Parallel Execution in Browser

ECSE 420 – Parallel Computing

Group 14

Anssam Ghezala – 260720743

Elsa Emilien – 260735998

Garrett Kinman – 260763260

Ahmed Azhar – 260733580

Introduction

The Problem

For the longest time, most web browsers only supported a select few languages: HTML, CSS and JavaScript. Among these, JavaScript would be the only programming language that would run in the browser. JavaScript imposes certain limitations in regards to program development on the browser, which thus imposes limitations as to what can be made to run in-browser. JavaScript limits in-browser programs to run as a single-thread. Therein lies the problem of having a long sequential program that would have a significant run-time, which could have been massively reduced through parallelization.

Only recently, a new language, WebAssembly, has been added to browsers, which offers the ability to compile languages such as C++, Rust, Go, and many others into a format that can execute in-browser. This allows multithreading, parallelism, and access to lower-level hardware functionality that was otherwise not possible with JavaScript. The problem we wish to address is to use WebAssembly to run parallel code in the browser. Our aim of this project is to implement parallelization by using WebAssembly to run parallel code in the browser and optimize its runtime.

The Project

Background

To give some perspective to the project, we will first discuss the different tools that we used and explain their usage for this project. We will explain some terms and concepts that are useful to better understand the overall implementation.

Pthreads: Posix threads are defined as a set of C language programming types and procedure calls[1]. There are many existing code libraries written in C or C++ that use pthreads, and those can be compiled to WebAssembly (Wasm) and run in true threaded mode, allowing more cores to work on the same data simultaneously. Pthreads are well-suited to the purpose of our project since they are light-weight, have a shared-memory model and a well defined API to allow for synchronization.

WebWorkers: WebWorkers are objects that offload some of the processing tasks that run on the browser. Since JavaScript is a single thread programming language, WebWorkers were introduced as a way to provide means for concurrent processing. Hence, WebWorkers enable developers to benefit from parallel programming in JavaScript, thus allowing applications to run different computations at the same time[2]. WebWorkers allow the user to create background threads that are separate from the main execution thread, where the user interface logic is run. The core advantage of this workload separation is that you can run expensive operations within an isolated thread and without interrupting or affecting the responsiveness and usability of the main thread[2].

The Browser: For the purpose of our project, we decided to work with Chrome 70 since it supports WebAssembly threads. In order to check if a browser supports the use of WebAssembly threads, one can go to *chrome://flags* and make sure that the *WebAssembly thread support* field is available, and if it is, it should be updated to *Enabled*. Browsers have supported parallelism via WebWorkers since 2012 in Chrome 4[2]. The structure that we are familiar with is one main thread and multiple background threads that run through WebWorkers. However, WebWorkers do not share mutable data between them, instead they rely on message-passing for communication, thus they cannot share mutable data like pthreads.[3]

WebAssembly threads, however, can share the same Wasm memory. A SharedArrayBuffer, which allows one single ArrayBuffer to be shared concurrently between WebWorkers, is used as a shared memory. Each WebAssembly thread runs in a WebWorker, but their shared Wasm memory allows them to work much like they do on native platforms[3]. Since there is concurrent access to the same data simultaneously, it is important that applications that use Wasm threads take care of handling the data race and potential deadlock situations that may arise from this.

Chrome OS uses the Completely Fair Scheduler. This scheduler is designed to balance or maintain fairness, by splitting up processor time to tasks using a virtual runtime to keep track of time[4].

Emscripten: Emscripten is an open source compiler toolchain to WebAssembly. Emscripten generates two files, one the binary WebAssembly file and one JavaScript file.The browser engine then converts both modules into machine code. Emscripten can be git cloned and then activated using the emsdk commands:

```
./emsdk install latest
./emsdk activate latest
```

In addition to this, Emscripten also comes with a helper bash script to set up the environment and environment variables.

WebAssembly: With the previous terms defined, we can have a better understanding of WebAssembly (Wasm). Wasm is supported by all the major browsers, allows for faster execution time, and is an open standard from the World Wide Web Consortium (W3C). Another WebAssembly advantage stems from the fact that code runs within a virtual machine. As a result, each WebAssembly module executes within a sandboxed environment, separated from the host runtime using fault isolation techniques[5]. Wasm allows users to write code for the web using compiled languages such as C/C++, Rust, Kotlin, and others. Hence C code can be parallelized using pthreads and then compiled using Emscripten to be run through WebWorkers on the browser.

Workflow

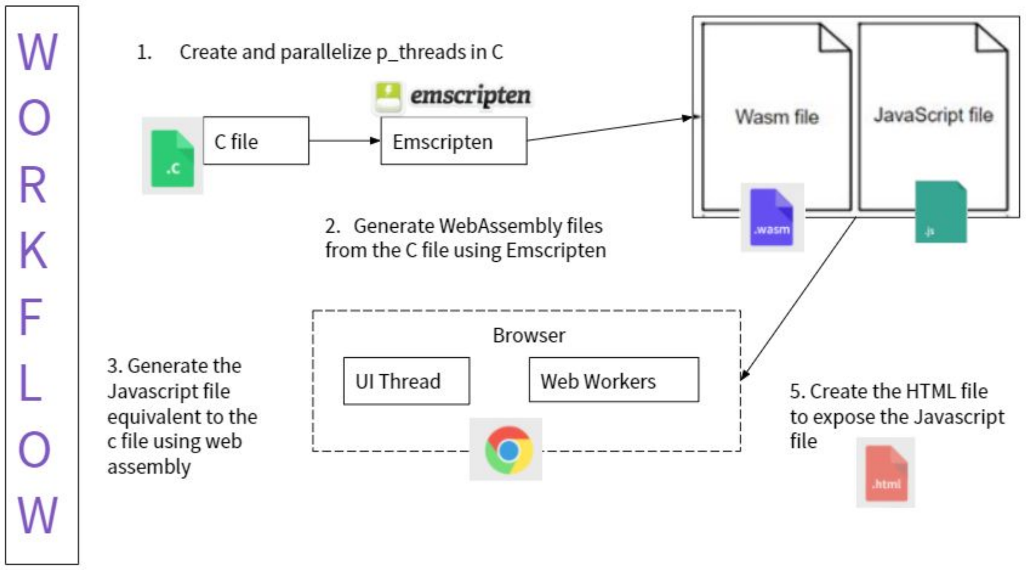

W O R K F L O W
1. Create and parallelize p_threads in C
emscripten
C file
.c
Emscripten
Wasm file
JavaScript file
.wasm
.js
2. Generate WebAssembly files from the C file using Emscripten
Browser
UI Thread
Web Workers
3. Generate the Javascript file equivalent to the c file using web assembly
5. Create the HTML file to expose the Javascript file
.html

Fig. 1 Workflow from a C file into the browser using WebAssembly

We start by implementing the C code. As a proof of concept, we used the max pooling algorithm that we implemented in the Lab 0. For pooling, our parallelization scheme was to give *x* number of windows to each thread. Here we defined a window as the 4 (2x2) pixels. We also defined *x* (number of windows per thread) as the size of the image (width*height) divided by (4 * number of threads). Once we determine the number of windows per thread, we know how many pooling operations will be carried out by each thread. For each window, each thread will do the pooling operation on the RGBA channels that constitute each pixel.

As the next step, we install Emscripten to compile the C code. To install Emscripten we can clone the compiler at https://github.com/emscripten-core/emsdk. We then use the emsdk command to the latest stable build of Emscripten, and to activate it. Then, using the source command to set up the path to find the emcc, the Emscripten compiler.

```
git clone https://github.com/emscripten-core/emsdk.git
cd emsdk
./emsdk install latest
./emsdk activate latest
source ./emsdk_env.sh
```

Our code in this github repository which can cloned:
https://github.com/AnssamGhezala/WASM_Source
It is also submitted as a ZIP file to the deliverable. **Please look at the README.md for more information on how to compile the code.**

In summary, to compile our pooling parallel code we use the following command:

```
emcc lodepng.c -o test.js functions.c -s USE_PTHREADS=1 -s
INITIAL_MEMORY=672137216 -s --embed-file Test_3.png -s PTHREAD_POOL_SIZE
=3
```

Where *emcc* is the gcc-equivalent compiler command for Emscripten. We want to compile both the *lodepng.c* and the *functions.c* files where *lodepng.c* is the same file we used to flatten the 2D colored image into a 1D array, and *functions.c* is where we have the parallelized pooling algorithm. *Test.js* is the JavaScript file that is created as the output file that calls the generated *.wasm* binary file. Every time we want to add more specifications to use -s. We specified an initial memory of 672137216, since, without this field, an error was thrown, specifying the set the initial memory to the value mentioned above. We also embedded the input file *Test_3.png* since the Wasm environment cannot retrieve files from outside the wasm sandbox. In addition to this, the *USE_PTHREADS* flag was specified and takes in either 1 if we used pthreads in our implementation and 0 if we did not. If we did use pthreads, we also specify the *PTHREAD_POOL_SIZE* which is the number of pthreads that we want our program to execute with.

Once the *test.js* file was obtained, we implemented a simple HTML file to call it and hence expose the parallelized max pooling program.

Critical path

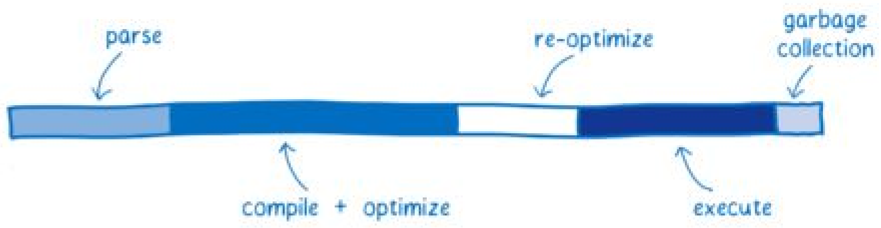


Fig. 2 critical path of JavaScript source code when opening it to run on the browser

Clark explains that each bar roughly represents the time allocated for a task performed by the engine[12]:

- Parsing: the time to convert the source code into an interpretable version.
- Compiling + optimizing: the time that is spent in the baseline compiler and optimizing compiler.
- Re-optimizing: the time the Just In Time[10] compiler spends re-optimizing code and bailing out of optimized code back to the baseline code.
- Execution: time taken to run the code.
- Garbage collection: time to clean up memory.

Here's an approximation to how Web Assembly would compare, the first bar being the JavaScript engine bar and the one below for Web Assembly:

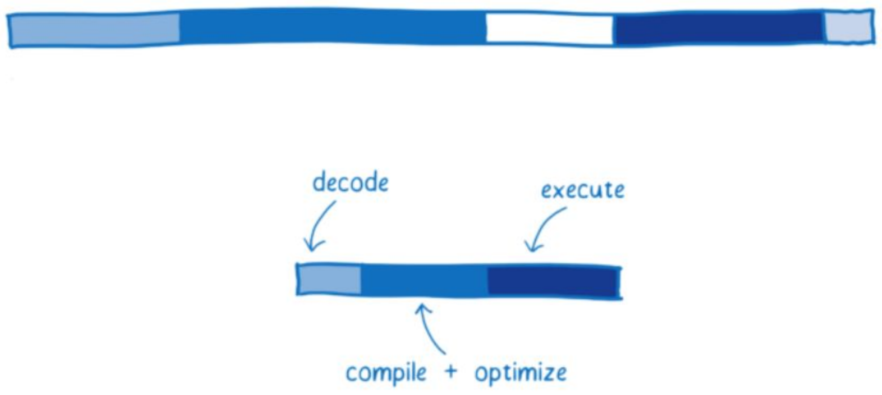


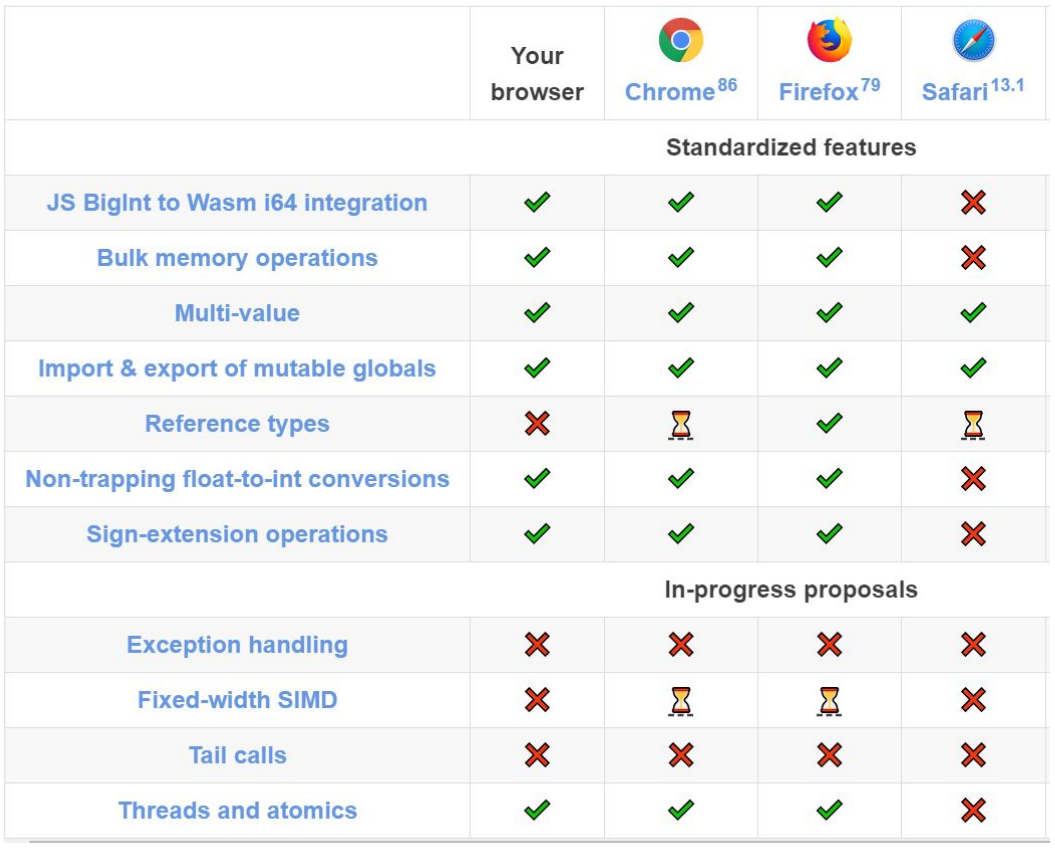
Fig. 3 critical path of Web Assembly code interpretation by the browser

We notice a lower average time for the compiling and optimization phase. This is due to the fact that Web Assembly is code that is much closer to machine code. Web Assembly's critical path does not include re-optimization phase. JIT needs a re-optimization phase due to the assumptions that it makes when running JavaScript code. JIT fixes these assumptions during re-optimization when they turn out to be wrong, for example, a wrong assumption on the type of a variable. The execution time for Web Assembly interpreted code is much shorter as well. This is due to the fact that although it is possible to write performant code to run on the JIT, it is on the developer's responsibility to know how to write code such that the compiler won't take too much time re-optimising[11]. Because of this, executing code in WebAssembly is generally faster, since the compiler generates the code (instead of a human) so it is more ideal for the machine. Last but not least, WebAssembly's critical path does not include garbage collection as it is managed manually through our C code.

Challenges

Over the course of this project, we encountered numerous challenges, which can be broken down into three categories: 1) learning curve regarding WebAssembly workflow, 2) researching the state of parallelism support within WebAssembly, and 3) modifying the max pooling algorithm from using a GPU to using pthreads.

First was the challenge of the basic WebAssembly workflow. Within this, there were challenges with getting Emscripten installed and consistently working, in how to compile included libraries and header files to Wasm, and in allocating memory for Wasm. The compiled Wasm module is called from JavaScript, and it is necessary to pass in some arguments regarding the pages of memory to be made available to the Wasm sandbox. With limited, sometimes conflicting, documentation, it took some trial-and-error to determine how to allocate enough memory to store the image arrays.



	Your browser	Chrome 86	Firefox 79	Safari 13.1
	Standardized features			
JS BigInt to Wasm i64 integration	✓	✓	✓	✗
Bulk memory operations	✓	✓	✓	✗
Multi-value	✓	✓	✓	✓
Import & export of mutable globals	✓	✓	✓	✓
Reference types	✗	⌛	✓	⌛
Non-trapping float-to-int conversions	✓	✓	✓	✗
Sign-extension operations	✓	✓	✓	✗
	In-progress proposals			
Exception handling	✗	✗	✗	✗
Fixed-width SIMD	✗	⌛	⌛	✗
Tail calls	✗	✗	✗	✗
Threads and atomics	✓	✓	✓	✗

Fig. 4 Current supported features according to Web Assembly documentation

After addressing the overall WebAssembly workflow, researching and implementing parallelism within WebAssembly proved to be its own major set of challenges. First within this was simply determining the status of the various parallelism mechanisms. It was quickly made clear the GPUs were unsupported, but regarding threads and SIMD, it was much less clear. Some sources seemed to indicate the SIMD and threads were merely proposals, i.e., they were specifications, not implementations. Upon further research, it was determined they were indeed functional—if experimental—with the right compiler flags and the enabling of appropriate experimental browser settings. As can be seen in the image above, taken from the official WebAssembly website, official WebAssembly documentation itself was not completely clear either.

Finally, after addressing the previous two categories, there came the matter of translating the max pooling algorithm from CUDA using GPU threads to regular C using pthreads. The primary challenge here came not from the inherent difficulty of the translation, but from its interaction with Wasm parallelism. Because the primary way we tested if parallelism was by checking if there was speedup, a lack of speedup could mean Wasm parallelism was not working, our parallel algorithm was not working, or the compiler was not compiling to use true parallelism. There were multiple points of failure, and thus took much trial and error (and finally discovering the matter of experimental browser settings) before the algorithm worked with a significant speedup.

Evaluation of Performance

Description

Given that the aim of this project was to achieve parallelism within WebAssembly, we chose to evaluate performance in two ways. First was to measure speedup from a parallel algorithm to a sequential algorithm within WebAssembly. This was not only to measure the speedup, but to also verify that parallelism was indeed achieved. The second way we chose to evaluate performance was to compare the sequential WebAssembly implementation with a sequential JavaScript implementation. This was done, although not strictly in the scope of parallel computing, as a basic demonstration of WebAssembly's value in the first place even without parallelism. The parallelism WebAssembly offers would then be a terrific bonus in terms of performing intensive computations in-browser. This is mainly due to the fact that WebAssembly offers interpretation of parallelizing APIs such as the POSIX API in C: this means that Web Assembly can interpret our code that uses p-threads to divide up computational work into WebWorkers (background threads running in the browser) that will take on that work.

Results

Here are the results we found:

Case 1: Using multithreading with Web assembly

The following graph shows the average runtime over 10 trials per background thread:

Number of background threads	**Average Runtime (ms)**	*Trial#1*	*Trial#2*	*Trial#3*	*Trial#4*	*Trial#5*	*Trial#6*	*Trial#7*	*Trial#8*	*Trial#9*	*Trial#10*
1	28.31	*0.295*	*0.267*	*0.258*	*0.262*	*0.267*	*0.38*	*0.28*	*0.281*	*0.283*	*0.258*
2	16.429	*0.213*	*0.213*	*0.1288*	*0.1482*	*0.1419*	*0.179*	*0.16*	*0.146*	*0.164*	*0.149*
3	14.48	*0.133*	*0.161*	*0.142*	*0.128*	*0.141*	*0.181*	*0.193*	*0.1*	*0.125*	*0.144*

4	13.05	*0.158*	*0.133*	*0.113*	*0.153*	*0.124*	*0.131*	*0.116*	*0.131*	*0.136*	*0.11*
5	11.64	*0.11*	*0.115*	*0.099*	*0.155*	*0.114*	*0.102*	*0.108*	*0.123*	*0.112*	*0.126*
6	10.09	*0.116*	*0.121*	*0.111*	*0.097*	*0.096*	*0.13*	*0.083*	*0.078*	*0.087*	*0.09*
7	8.97	*0.106*	*0.103*	*0.093*	*0.095*	*0.095*	*0.083*	*0.08*	*0.075*	*0.093*	*0.074*
8	7.99	*0.075*	*0.115*	*0.075*	*0.075*	*0.075*	*0.075*	*0.079*	*0.078*	*0.075*	*0.077*
9	7.75	*0.076*	*0.072*	*0.078*	*0.073*	*0.076*	*0.078*	*0.086*	*0.077*	*0.083*	*0.076*
10	7.74	*0.075*	*0.081*	*0.078*	*0.072*	*0.071*	*0.074*	*0.088*	*0.076*	*0.078*	*0.081*

Fig. 5 Average runtime (ms) of parallelized C implementation of Max-Pooling using WebAssembly

The graph illustrates the average measure runtime vs the number of threads running:

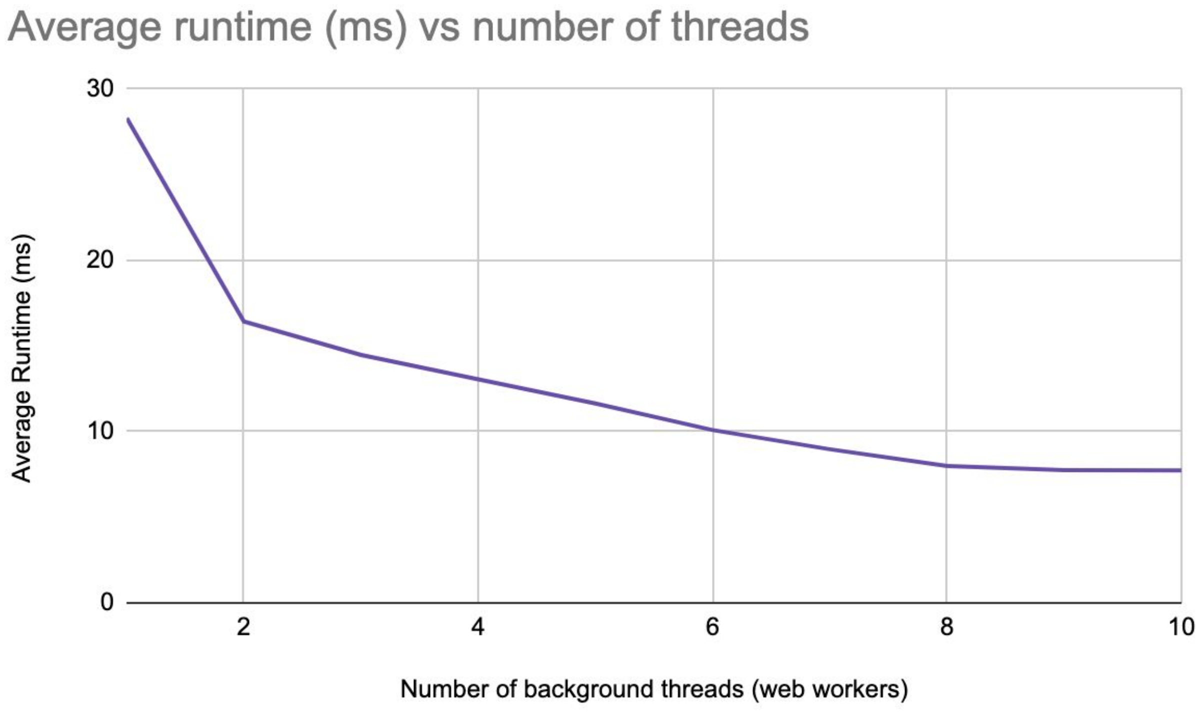


Fig. 6 Average runtime (ms) of parallelized C implementation of Max-Pooling using WebAssembly vs number of threads

The calculated speedup is:

Number of background threads	Average Runtime (ms)	Speedup
1	28.31	1
2	16.429	1.72
3	14.48	1.96
4	13.05	2.17
5	11.64	2.43
6	10.09	2.81
7	8.97	3.16
8	7.99	3.54
9	7.75	3.65
10	7.74	**3.66**

Fig. 7 Speedup of parallelized C implementation of Max-Pooling using WebAssembly vs number of threads

This table gives the following graph in which we can better see progression of the speedups:

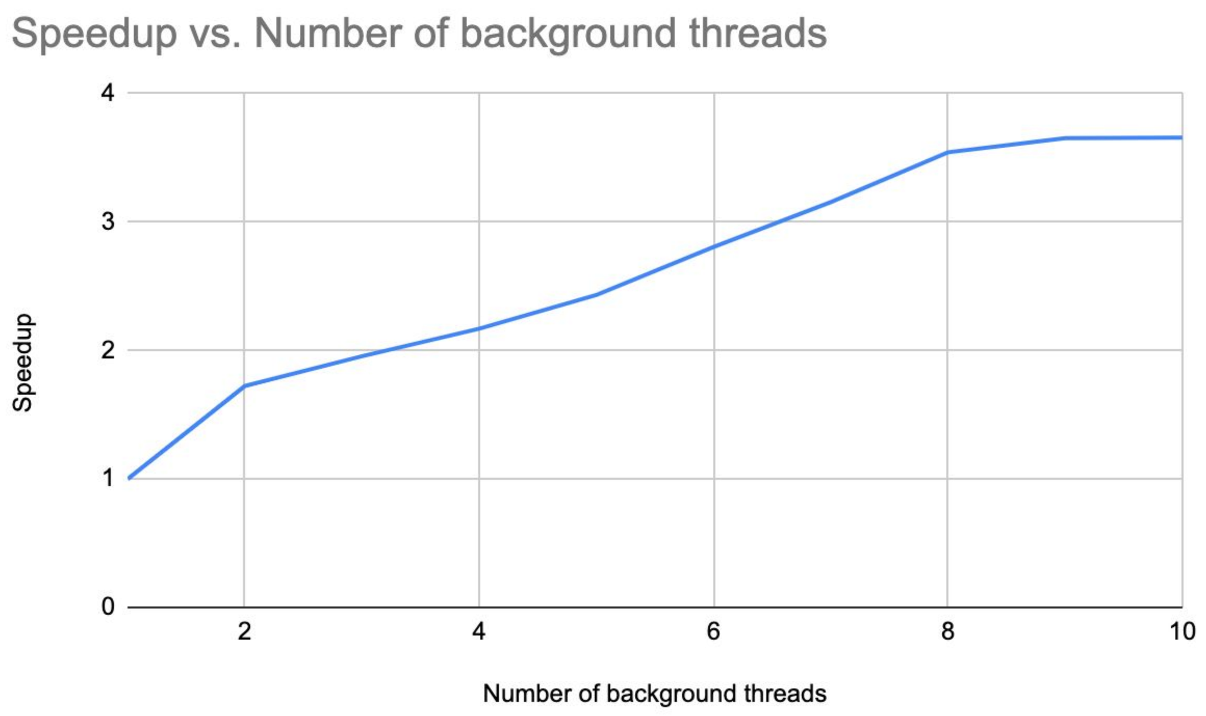


Fig. 8 Speedup of parallelized C implementation of Max-Pooling using WebAssembly vs number of threads

Case 2: Using WebAssembly sequentially

In this case, we run our code on 1 thread, being the main UI thread (no background threads/WebWorkers). The code running on that thread is written in C and compiled to WebAssembly.

We get the following average runtime after running 10 trials to measure the runtime:

Number of threads	**Average runtime (ms)**	*Trial# 1*	*Trial# 2*	*Trial# 3*	*Trial# 4*	*Trial# 5*	*Trial# 6*	*Trial# 7*	*Trial# 8*	*Trial# 9*	*Trial#10*
1	243	*245*	*239*	*240*	*252*	*239*	*242*	*241*	*242*	*245*	*245*

Fig. 9 Average runtime (ms) of sequential C implementation of Max-Pooling using WebAssembly

Case 3: Using JavaScript directly on the browser

In this case, we run our max pooling directly on the browser without using WebAssembly. This means that we have to run a JavaScript implementation of max pooling on the main UI thread of the browser. With that in mind, we get the following average runtime after 10 measurements:

Number of threads	**Average runtime (ms)**	*Trial# 1*	*Trial# 2*	*Trial# 3*	*Trial# 4*	*Trial# 5*	*Trial# 6*	*Trial# 7*	*Trial# 8*	*Trial# 9*	*Trial#10*
Main UI thread (1)	613	*610*	*621*	*615*	*607*	*600*	*620*	*616*	*611*	*618*	*612*

Fig. 10 Average runtime (ms) of sequential JavaScript implementation of Max-Pooling

Interpretations and conclusions

We obviously see that the slowest runtime is when running the JavaScript sequential implementation of the max pooling, giving us a runtime of 613 ms on average. This makes sense, as the code is written in a higher level language and is running on 1 thread, being the main UI thread exposed by the browser (open developer tools on the browser, and whatever JavaScript code written there will run on the Main UI thread). We get an increase in performance of 60% when running the WebAssembly version of max-pooling again on only 1 thread, with a runtime of 243ms. This goes in accordance to the explained benefits of using WebAssembly, which gives an increase on performance as it interprets lower level code. However, we can increase performance by 98% when using more than 8 threads in the WebAssembly version of max-pooling, with a runtime as low as around 7 ms. This makes sense since each pthread divides up the work that needs to be done to finish the max pooling computation, and WebAssembly takes care of assigning the work of each p-thread to a background thread (WebWorker) running on the browser. We can clearly see on the speedup graph that we have a higher speedup with higher number of pthreads/WebWorkers, but we do not exceed a max speedup of 3.5-3.6 (or a runtime of 7ms). This is because, depending on the CPU specifications that run the max pooling, WebWorkers use a certain percentage of the CPU capacity. For instance, in a 4-core CPU such as the one that gave the previous data, a max of 8 WebWorkers can run in parallel so 8 WebWorkers use 100% CPU capacity, so using more than 8 WebWorkers gives the same performance[8]. This explains why using more than 8 pthreads gives approximately the same performance. All in all, we can see that we are to run a multi-threaded code to the browser using WebAssembly, which gives a significant boost in performance (98%).

Potential Applications

A major advantage of multi-threading with WebAssembly is the major increase in performance with the ability to run low level code on the browser that was originally not meant to be for the browser. This boost in performance (refer to previous section with Interpretations and conclusions of result) can be a valuable asset when performing many computationally expensive tasks, such as 3D image rendering. Mears explains in his article[9] how this was an advantage for running Google Earth in the browser, which took advantage of WebAssembly's multithreading proposal. For instance, his data[9] shows a higher image frame rate using multi-threaded WebAssembly (>40 fps) due to the rendering of the imagery being faster.

Another example is within the gaming industry. WebAssembly provides a means to port games to the web. It brings the performance of native applications to the web. Unlike other approaches that have required plug-ins to achieve near-native performance in the browser, WebAssembly runs entirely within the web platform. This means that developers can integrate WebAssembly libraries for CPU-intensive calculations (e.g. compression, face detection, physics) into existing web apps that use JavaScript for less intensive work. Moreover, users can exploit existing libraries written in C for their games and still be able to run on the web. With the launch of Unity 2018.2 release, Unity made the switch to WebAssembly as their output format for the Unity WebGL build target[6], while other gaming platforms are following the lead.

Statement of Contribution

The work for this project was split across all four members of the team. Garrett was responsible for setting up the github repository. Ahmed researched WebAssembly. Moreover, Garrett researched the states of parallelism in WebAssembly, and made the design decision to use pthreads to exploit parallelism in WebAssembly. Anssam and Garrett researched ways to install webassembly and emscripten to run a simple ‘Hello World’ program to test the installations. In addition to this Elsa researched ways to handle parsing image inputs to the wasm environment. Anssam and Elsa then implemented a simple Fibonnaci program to test the use of pthreads and run them with WebAssembly on the browser. Anssam and Garrett then used the max pooling implementation of lab 0, to replace the GPU thread calls to a pthread implementation and updated the implementation based on the needs of our project. Anssam, Elsa and Garrett implemented the 3 cases that we are evaluating i.e the sequential max-pooling that runs directly on the browser, the sequential max-pooling that ran using webassembly and the parallelized max-pooling that ran using WebAssembly. Anssam then collected the performance results. Elsa researched the core limitation results that we observed due to the number of WebWorkers on Chrome 70 and how the tasks were partitioned on the 4 cores.

References

1. Barney, B. (n.d.). Retrieved December 10, 2020, from https://computing.llnl.gov/tutorials/pthreads/
2. Arias, D. (2018, August 30). Speedy Introduction to Web Workers. Retrieved December 10, 2020, from https://auth0.com/blog/speedy-introduction-to-web-workers
3. Developers, G. (n.d.). WebAssembly Threads ready to try in Chrome 70 | Google Developers. Retrieved December 10, 2020, from https://developers.google.com/web/updates/2018/10/wasm-threads
4. Chrome, G. (n.d.). Chrome OS. Retrieved December 10, 2020, from https://www.scribd.com/doc/123997185/Chrome-OS
5. 15 Jan 2020Mike Bursell (Red Hat, C., Levick, R., & Kalin, M. (n.d.). Why everyone is talking about WebAssembly.Retrieved December 10, 2020, from https://opensource.com/article/20/1/webassembly

6. Lahoti, S. (2018, August 16). Unity switches to WebAssembly as the output format for the Unity WebGL build target. Retrieved December 10, 2020, from https://hub.packtpub.com/unity-switches-to-webassembly-as-the-output-format-for-the-unity-webgl-build-target/

7. Maisonneuve, S. (1963, June 01). What about multiple core load management with Web Workers? Retrieved December 10, 2020, from https://stackoverflow.com/questions/23621883/what-about-multiple-core-load-management-with-web-workers

8. Mozilla, D. (n.d.). Navigator.hardwareConcurrency. Retrieved December 10, 2020, from https://developer.mozilla.org/en-US/docs/Web/API/NavigatorConcurrentHardware/hardwareConcurrency

9. Google, E. (2019, May 22). Performance of WebAssembly: A thread on threading. Retrieved December 10, 2020, from https://medium.com/google-earth/performance-of-web-assembly-a-thread-on-threading-54f62fd50cf7

10. Doglio, F. (2020, August 12). The JIT in JavaScript: Just In Time Compiler. Retrieved December 10, 2020, from https://blog.bitsrc.io/the-jit-in-javascript-just-in-time-compiler-798b66e44143

11. Clark, L. (n.d.). A crash course in just-in-time (JIT) compilers – Mozilla Hacks - the Web developer blog. Retrieved December 10, 2020, from https://hacks.mozilla.org/2017/02/a-crash-course-in-just-in-time-jit-compilers/

12. Clark, L. (n.d.). What makes WebAssembly fast? – Mozilla Hacks - the Web developer blog. Retrieved December 10, 2020, from https://hacks.mozilla.org/2017/02/what-makes-webassembly-fast/