaque(this.ctx, " ++ readout ++ ",
            ↪  _futhark_free_" ++ typ ++ ");"
273         else readout ++ if typ == "bool" then "!==0;" else ";"
274     where
275       res = "out[" ++ show i ++ "]"
276       readout = typeHeap typ ++ "[" ++ res ++ " >> " ++ show (typeShift
            ↪  typ) ++ "]"
277       signature = baseType typ  ++ "_" ++ show (dim typ) ++ "d"
278
279   baseType :: String -> String
280   baseType ('[' : ']' : end) = baseType end
281   baseType typ = typ
282
283   dim :: String -> Int
284   dim ('[' : ']' : end) = dim end + 1
```

```
285 dim _ = 0
286
287 isArray :: String -> Bool
288 isArray typ = take 2 typ == "[]"
289
290 isOpaque :: String -> Bool
291 isOpaque typ = take 6 typ == "opaque"
292
293 isScalar :: String -> Bool
294 isScalar typ = not (isArray typ || isOpaque typ)
295
296 typeSize :: String -> Integer
297 typeSize typ =
298   case typ of
299     "i8"   -> 1
300     "i16"  -> 2
301     "i32"  -> 4
302     "i64"  -> 8
303     "u8"   -> 1
304     "u16"  -> 2
305     "u32"  -> 4
306     "u64"  -> 8
307     "f32"  -> 4
308     "f64"  -> 8
309     "bool" -> 1
310     _      -> 4
311
312 typeShift :: String -> Integer
313 typeShift typ =
314   case typ of
315     "i8"   -> 0
316     "i16"  -> 1
317     "i32"  -> 2
318     "i64"  -> 3
319     "u8"   -> 0
320     "u16"  -> 1
321     "u32"  -> 2
322     "u64"  -> 3
323     "f32"  -> 2
324     "f64"  -> 3
325     "bool" -> 0
326     _      -> 2
327
328 typeHeap :: String -> String
329 typeHeap typ =
330   case typ of
331     "i8"   -> "HEAP8"
332     "i16"  -> "HEAP16"
333     "i32"  -> "HEAP32"
334     "i64"  -> "HEAP64"
335     "u8"   -> "HEAPU8"
336     "u16"  -> "HEAPU16"
337     "u32"  -> "HEAPU32"
338     "u64"  -> "HEAPU64"
339     "f32"  -> "HEAPF32"
340     "f64"  -> "HEAPF64"
341     "bool" -> "HEAP8"
342     _      -> "HEAP32"
343
344 toFutharkArray :: String -> String
345 toFutharkArray typ =
346   unlines [
347     new ++ "_from_jsarray(" ++ arraynd ++ ") {",
348     "  return this." ++ new ++ "(" ++ arraynd_flat ++ ", " ++
      ↪  arraynd_dims ++ ");",
349     "}",
350     new ++ "(array, " ++ dims ++ ") {",
351     "  console.assert(array.length === " ++ dims_multiplied ++ ",
      ↪    'len=%s,dims=%s', array.length, [" ++ dims ++
      ↪    "].toString());",
352     "  var copy = _malloc(array.length << " ++ show (typeShift ftype)
      ↪  ++ ");",
```

```
353       "  " ++ typeHeap ftype ++ ".set(array, copy >> " ++ show
          ↪  (typeShift ftype) ++ ");",
354       "  var ptr = " ++ fnew ++ "(this.ctx, copy, " ++ bigint_dims ++
          ↪  ");",
355       "  _free(copy);",
356       "  return this." ++ new ++ "_from_ptr(ptr);",
357       "}",
358       new ++ "_from_ptr(ptr) {",
359       "  return new FutharkArray(this.ctx, ptr, "
360         ++ intercalate ", " ["'" ++ ftype ++ "'", show d, heap, fshape,
            ↪  fvalues, ffree] ++ ");",
361       "}"
362     ]
363     where
364       d = dim typ
365       ftype = baseType typ
366       heap = typeHeap ftype
367       signature = ftype ++ "_" ++ show d ++ "d"
368       new = "new_" ++ signature
369       fnew = "_futhark_new_" ++ signature
370       fshape = "_futhark_shape_" ++ signature
371       fvalues = "_futhark_values_raw_" ++ signature
372       ffree = "_futhark_free_" ++ signature
373       arraynd = "array" ++ show d ++ "d"
374       arraynd_flat = if d > 1 then arraynd ++ ".flat()" else arraynd
375       arraynd_dims = intercalate ", " [ arraynd ++ mult i "[0]" ++
          ↪  ".length" | i <- [0..d-1] ]
376       dims = intercalate ", " [ "d" ++ show i | i <- [0..d-1] ]
377       dims_multiplied = intercalate "*" [ "d" ++ show i | i <- [0..d-1]
          ↪  ]
378       bigint_dims = intercalate ", " [ "BigInt(d" ++ show i ++ ")" | i
          ↪  <- [0..d-1] ]
379       mult i s = concat $ replicate i s
380
381 runServer :: String
382 runServer =
383   T.unpack
384     [text|
385    Module.onRuntimeInitialized = () => {
386      var context = new FutharkContext();
387      var server = new Server(context);
388      server.run();
389    }
390   |]
```

Listing 31: Haskell source code for JavaScript wrapper code generation

A.2 mandelbrot.fut

```
1  let dot (r: f32, i: f32): f32 =
2    r * r + i * i
3
4  let multComplex (a: f32, b: f32) (c: f32, d: f32): (f32,f32) =
5    (a*c - b * d,
6     a*d + b * c)
7
8  let addComplex (a: f32, b: f32) (c: f32, d: f32): (f32,f32) =
9    (a + c,
10    b + d)
11
12 let divergence (depth: i32) (c0: (f32,f32)): i32 =
13   let (_, i) = loop (c, i) = (c0, 0) while (i < depth) && (dot c <
     ↪  4.0) do
```

```
    (addComplex c0 (multComplex c c),
     i + 1)
  in i

let mandelbrot (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 ->
            map (\x ->
                   let c0 = (xmin + (f32.i64 x * sizex) / f32.i64
                   ↪  screenX,
                             ymin + (f32.i64 y * sizey) / f32.i64
                             ↪  screenY)
                   in (divergence depth c0))
                (iota screenX))
            (iota screenY)

let escapeToColour (depth: i32) (divergence: i32): i32 =
  if depth == divergence
  then 0
  else
    let r = 3 * divergence
    let g = 5 * divergence
    let b = 7 * divergence
    in (r<<16 | g<<8 | b)

let main (screenX: i64) (screenY: i64)
         (depth: i32) (xmin: f32)
         (ymin: f32) (xmax: f32)
         (ymax: f32): [screenY][screenX]i32 =
  let escapes = mandelbrot screenX screenY depth xmin ymin xmax ymax
  in map (\row ->
            map (escapeToColour depth) row)
         escapes
```

Listing 32: mandelbrot.fut source code

A.3 raytracer.fut

```
import "lib/github.com/athas/vector/vspace"

module vec3 = mk_vspace_3d f32
type vec3 = vec3.vector

-- A convenient alias so we don't have to indicate the fields all the
-- time.
let vec (x, y, z) : vec3 = {x,y,z}

type ray = {origin: vec3, direction: vec3}

let point_at_parameter (r: ray) (t: f32) =
  vec3.(r.origin + scale t r.direction)

let reflect (v: vec3) (n: vec3) : vec3 =
  v vec3.- (2 * vec3.dot v n `vec3.scale` n)

type refraction = #no_refract | #refract vec3

let refract (v: vec3) (n: vec3) (ni_over_nt: f32) : refraction =
  let uv = vec3.normalise v
  let dt = vec3.dot uv n
  let discriminant = 1 - ni_over_nt*ni_over_nt*(1-dt*dt)
  in if discriminant > 0
     then #refract ((ni_over_nt `vec3.scale` (uv vec3.- (dt
     ↪  `vec3.scale` n)))
```

```
                    vec3.- (f32.sqrt discriminant `vec3.scale` n))
    else #no_refract

let schlick (cosine: f32) (ref_idx: f32) =
  let r0 = (1-ref_idx) / (1+ref_idx)
  let r0 = r0*r0
  in r0 + (1-r0)*(1-cosine)**5

import "lib/github.com/diku-dk/cpprandom/random"

module rnge = pcg32
module dist = uniform_real_distribution f32 rnge
type rng = rnge.rng

let rand : rng -> (rng, f32) = dist.rand (0,1)

let random_in_unit_sphere rng =
  let new rng = let (rng, x) = dist.rand (-1, 1) rng
                let (rng, y) = dist.rand (-1, 1) rng
                let (rng, z) = dist.rand (-1, 1) rng
                in (rng, vec(x,y,z))
  let outside_sphere = vec3.quadrance >-> (>=1)
  in iterate_while ((.1) >-> outside_sphere) ((.0) >-> new) (new rng)

type camera = { origin: vec3
              , lower_left_corner: vec3
              , horizontal: vec3
              , vertical: vec3
              , u: vec3, v: vec3, w: vec3
              , lens_radius: f32}

let camera (lookfrom: vec3) (lookat: vec3) (vup: vec3) (vfov: f32)
↪  (aspect: f32)
           (aperture: f32) (focus_dist: f32) : camera =
  let theta = vfov * f32.pi / 180
  let half_height = f32.tan (theta / 2)
  let half_width = aspect * half_height
  let origin = lookfrom
  let w = vec3.normalise (lookfrom vec3.- lookat)
  let u = vec3.normalise (vec3.cross vup w)
  let v = vec3.cross w u
  in { lower_left_corner = origin vec3.-
                           (half_width * focus_dist `vec3.scale` u)
                           ↪  vec3.-
                           (half_height * focus_dist `vec3.scale` v)
                           ↪  vec3.-
                           (focus_dist `vec3.scale` w)
     , horizontal = (2*half_width*focus_dist) `vec3.scale` u
     , vertical = (2*half_height*focus_dist) `vec3.scale` v
     , origin, u, v, w
     , lens_radius = aperture / 2}

let get_ray (c: camera) (s: f32) (t: f32) (rng: rng) : (rng, ray) =
  let {origin, lower_left_corner, horizontal, vertical, u, v, w=_,
  ↪  lens_radius} = c
  let (rng, p) = random_in_unit_sphere rng
  let rd = lens_radius `vec3.scale` p
  let offset = vec3.((rd.x `scale` u) + (rd.y `scale` v))
  in (rng,
      { origin = offset vec3.+ c.origin
      , direction = vec3.(lower_left_corner +
                          (s `scale` horizontal) +
                          (t `scale` vertical) -
                          origin -
                         offset)})

type material = #lambertian {albedo: vec3}
              | #metal {albedo: vec3, fuzz: f32}
              | #dielectric {ref_idx: f32}
```

```
91
92  type hit_info = {t: f32, p: vec3, normal: vec3, material: material}
93
94  type hit = #no_hit | #hit hit_info
95
96  type sphere = {center: vec3, radius: f32, material: material}
97
98  let sphere_hit {center, radius, material} (r: ray) (t_min: f32)
    ↪  (t_max: f32) : hit =
99    let oc = vec3.(r.origin - center)
100   let a = vec3.dot r.direction r.direction
101   let b = vec3.dot oc r.direction
102   let c = vec3.dot oc oc - radius*radius
103   let discriminant = b*b - a*c
104   let try_hit (temp: f32) =
105     if temp < t_max && temp > t_min
106     then (#hit { t = temp
107                , p = point_at_parameter r temp
108                , normal = (1/radius) `vec3.scale` (point_at_parameter
                   ↪  r temp vec3.- center)
109                , material
110                })
111     else #no_hit
112   in if discriminant <= 0
113      then #no_hit
114      else match try_hit ((-b - f32.sqrt(b*b-a*c))/a)
115           case #hit h -> #hit h
116           case #no_hit -> try_hit ((-b + f32.sqrt(b*b-a*c))/a)
117
118 type obj = #sphere sphere
119
120 let hit [n] (objs: [n]obj) (r: ray) (t_min: f32) (t_max: f32) : hit =
121   (loop (hit, closest_so_far) = (#no_hit, t_max) for obj in objs do
122    let hit' = match obj
123               case #sphere s -> sphere_hit s r t_min closest_so_far
124    in match hit'
125       case #no_hit -> (hit, closest_so_far)
126       case #hit h -> (#hit h, h.t)).0
127
128 type scatter = #scatter {attenuation: vec3, scattered: ray}
129              | #no_scatter
130
131 let scattering (r: ray) (h: hit_info) (rng: rng) : (rng, scatter) =
132   match h.material
133
134   case #lambertian {albedo} ->
135     let (rng, bounce) = random_in_unit_sphere rng
136     let target = vec3.(h.p + h.normal + bounce)
137     in (rng, #scatter {attenuation=albedo,
138                        scattered={origin = h.p, direction = target
                           ↪  vec3.- h.p}})
139
140   case #metal {albedo, fuzz} ->
141     let reflected = reflect (vec3.normalise r.direction) h.normal
142     let (rng, bounce) = random_in_unit_sphere rng
143     let scattered = {origin = h.p, direction = reflected vec3.+ (fuzz
        ↪  `vec3.scale` bounce)}
144     in if vec3.dot scattered.direction h.normal > 0
145        then (rng, #scatter {attenuation=albedo,
146                             scattered})
147        else (rng, #no_scatter)
148
149   case #dielectric {ref_idx} ->
150     let reflected = reflect r.direction h.normal
151     let attenuation = vec(1, 1, 1)
152     let (outward_normal, ni_over_nt, cosine) =
153       if vec3.dot r.direction h.normal > 0
154       then (vec3.map f32.neg h.normal,
155             ref_idx,
```

```
156             ref_idx * vec3.dot r.direction h.normal / vec3.norm
                ↪  r.direction)
157       else (h.normal,
158             1/ref_idx,
159             -vec3.dot r.direction h.normal / vec3.norm r.direction)
160    in match refract r.direction outward_normal ni_over_nt
161       case #refract refracted ->
162         let reflect_prob = schlick cosine ref_idx
163         let (rng, x) = rand rng
164         let direction = if x < reflect_prob then reflected else
            ↪  refracted
165         in (rng, #scatter {attenuation, scattered={origin=h.p,
            ↪  direction}})
166       case #no_refract ->
167         (rng, #scatter {attenuation, scattered={origin=h.p,
            ↪  direction=reflected}})
168
169 let color (max_depth: i32) (objs: []obj) (r: ray) (rng: rng) : (rng,
    ↪  vec3) =
170   let ((rng, _), (_, color)) =
171     loop ((rng, r), (depth, color)) = ((rng, r), (0, vec(1,1,1)))
        ↪  while depth < max_depth
172     do match hit objs r 0.00001 f32.highest
173        case #hit h ->
174          (match scattering r h rng
175           case (rng, #scatter {attenuation, scattered}) ->
176             ((rng, scattered), (depth+1, attenuation vec3.* color))
177           case (rng, #no_scatter) ->
178             ((rng, r), (max_depth, vec(0,0,0))))
179        case #no_hit ->
180          let unit_direction = vec3.normalise r.direction
181          let t = 0.5 * (unit_direction.y + 1)
182          let color' = color vec3.*
183                       (((1-t) `vec3.scale` vec(1, 1, 1)) vec3.+
184                        (t `vec3.scale` vec(0.5, 0.7, 1.0)))
185          in ((rng, r), (max_depth, color'))
186   in (rng, color)
187
188 let random_object_at (a: f32) (b: f32) (rng: rng) : (rng, obj) =
189   let (rng, center) = let (rng, xd) = rand rng
190                       let (rng, yd) = rand rng
191                       in (rng, vec(a+0.9*xd, 0.2, b+0.9*yd))
192   let randp rng = let (rng, x) = rand rng
193                   let (rng, y) = rand rng
194                   in (rng, x * y)
195   let (rng, choose_mat) = rand rng
196   let (rng, material) =
197     if choose_mat > 0.95 then
198       (rng, #dielectric {ref_idx=1.5})
199     else
200     let (rng, x) = randp rng
201     let (rng, y) = randp rng
202     let (rng, z) = randp rng
203     let albedo = vec(x,y,z)
204     let (rng, fuzz) = rand rng
205     in if choose_mat > 0.8
206        then (rng, #metal {albedo, fuzz})
207        else (rng, #lambertian {albedo})
208   in (rng,
209       #sphere {center, radius=0.2, material})
210
211 -- From http://stackoverflow.com/a/12996028
212 let hash (x: i32): i32 =
213   let x = ((x >> 16) ^ x) * 0x45d9f3b
214   let x = ((x >> 16) ^ x) * 0x45d9f3b
215   let x = ((x >> 16) ^ x) in
216   x
```

```
217
218 import "lib/github.com/athas/matte/colour"
219
220 let random_world (seed: i32) (n: i32) =
221   let mk_obj a b = let rng = rnge.rng_from_seed [seed, a ^ b]
222                    in random_object_at (r32 a) (r32 b) rng
223   let span = -n..<n
224   let (rngs, objs) = map (\a -> map (mk_obj a) span) span
225                      |> map unzip |> unzip
226   let rng = rnge.join_rng (flatten rngs)
227
228   let fixed_objs = [ #sphere {center=vec(0,-1000,0), radius=1000,
    ↪  material=#lambertian {albedo=vec(0.5,0.5,0.5)}}
229                    , #sphere {center=vec(0,1,0), radius=1,
                       ↪  material=#dielectric {ref_idx=1.5}}
230                    , #sphere {center=vec(-4,1,0), radius=1,
                       ↪  material=#lambertian {albedo=vec(0.4,0.2,0.1)}}
231                    , #sphere {center=vec(4,1,0), radius=1,
                       ↪  material=#metal {albedo=vec(0.6,0.6,0.5),
                       ↪  fuzz=0}}
232                    ]
233
234   let world = flatten objs ++ fixed_objs
235
236   in (rng, world)
237
238 let render (max_depth: i32) (nx: i64) (ny: i64) (nss: [ny][nx]i32)
    ↪  (world: []obj) (cam: camera) (rngs: [ny][nx]rng) =
239   let sample j i (rng, acc) = let (rng, ud) = rand rng
240                               let (rng, vd) = rand rng
241                               let u = (f32.i64(i) + ud) / f32.i64(nx)
242                               let v = (f32.i64(j) + vd) / f32.i64(ny)
243                               let (rng, r) = get_ray cam u v rng
244                               let (rng, col) = color max_depth world r
                                  ↪  rng
245                               in (rng, acc vec3.+ col)
246   let pixel j i = let rng = rngs[j,i]
247                   let ns = (reverse nss)[j,i]
248                   let (rng, col) = iterate ns (sample j i) (rng,
                    ↪  vec(0,0,0))
249                   let col = ((1/r32 ns) `vec3.scale` col) |> vec3.map
                    ↪  f32.sqrt
250                   in (rng, argb.from_rgba col.x col.y col.z 0)
251   in tabulate_2d ny nx pixel |> reverse
252
253
254 let main (nx: i64) (ny: i64) (ns: i32) (nobj: i32):
    ↪  [ny][nx]argb.colour =
255   let lookfrom = vec(13,2,3)
256   let lookat = vec(0,0,0)
257   let dist_to_focus = 10
258   let aperture = 0.1
259   let cam = camera lookfrom lookat (vec(0,1,0)) 20 (f32.i64 nx /
    ↪  f32.i64 ny)
260                   aperture dist_to_focus
261   let (rng, world) = random_world (i32.i64 nx ^ i32.i64 ny ^ ns) nobj
262   let rngs = rnge.split_rng (nx*ny) rng |> unflatten ny nx
263   let nss = replicate ny (replicate nx ns)
264   in render 50 nx ny nss world cam rngs |> map (map (.1))
```

Listing 33: raytracer.fut source code, originally taken from `https://github.com/athas/raytracinginoneweekendinfuthark`

Bibliography

[1] Guy E Blelloch. *Vector models for data-parallel computing*. Vol. 2. MIT press Cambridge, 1990.

[2] Nikhil Thorat Daniel Smilkov and Ann Yuan. *Introducing the WebAssembly backend for TensorFlow.js*. 2020. URL: https://blog.tensorflow.org/2020/03/introducing-webassembly-backend-for-tensorflow-js.html.

[3] Gerard Gallant. *WebAssembly in action : with examples using C++ and Emscripten*. eng. Shelter Island, NY, 2019.

[4] Andreas Haas et al. "Bringing the web up to speed with WebAssembly". In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*. Ed. by Albert Cohen and Martin T. Vechev. ACM, 2017, pp. 185–200. DOI: 10.1145/3062341.3062363. URL: https://doi.org/10.1145/3062341.3062363.

[5] Troels Henriksen. "Design and Implementation of the Futhark Programming Language". English. PhD thesis. 2017.

[6] Troels Henriksen et al. "Futhark: Purely Functional GPU-Programming with Nested Parallelism and in-Place Array Updates". In: *SIGPLAN Not.* 52.6 (June 2017), pp. 556–571. ISSN: 0362-1340. DOI: 10.1145/3140587.3062354. URL: https://doi.org/10.1145/3140587.3062354.

[7] ECMA International. "ECMAScript 2022 Language Specification". In: (2021). URL: https://tc39.es/ecma262.

[8] Chris Lattner and Vikram S. Adve. "The LLVM Compiler Framework and Infrastructure Tutorial". In: *Languages and Compilers for High Performance Computing, 17th International Workshop, LCPC 2004, West Lafayette, IN, USA, September 22-24, 2004, Revised Selected Papers*. Ed. by Rudolf Eigenmann, Zhiyuan Li, and Samuel P. Midkiff. Vol. 3602. Lecture Notes in Computer Science. Springer, 2004, pp. 15–16. DOI: 10.1007/11532378_2. URL: https://doi.org/10.1007/11532378%5C_2.

[9] Jianming Liang, Jianhua Gong, and Wenhang Li. "Applications and impacts of Google Earth: A decadal review (2006–2016)". In: *ISPRS Journal of Photogrammetry and Remote Sensing* 146 (Sept. 2018), pp. 91–107. DOI: 10.1016/j.isprsjprs.2018.08.019.

[10] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[11] Mozilla. *JavaScript Reference*. 2021. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Atomics.

[12] Thoomas Nattestad. *WebAssembly brings Google Earth to more browsers*. 2019. URL: https://blog.chromium.org/2019/06/webassembly-brings-google-earth-to-more.html.

[13] *pthread(7) - Linux Manual Page*. 2020. URL: https://man7.org/linux/man-pages/man7/pthreads.7.html.

[14] Daniel Smilkov et al. "TensorFlow.js: Machine Learning for the Web and Beyond". In: *CoRR* abs/1901.05350 (2019). arXiv: 1901.05350. URL: http://arxiv.org/abs/1901.05350.

[15] Duc Minh Tran. *Multicore backend for Futhark*. 2020.

[16] Javier Verdú and Alex Pajuelo. "Performance Scalability Analysis of JavaScript Applications with Web Workers". In: *IEEE Computer Architecture Letters* 15.2 (2016), pp. 105–108. DOI: 10.1109/LCA.2015.2494585.

[17] *Web Workers in HTML Living Standard*. URL: https://html.spec.whatwg.org/multipage/workers.html#workers.

[18] Bennet Yee et al. "Native Client: A Sandbox for Portable, Untrusted x86 Native Code". In: *IEEE Symposium on Security and Privacy (Oakland'09)*. IEEE, 3 Park Avenue, 17th Floor, New York, NY 10016, 2009. URL: http://nativeclient.googlecode.com/svn/data/docs_tarball/nacl/googleclient/native_client/documentation/nacl_paper.pdf.

[19] Alon Zakai. "Emscripten: an LLVM-to-JavaScript compiler". In: *Companion to the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*. Ed. by Cristina Videira Lopes and Kathleen Fisher. ACM, 2011, pp. 301–312. DOI: 10.1145/2048147.2048224. URL: https://doi.org/10.1145/2048147.2048224.