Image Processing using JavaScript and Web Assembly

Student Name: Sam Robbins

Supervisor Name: Tom Friedetzky

Submitted as part of the degree of BSc Computer Science to the

Board of Examiners in the Department of Computer Sciences, Durham University

Abstract —

A   Context/Background

Web Assembly is a relatively new method for computation on the web, allowing for the use of a much wider range of languages. This is being used in high intensity contexts to improve computation time.

B   Aims

Basic Aim: Compare one complex algorithm
Intermedia Aim: Compare multiple complex algorithms
Advanced Aim: Look into tweaks that can be used to improve performance

C   Method

Implement both high complexity and low complexity image processing algorithms in both JavaScript and Web Assembly, then look into if there is any possibility of optimization. Then measure these algorithms to determine if there is a benefit to using Web Assembly.

D   Results

Web assembly can offer an improvement of over $8\times$ compared to JavaScript in some tasks. However, the implementation of the algorithm can often be more important than the language it's running in, and if it's possible to use neither Web Assembly nor JavaScript, then this is often the best choice.

E   Conclusions

Web Assembly has the capability for providing large increases in performance, however it is not a magic bullet for performance, and the individual task needs considering before using it.

Keywords — Web Assembly, JavaScript, Image Processing

I   INTRODUCTION

Image processing is a widely used technique for a range of problems. In most modern cameras there will be some aspect of image processing at the time of capture, whether it is enhancements to the image or an image codec, such as JPEG. After capture there is often additional image processing, be it applying filters or compression for upload online. Sophisticated image processing techniques don't just have benefit for the sharing of photos, but are also used medically, such as for processing CT images (Zhang et al. 2017).

The performance of image processing was greatly improved with the introduction of Digital Signal Processors (DSP), these are specialized chips for performing signal processing tasks, such as the discrete cosine transform. In mobile devices these are now often integrated into the System on a Chip (SOC) (Angoletta 2008), however with the increase in computational performance since DSPs were introduced, these algorithms can also be run on the main processor.

Traditionally image processing has been done in applications like Adobe Photoshop, however as the amount of time people spend in browsers is increasing web based photo editors are becoming more common. In a study by Forrester Consulting, workers are spending 1/3 of the work day on average in a web browser, so it is an area many companies are targeting to launch new products (Forrester 2018).

In 1995 JavaScript was introduced as a method for introducing interactivity to web applications, since then there were some improvements made for programmers who wanted to perform computationally intensive tasks with their website, such as Web Workers, introduced in 2009 (Hickson 2009). However, one of the biggest steps forward is the introduction of Web Assembly in 2017, allowing for assembly code to be executed by web browsers (Haas et al. 2017).

Web Assembly is a compilation target for languages that support it, meaning that you take the code in the language of your choice and compile it into a `.wasm` file. This file can then be called using JavaScript to make it behave like a standard JavaScript library.

Using Web Assembly over JavaScript proposes advantages such as being able to use your existing codebase, rather than having to translate it into JavaScript. There are also additional benefits depending on what language is chosen, such as static types, which can allow for easier debugging. JavaScript offers this through TypeScript, but as it is an additional feature, it isn't as prevalent as a language that offers it natively.

When it comes to producing a web application, the computation can be run on either the client (web browser) or the server, each coming with advantages and disadvantages. Running code on the server allows for the site to be performant on low power devices such as mobile phones, however by running code on the client it isn't required to send data between the client and server on every interaction. This allows the site to continue being ran during times of intermittent connection and reduces the latency of operations. Running code on the client also allows for the site to be turned into a progressive web app, which only need to be connected to the internet once for installation similarly to an app (Biørn-Hansen et al. 2017).

A   Research Question

The Research question is to find in which cases Web Assembly offers a benefit over JavaScript, finding this out by implementing image processing algorithms using both mechanisms.

B   Objectives

Basic Aim: Compare one complex algorithm
Intermedia Aim: Compare multiple complex algorithms
Advanced Aim: Look into tweaks that can be used to improve performance

II   RELATED WORK

A   *Language options for Web Assembly*

Web Assembly is compatible with any language that compiles to assembly, meaning there is a large range of languages to choose from. However, as Web Assembly doesn't currently have a garbage collector, it is most efficient to choose a language that doesn't do garbage collection, such as C (Haas et al. 2017). Garbage collection is the process of reclaiming memory that is no longer in use by the program, this helps to reduce the memory impact of a program. Alongside C, Rust is another language that doesn't have garbage collection and advertises itself as good to use for Web Assembly.

One advantage of Rust is that it has a large ecosystem of packages to help with the production of Web Assembly code. One such library is wasm-bindgen which allows for binding functions and other features in Rust to the JavaScript produced, allowing for multiple functions to be included in one Web Assembly file (Rust and WebAssembly Working Group 2017). It also provides availability for using native browser functions, such as `window.alert`, within Rust functions. Another is wasm-pack which creates a JavaScript library containing the generated Web Assembly code, making it easier to integrate into a web application.

B   *History of Rust*

The Rust language came from a project by an employee of the Mozilla foundation and was supported by Mozilla and their employees. Since then, the ownership has been transferred to the Rust foundation in order to become more run by the community (Williams 2021). The intention of Rust is to offer a new alternative to C++, a language that first appeared in 1985, and so over this time has developed a range of legacy issues caused by having to not cause large changes that would require existing code to be changed.

C   *Important metrics*

There are a range of metrics that are important when evaluating the performance of web tools. Response time is one such metric, and measures the time it takes to receive feedback from an operation. This applies for both the time it takes for a page to load and when doing something on a webpage. Research by Nielson discovered that (Nielsen 1994):

- 0.1 second is the limit for instantaneous feeling

- 1.0 seconds is the limit for the user's flow of thought to remain uninterrupted

- 10 seconds is the limit for keeping the user's attention.

While this paper was written many years ago, the results are still regarded as correct with current computer systems, with the author writing an update in 2014 that:

> These guidelines have been the same for 46 years now, so they are also not likely to change with whatever implementation technology comes next. (Nielsen 2014)

This is because the measurements were made based on human psychology, rather than with interaction of a specific computer, allowing the results to remain applicable through the years.

Making these results actionable, Google introduced Core Web Vitals, which are metrics about how fast a page loads to determine how well it performs (Walton 2020). One such metric is Largest Contentful Paint, which measures the time taken for most of the page to be loaded in, and suggests that it should occur within 2.5 seconds to be classified as "Good".

D   Web Assembly Image Processing in use

Squoosh by Google Chrome is a tool to compress images and implements many of its codecs using Web Assembly. This approach was also adopted by Next.js for their image component to improve performance, reducing the installed size by 27.3 MB from a total installed size of 96.5 MB (Haddad 2021). This did lead to a large increase in the lines of code in the project as they were previously using an external package for this, however, as the popularity of Web Assembly increases, the number of packages to do this will increase. One example of such a package is `photon` which provides abstractions on top of the `image` Rust library (Odwyer 2019).

E   Performance of Web Assembly

Alongside image processing, Web Assembly is also used for a range of other purposes. Figma is a tool for designers, and implemented Web Assembly to improve their load times $3\times$ compared to their previous solution which translated assembly to JavaScript (Wallace 2017).

Another implementation of Web Assembly is in determining the quality of DNA sequencing data, as used by fastq.bio (Aboukhalil 2019). In this implementation they achieved a $9\times$ performance improvement, managing to increase that to $21\times$ with some additional optimizations such as using a single function call.

However, when running code natively, Web Assembly may not be the best solution. A JavaScript bundler is a program which creates large files from the smaller JavaScript files needed for a web application; this is useful as better performance can be achieved by sending a few large files compared to lots of smaller files. The author of a native JavaScript bundler observed that there was a $10\times$ speed improvement by running it natively, compared to using Web Assembly (Wallace 2020).

This behaviour was also observed in (Jangda et al. 2019), where peak slowdowns were observed of $2.5\times$ in Google Chrome compared to the native code. These benchmarks are taken from SPEC CPU2006 and SPEC CPU2017. A sample of these results are shown in Table 1.

Table 1: Comparison of algorithms between native and wasm

Benchmark	Field	Native time(s)	Google Chrome time(s)
bzip2	Compression	370	864
mcf	Combinatorial	221	180
milc	Chromodynamics	375	369
namd	Molecular Dynamics	271	369
gobmk	Artificial Intelligence	352	537

As one can see from these results, Native isn't guaranteed to be faster than Web Assembly, however this is often the case.

F   Current Image Processing Techniques

One of the most popular image processing libraries for JavaScript is Sharp with over 1,700,000 weekly downloads, this uses native code in the form of libvips and claims to be the fastest module for resizing. A slightly less popular library is jimp with over 1,400,000 weekly downloads, the difference with this library is that it is written entirely in JavaScript and so can be run in the browser. The difference between these and other libraries was measured on a task of resizing an image (Fuller 2020). These results are shown in Table 2.

Table 2: Performance of Image Processing Libraries

Library	Best ops/sec	Best speedup
jimp	0.77	1.0
mapnik	3.39	4.4
gm	4.33	5.6
imagemagick	4.39	5.7
sharp	25.60	33.2

This shows native libraries being significantly faster than the solution in pure JavaScript, which means that even with the slowdown of Web Assembly compared to native code as discussed before, Web Assembly should still outperform JavaScript in this regard.

G   Image formats

There are a range of image formats in use, and the encoding and decoding of images is a computationally intensive task. The Web Almanac studies over 7 million websites and found that 40.26% of images are of the jpg image format and 26.90% in png (Levien et al. 2020). JPEG is a high performance image codec with a study by Cloudinary showing JPEG to have an encoding speed of 49 MP/s and a decoding speed of 108 MP/s (Sneyers 2020). JPEG has both lossless and lossy versions, and in a study of lossless compression of 382 images, these results are shown in Table 3 (Ukrit et al. 2011).

Table 3: Compression speed and ratio for various algorithms

Algorithm	Compression Speed (MB/s)	Compression Ratio
Lossless JPEG	11.9	3.04
JPEG-LS	19.6	4.21
JPEG 2000	4.0	3.79
PNG	3.6	3.35

As one can see from these results, the compression ratio is very comparable between the various implementations of JPEG and PNG, however the compression speed greatly differs, with some JPEG algorithms significantly outperforming PNG.

III   SOLUTION

A   Platform

Web Assembly can run in a wide variety of places, there are the different browsers, Chrome, Firefox and Safari, along with different platforms, web and mobile. Each of these will implement their processing of both JavaScript and Web Assembly differently, which makes choosing the platform an important topic. Google Chrome has a majority market share, making it a good choice as the results are most widely applicable (Statcounter 2021).
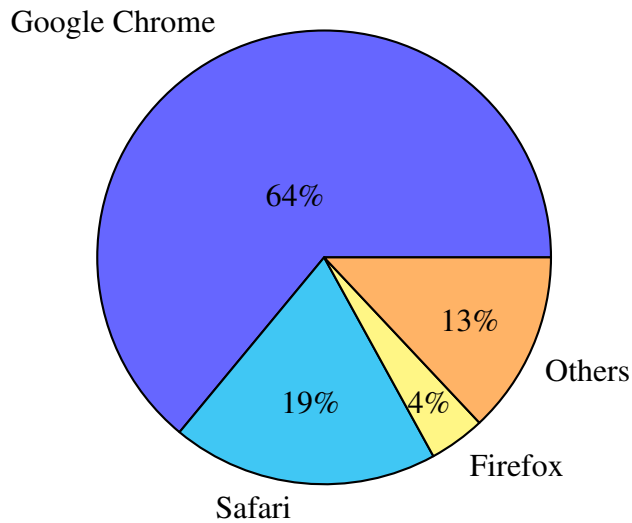

Figure 1: Browser Market Share

While the browser itself is important, the JavaScript engine is what will impact the results. This is the computer program responsible for executing JavaScript and Web Assembly code. The JavaScript engine inside Google Chrome is called V8, and is open source, allowing for other projects to use it. One such project is Node.js which allows for running JavaScript from the command line. This has become incredibly popular for the construction of web backends, with over 78,000 stars on GitHub. The benefit of using this over a browser is that the metrics will still be transferable to browsers, and allows for reducing the number of external factors that impact the computation. An example of this is that a browser will manage the resources available to each of its tabs, meaning that having more tabs open could affect the result, whereas the resources of Node.js are managed by the operating system. This also makes it easier to collect metrics such as memory usage and timing and the process runs as a system process.

B   Use of existing libraries

In order to speed up the development of areas whose performance I'm not measuring, I use existing image processing libraries to do some of the work. For example in decoding a provided image into the simple r,g,b values so that I can then run an encoder on them.

For Rust these libraries are referred to as "crates" and for JavaScript these are provided as npm libraries. Rust is both a newer language, and less widely used than JavaScript, meaning that

it has a smaller range of available libraries, so when it came to choosing a crate, there was only one major option, the `image` crate.

With JavaScript there is more choice, allowing for choosing based on the desired features. Jimp is being used for this experiment as it has a wide range of features and is very popular.

C Choice of algorithms

There is a wide range of algorithms to choose from when it comes to image processing. The algorithms have been chosen to ensure they cover a wide range of performance characteristics. Some of the algorithms require a large amount of processing and are expected to take over a second, whereas others just manipulate each pixel in turn and so are expected to complete much faster. This is to provide a full picture of if there is a certain type of algorithm that benefits from one of the two approaches.

When editing photos, the operation that tends to take the most time is exporting the images to JPEG, so I decided to explore implementing the JPEG algorithm (Int 1993).

D Implementation of JPEG

Both implementation use prewitten code to decode an image into a long string of r,g,b,a values and the width and height of the image. The first step is the transformation of the image from the r,g,b colour space into the YCbCr colour space. Doing this is the simple task of calculating fractions based on the r,g and b values, so is not especially intensive.

Next is downsampling; this isn't required by the JPEG standard, but can be used to further reduce image size. This reduces the resolution of the Cb and Cr components. It is done on these components as brightness (Y) is much more noticeable to the human eye compared to the other components, meaning the resolution can be reduced without a noticeable decrease in quality. To do this, each 2×2 pixel area is averaged to a single pixel. On the borders where it isn't possible to get this area, the average is just calculated based on the available pixels.

For the next part of the algorithm, each component is split into 8×8 blocks. This exhibits the same problem as the previous step as at the edges a full 8×8 block may not be possible. The implementation created solves this by replacing the pixels with black, however other implementations may use an average of the existing pixels to ensure there is less influence on the image.

The next step, the discrete cosine transform, is the most intensive step of the whole algorithm. The simplest way to implement this is to multiply each 8×8 matrix by another 8×8 matrix. For each entry in the resulting matrix there will be 8 multiplications. In my implementation I use the naive implementation of Matrix multiplication, giving $8^3 = 512$ multiplications in total per block. There is an opportunity to improve the number of multiplications required here using Strassen's algorithm, which would use $8^{\log_2(7)} = 343$ multiplications (Strassen 1969). However, it has been found that this algorithm is only better than the naive method for large matrices, and so on an 8×8 matrix, the naive algorithm has better performance (Huang et al. 2016).

The next step is quantization, this is a lossy operation to keep only the lower frequency data as the human eye struggles to distinguish the strength of high frequency brightness variation. In order to apply quantization, for each element in the matrix, it is divided by the value in the same position of a quantization matrix and rounded. JPEG has a variety of quantization matrices, allowing for the user to specify their desired quality, with the matrices having lower values for

higher qualities. Implementing the quantization matrix will lead to many of the elements being zero, allowing for easier encoding.

This encoding begins with run length encoding, intended for allowing for the encoding of runs of zeroes. This algorithm uses "zigzag encoding" in which the order in which the elements are processed is defined by a zigzag pattern, shown below.
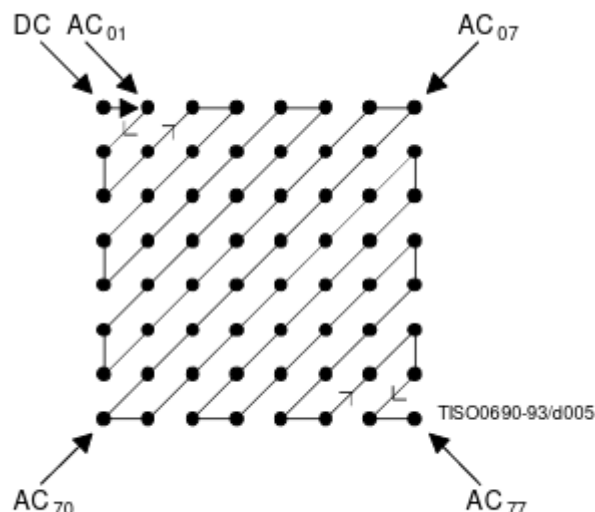
Figure 2: Zigzag order (Int 1993)

This then forms a sequence of values, ideally with many zeros at the end. As the length is something that is already known, the encoding can stop when the rest of the sequence is just zeroes. The most common encoding to use for this is Huffman, however the specification does allow for using arithmetic coding.

D.1 Writing to the file

The remainder of the algorithm involves correctly writing the data to a file. For this the source code for the JPEG encoders in the libraries discussed previously is being used, as this work isn't computationally intensive, but requires a lot of detail in order to ensure it is done correctly.

E Library provided functions

As both libraries provide algorithms for Gaussian Blur, Brightness and Contrast, they have been used for the testing, rather than writing the algorithms myself.

E.1 Gaussian Blur

Gaussian blur has a wide variety of applications, but is particularly useful for image denoising. The blurring effect helps to reduce noise artefacts caused by low light, and this is particularly useful in low end cameras such as mobile phones due to their small sensor size leading to poor performance in low light.

For the Gaussian Blur implementation, I have used the built-in implementations available in jimp and the image crate. These behave in a very similar way and so are suitable for comparison, and the same values are used for both implementation.

Below is an image before and after a Gaussian blur, for this comparison, the Gaussian blur has been done with a large sigma, resulting in a large blur to make it clear what the process is doing, however in denoising, this value will be much lower.

Figure 3: Before blur

Figure 4: After blur

E.2 Brightness

The implementation of brightness has two sections, depending on if the provided value is positive or negative. If the value is negative then the desired operation is to reduce the values of all the pixels, and so needing to multiply by a value less than one. So for a pixel value V and brightness value B, the new pixel value will be V'.

$$V' = V \times (1 + B)$$

However, if the brightness value is greater than 1, then the new pixel value will be

$$V' = (255 - V) \times B$$

E.3 Contrast

The exact implementation of contrast can vary, but the basic algorithm follows the following structure:

The user provides a value C of the contrast they want implemented, from this F is calculated in the following manner:

$$F = \frac{259(C + 255)}{255(259 - C)}$$

Then for each channel (r, g or b) on each pixel, the new value V' is calculated from the initial value V

$$V' = F(V - 128) + 128$$

9

F   Web Assembly optimizations

JavaScript comes with no options for optimizing performance, as it is run by the browser as JavaScript. However, as Rust is first compiled to assembly, it offers a range of ways to trade off between size and speed, and some methods that have the potential to improve both. Each of the following methods can be used at the same time to provide further improvements.

1. Link Time Optimization

   This takes the intermediate representation of the code, allowing the whole codebase to be optimized together, rather than optimizing each file individually. Doing this can help to improve both the size and performance of the code.

2. Rust compiler opt-level

   This allows the user to specify how far they are willing to optimize size at the detriment of performance, this has two options, `s` and `z`, with z being the most optimized for size.

3. `wasm-opt`

   This is a tool which allows for optimizing a `.wasm` file after it has been created, to optimize for either speed or size. It provides the following options:

   - `-Os` : optimize for size
   - `-Oz` : optimize aggressively for size
   - `-O` : optimize for speed
   - `-O3` : optimize aggressively for speed

F.1   Results

The performance of the various options are to be evaluated in order to determine what should be used when collecting the results, as both JavaScript and Web Assembly should be performing at the best they can be made. For this experiment, the Gaussian blur method has been used as it is computationally intensive.

For the Web Assembly optimizations, the measurements are based on the Gaussian blur task. The first measurement to take is how long each process takes.
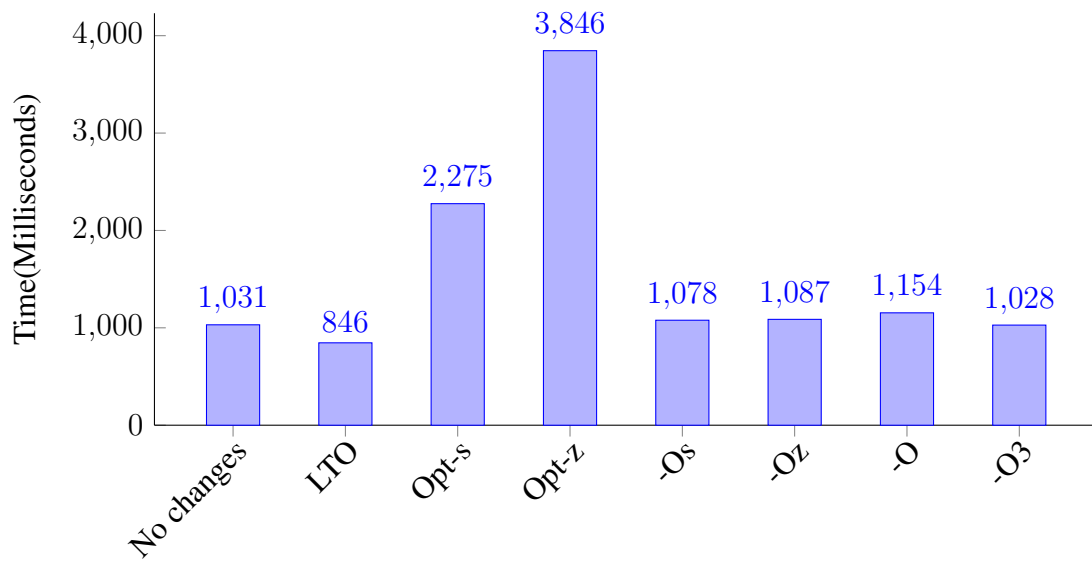
Figure 5: Processing time of Web Assembly optimizations

These measurements need to be contrasted with the file sizes they produce to fully evaluate the benefit of each optimization, as if for the small increase in performance there was a large cost in file size it would not be feasible.
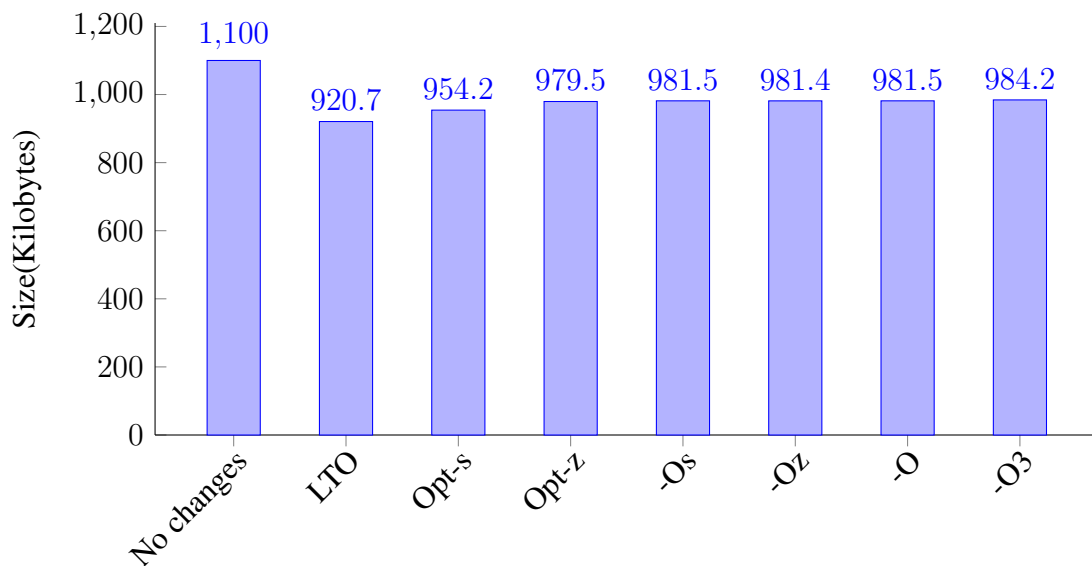

Figure 6: File size of Web Assembly optimizations

From the results, the file size isn't greatly affected by the optimization method chosen, however choosing any does provide a reduction in comparison to not choosing any optimizations. However, when it comes to computation time, there is a large range of results, with only Link Time optimization providing an improvement over not making any changes, with the methods designed to provide a smaller file size taking considerably longer, despite not leading to a large difference in file size. Overall, compiling with Link Time Optimization seems to provide an overall better performing program, both when it comes to size and speed.

G   Testing

For JPEG, as the intensive section is the multiplication of matrices, there aren't properties of the image that lead to one taking longer than another. The exception to this is the overall size of the image as this determines the number of blocks that are created from the image. So for evaluation of the algorithms the size is the only factor to consider when choosing an image.

The other algorithms similarly don't change with anything other than the size of the image as they are going over each pixel of the image and performing manipulation on them, so the choice of image isn't important for these either.

The first evaluation is to determine the running time of the algorithm. The for JavaScript the testing method uses the built-in `console.time` method in JavaScript to time how long a function takes. In Rust the `web_sys` crate is used, which allows for the performing of the `console.time` method over commands in Rust, helping to gather more accurate measurements.This is measured just over the actual function that performs the algorithm, rather than the whole process, so get more accurate results about the algorithm itself.

Memory usage is also an important metric, especially on mobile devices that might have small amounts of memory. This can be measured using the `/usr/bin/time` command on Linux devices, which measures both the time taken and memory usage of a command. This metric is less accurate than the timing evaluation already discussed as it measures the memory usage of the whole command, rather than just of the JPEG conversion. To try and counteract this difference, the code outside the JPEG conversion is kept as similar as possible to ensure it has the same memory consumption.

In order to improve the accuracy of the calculations of the Web Assembly implementation, I have used the `web_sys` crate which allows me to run `console.time` inside my rust code. This displayed a discrepancy of around 100ms between the internal and external time, which is insignificant on some of the longer running programs, but is useful for the shorter running programs.

A full profiler hasn't been used here as there aren't widely used JavaScript profilers that evaluate at the level of how long each function takes. Instead, they focus on things such as response time and CPU utilization of the overall program, which doesn't deliver the results I'd look for.

H   Verification and Validation

In order to verify the testing setup was only measuring the algorithm, an experiment was done to test if there were any constants involved in the implementation of brightness. To do this, a 100 MP photo was used, and scaled down to increments of 10 MP between 10 and 80 MP. If there was no constant, the expected result would be for the y-axis intercept to be zero, otherwise the intercept will show what the constant is.
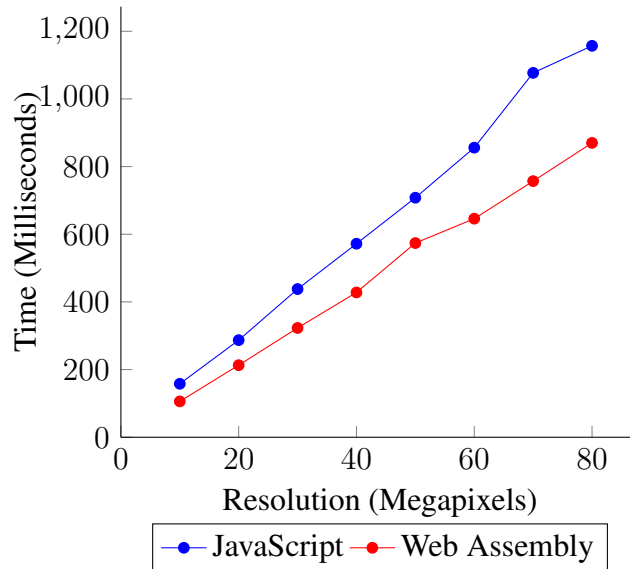
Figure 7: Relationship between time and resolution

Extrapolating the results found to a resolution of 0 megapixels, it can be seen that the y-axis intercept is very close to zero. This verifies that there isn't a significant constant included, and so that the results are more accurate.

To verify the results, they are compared against a basic conversion using imagemagick, this is a popular command line tool for converting between image formats. Due to this not having the overhead of either Web Assembly or JavaScript it is expected for this to be faster than both implementations, but it allows for getting a rough idea of how long the algorithm should take. As this command does both the decoding from one format and the encoding to another, it is required to also measure my implementations in the same way for verification.

I *Stages of the life cycle undertaken*

The development of this project was done using the Waterfall methodology (Royce 1987). The main components of this are shown in Figure 8.
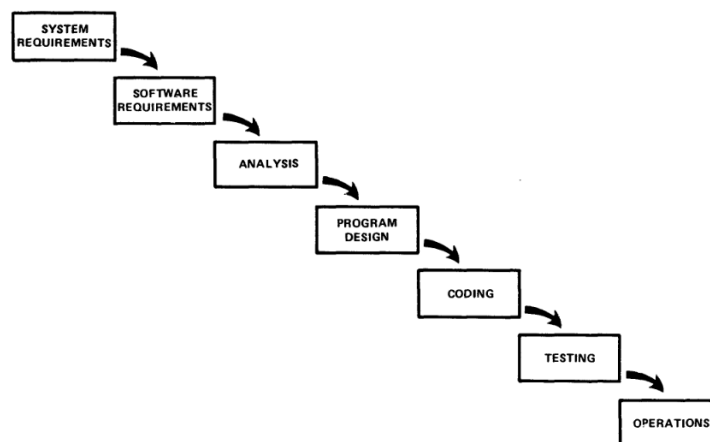

Figure 8: Waterfall process

13

Emphasis was particularly places on the system requirement and testing sections of this cycle, ensuring that appropriate algorithms were used and that testing was done to see if they followed the expected pattern or produced different results.

IV   RESULTS

Testing was done on a 24MP, 11 Megabyte, PNG format photo of a deer. The computation was done on an Intel i5 6600 processor with 16 GB of RAM on Node.js version 14.6.0.

The below chart shows the time comparison of the JavaScript and Web Assembly implementations.
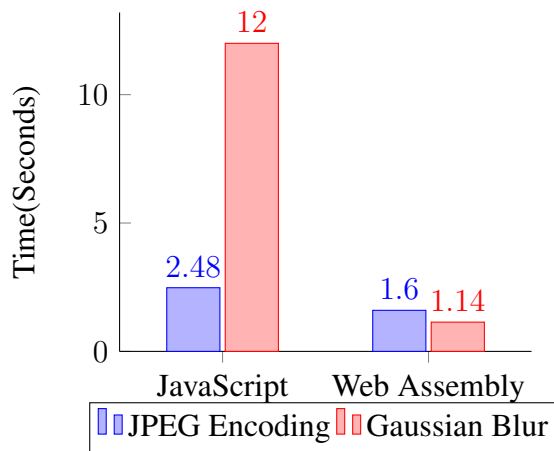

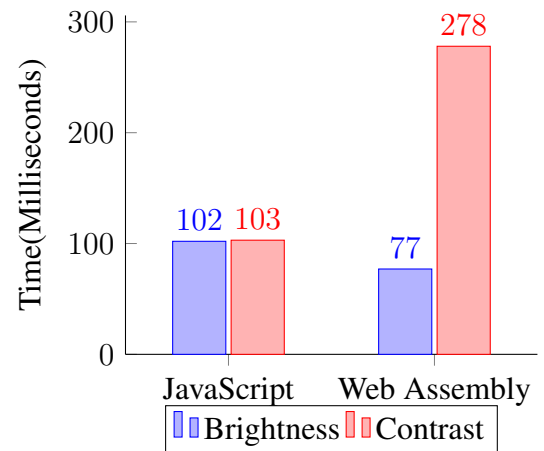Figure 9: Algorithm timings

Figure 10: Lightweight algorithm timing

In addition to time taken, memory consumption was also measured for the JPEG encoding, including comparison to imagemagick, a command line conversion tool. This measurement was only applicable for JPEG encoding as in every process imagemagick will do image encoding to produce an output, so the measurement wouldn't just be measuring the algorithm in question.
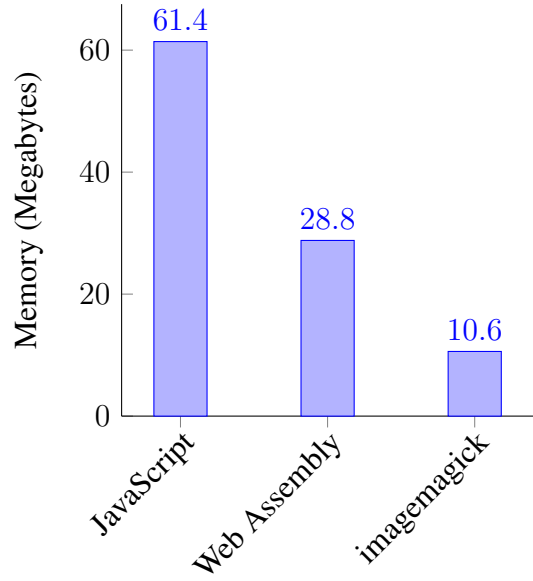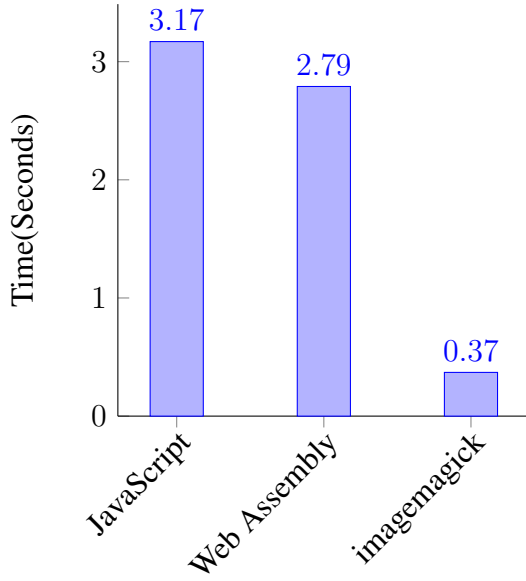
Figure 11: Time taken for the full process

Figure 12: Memory consumption of process

V EVALUATION

A Solution Strengths

The Web Assembly solution offers significant improvement over JavaScript for complex algorithms, being $8.3\times$ faster at performing the Gaussian blur task and $1.55\times$ faster at performing JPEG encoding. For the Gaussian blur, with a processing time of 12 seconds for JavaScript, this takes it above the 10-second limit for keeping the user's attention, as discussed in (Nielsen 1994), giving Web Assembly a big advantage in making the application feel responsive. These improvements are roughly in line with the results found by fastq.bio in their implementation (Aboukhalil 2019).

The Web Assembly implementation also has much lower memory consumption, which is important for low power devices such as mobile phones with low memory capacity. This is likely due to the increased control the language has for memory management, allowing the programmer to specify how much memory they want and when they don't want it anymore. In contrast, JavaScript handles all the memory management itself, making assumptions about when memory can be freed.

Rust has a big advantage in developer experience for having static types, meaning that the language server is capable of telling the developer what parameters a function takes, and what their types are. This is very useful for debugging as it may be the case that an incorrect type of variable has been passed to a function, in JavaScript this could either lead to unexpected behaviour, or cause an error further down the chain, whereas in Rust, this problem will be immediately highlighted to the user in their IDE. This also makes it easier to remember what functions do without looking at documentation as you can see what parameters they need and what type of data you will get out from it. JavaScript does offer the ability for static types through Type-

Script, a superset of the language that includes types, however due to this being an addition to the language, not all packages provide it, so the benefit isn't as large as with Rust.

Being able to easily package up Web Assembly using a tool like `wasm-pack` helps to eliminate the complexity of learning a new language for some developers. This is because developers who create Web Assembly code can package it in a way that from the outside is indistinguishable from if it had been written in JavaScript. Other developers can then easily use this in their code without having to know anything about Web Assembly, and just enjoy the performance benefits it can provide.

B   Solution Limitations

Both the JavaScript and Web Assembly options had significantly worse performance in comparison to the native code of imagemagick, with Web Assembly being $7.5\times$ slower, which is in line with the results found in the related work section of this paper. There is still significant overhead to running code in websites, and the benefit of the flexibility has to outweigh it.

While I did discuss the benefits of static types in the strengths section, this also comes with its downsides in that it makes code much more verbose when having to include types for everything, and requires additional thinking to ensure everything has the correct type. JavaScript also has a much wider range of built-in functions which came in useful when creating the algorithm, such as the `map` function which condensed the size of the matrix multiplication code.

JavaScript also has the benefit of being able to write your entire codebase in one language, rather than having to separate out into a different language for some tasks. This approach has already gained popularity through the use of Node.js for a backend, allowing for the use of JavaScript there as well as in the frontend. It also reduces the complexity of the build step for the application, as it would be required to build each piece of web assembly, before finally bundling everything together to publish on the web. These disadvantages particularly take place in small tasks such as adjusting the brightness where all the effort of converting to Web Assembly saves 25ms.

The results show that with Web Assembly, the contrast implementation is slower than both the JavaScript implementation and the Web Assembly implementation of brightness. This was surprising as both algorithms took the same time with JavaScript. On closer inspection of the source code of both libraries, while jimp (the JavaScript library) was using a contrast algorithm like described in the solution section, image (the Rust Crate), used a different algorithm for calculating contrast which was significantly less efficient.

For the JavaScript implementation, the formula for the final V' was

$$V' = \frac{(C + 1) \times (V - 127)}{(1 - C)} + 127$$

whereas in Rust it was implemented as

$$V' = \left(\left(\left(\frac{V}{\max} - 0.5\right) \times \left(\frac{100 + C}{100}\right)^2\right) + 0.5\right) \times \max$$

Note here that the JavaScript takes in the decimal version of the contrast (e.g. 0.2), whereas Rust takes in a percentage (e.g. 20), so these functions will produce very different results for the same value of C.

16

The Rust implementation has significantly more usage of both multiplication and division. Note that the `max` value is the maximum value a pixel's colour can be, so 255 for standard images. Whilst it is possible that this contrast calculation provides a more desired effect, it also causes the decrease in performance found in the results.

However, on observation of the images, the effect seems to be extremely similar, I used a contrast of 20%, which produces a noticeable difference compared to the original image, but the two images look extremely similar.

Figure 13: Original Image

Figure 14: JavaScript implementation

Figure 15: Web Assembly implementation

C   Self appraisal

The decision to use Rust for this project was a good choice, providing a range of tools that made the development process easier, such as `wasm-pack`. However, it may also have been a good experience using C or C++ due to their age in that they would have more algorithm examples and more complete libraries.

The functions chosen provided the expected results, making them useful in proving my hypothesis, however it would have been good to be able to find an algorithm for which JavaScript was faster than Web Assembly. This may lie outside the realm of image processing algorithms, requiring a different approach entirely, or may have occurred when implementing the same algorithms in different ways, such as comparing iterative and recursive methods.

VI   CONCLUSIONS

In this paper, we have demonstrated Web Assembly offering considerable performance improvements over JavaScript, with it being up to $8\times$ faster. However, we also demonstrated areas where Web Assembly is not suitable, such as when you have the option to run native code. We also evaluated what options to choose when compiling web assembly, finding that Link Time Optimization can provide an improvement both in performance and file size.

A   Further work

One new development that could help with image processing on the web is WebGPU (GPU for the Web Community Group 2021). This is a standard that allows for access of the GPU by both JavaScript and Web Assembly. Due to the nature of image processing algorithms in performing parallel operations, such as in JPEG where much of the computation is done on the 8×8 blocks, each of these can be parallelized. As GPUs are designed for such operations, often having many more cores than a CPU, this could lead to a large improvement in performance.

This work could also be developed by evaluating methods outside of image processing. Looking at more basic algorithms like sorting and searching may help to reveal underlying reasons

why Web Assembly is faster or expose cases where JavaScript is faster. It may also be interesting to use multiple languages for Web Assembly to see if this benefit is restricted to Rust, or if it applies to all languages.

References

Rust and WebAssembly Working Group (2017), 'wasm-bindgen'.
URL: *https://github.com/rustwasm/wasm-bindgen*

Aboukhalil, R. (2019), 'How We Used WebAssembly To Speed Up Our Web App By 20X (Case Study)'.
URL: *https://www.smashingmagazine.com/2019/04/webassembly-speed-web-app/*

Angoletta, M. E. (2008), 'Digital signal processor fundamentals and system design'.

Biørn-Hansen, A., Majchrzak, T. A. & Grønli, T.-M. (2017), Progressive web apps: The possible web-native unifier for mobile development, *in* 'International Conference on Web Information Systems and Technologies', Vol. 2, SCITEPRESS, pp. 344–351.

Forrester (2018), 'Rethink technology in the age of the cloud worker'.

Fuller, L. (2020), 'Sharp performance'.
URL: *https://sharp.pixelplumbing.com/performance*

GPU for the Web Community Group (2021), Webgpu, Standard, W3C.
URL: *https://gpuweb.github.io/gpuweb/*

Haas, A., Rossberg, A., Schuff, D. L., Titzer, B. L., Holman, M., Gohman, D., Wagner, L., Zakai, A. & Bastien, J. (2017), Bringing the web up to speed with webassembly, *in* 'Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation', pp. 185–200.

Haddad, J. (2021), 'Remove sharp for wasm variant'.
URL: *https://github.com/vercel/next.js/pull/22253*

Hickson, I. (2009), Web workers, W3C note, W3C. https://www.w3.org/TR/2009/WD-workers-20090423/.

Huang, J., Smith, T. M., Henry, G. M. & Van De Geijn, R. A. (2016), Strassen's algorithm reloaded, *in* 'SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis', IEEE, pp. 690–701.

Int (1993), *INFORMATION TECHNOLOGY – DIGITAL COMPRESSION AND CODING OF CONTINUOUS-TONE STILL IMAGES – REQUIREMENTS AND GUIDELINES*.

Jangda, A., Powers, B., Berger, E. D. & Guha, A. (2019), Not so fast: analyzing the performance of webassembly vs. native code, *in* '2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)', pp. 107–120.

Levien, R. et al. (2020), *Web Almanac*, 2020 edn, HTTP Archive.

Nielsen, J. (1994), *Usability engineering*, Morgan Kaufmann.

Nielsen, J. (2014), 'Response times: The 3 important limits'.
   URL: *https://www.nngroup.com/articles/response-times-3-important-limits/*

Odwyer, S. (2019), 'Photon'.
   URL: *https://github.com/silvia-odwyer/photon*

Royce, W. W. (1987), Managing the development of large software systems: concepts and techniques, *in* 'Proceedings of the 9th international conference on Software Engineering', pp. 328–338.

Sneyers, J. (2020), 'How jpeg xl compares to other image codecs'.
   URL: *https://cloudinary.com/blog/how_jpeg_xl_compares_to_other_image_codecs*

Statcounter (2021), 'Browser market share worldwide'.
   URL: *https://gs.statcounter.com/browser-market-share*

Strassen, V. (1969), 'Gaussian elimination is not optimal', *Numerische mathematik* **13**(4), 354–356.

Ukrit, M. F., Umamageswari, A. & Suresh, G. (2011), 'A survey on lossless compression for medical images', *International Journal of Computer Applications* **31**(8), 47–50.

Wallace, E. (2017), 'Webassembly cut figma's load time by 3x'.
   URL: *https://www.figma.com/blog/webassembly-cut-figmas-load-time-by-3x/*

Wallace, E. (2020), 'esbuild'.
   URL: *https://esbuild.github.io/getting-started*

Walton, P. (2020), 'Web vitals'.
   URL: *https://web.dev/vitals/*

Williams, A. (2021), 'Hello world!', *Rust Foundation* .
   URL: *https://foundation.rust-lang.org/posts/2021-02-08-hello-world/*

Zhang, H., Zeng, D., Zhang, H., Wang, J., Liang, Z. & Ma, J. (2017), 'Applications of nonlocal means algorithm in low-dose x-ray ct image processing and reconstruction: A review', *Medical physics* **44**(3), 1168–1185.