Spring 2020

Benchmarking AssemblyScript for Faster Web Applications

Nischay Venkatram

Benchmarking AssemblyScript for Faster Web Applications

by

Nischay Venkatram

A report submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Engineering

Program of Study Committee:
Joseph Zambreno, Major Professor

The student author, whose presentation of the scholarship herein was approved by the program of study committee, is solely responsible for the content of this dissertation/thesis. The Graduate College will ensure this dissertation/thesis is globally accessible and will not permit alterations after a degree is conferred.

Iowa State University

Ames, Iowa

2020

TABLE OF CONTENTS

Page

LIST OF FIGURES

ACKNOWLEDGMENTS

Firstly, I would like to thank my parents and family for believing in me and being a constant source of motivation throughout my academic and personal endeavors. I would also like to thank Dr. Zambreno for his guidance and support throughout the research and writing of this paper. I am genuinely grateful to everyone who has helped and inspired me in any way throughout my career as a student and a software engineer.

ABSTRACT

As web applications are becoming increasingly complex, it is crucial now more than ever to be able to develop web apps with an emphasis on performance to ensure a responsive and smooth user experience. Since the introduction of Webassembly as a compilation target for the web, the promise of writing programs that can run at native speed seemed revolutionary in theory. But the real world performance benefits of Webassembly in comparison to Javascript is not clearly understood. This paper evaluates the current performance of Assemblyscript - a strict subset of TypeScript that compiles to Webassembly, and Javascript in the areas of numerical computing across multiple browsers. A set of benchmarks were developed in Assemblyscript that includes numerical computing problems from the Ostrich Benchmark suite. The tests were executed across Chrome and Firefox. After studying the results from the benchmarks that were created, we find that Assemblyscript demonstrates speedups that range between 1.1-7.2x. It is also noticed that writing idiomatic Typescript can slow down Assemblyscript in certain scenarios. In conclusion, this study suggests that Assemblyscript (and Webassembly) provides far more consistent and predictable performance in comparison to Javascript.

CHAPTER 1. Introduction

In the early 90's, web pages were static and did not support any dynamic behavior after a page was loaded in the browser. To enable scripting on the web, Brendan Eich and the Netscape team developed Javascript in 1995 [1]. This allowed developers to make web pages dynamic and perform simple tasks like form validation or HTML manipulation in the browser, without requiring any interaction with the server. Since then, Javascript has become one of the most prominent languages in the software engineering world. It is used across multiple different platforms and devices, and for a plethora of different applications. Over the past two decades, these applications have become incredibly complex and to ensure that users have a good experience, performance has become a key factor. Continuous performance improvements to browser engines and compilers eventually led to the creation of Webassembly - a new binary format that is secure, portable, and serves as a compilation target for typed languages like C/C++ and Rust, allowing faster computation times than Javascript [2]. This would allow developers to write the computationally intensive parts of an application in languages like C and compile it to Webassembly for faster performance. But software engineers predominantly working with Javascript may not have the expertise in languages like C/C++ and Rust and it is not ideal to learn another language to harness the benefits of Webassembly. This is why Assemblyscript was developed [3]. It is a strict subset of Typescript that compiles down to Webassembly, allowing developers to improve the performance of their application without having to become proficient in another language. Since the benchmarks for analyzing the performance of Assemblyscript for computationally intensive tasks are limited, that will be the contribution of this paper. In this study, five scientific computing problems will be used to measure and analyze the relative performance of Webassembly (compiled from Assemblyscript) and Javascript.

CHAPTER 2. Related Work

Since its inception in 1995 to be used as glue code within HTML, Javascript has become a widespread language being used across client side web applications, server side applications, mobile applications, and even desktop applications.

2.1 Javascript Performance

The increasing use of Javascript has prompted web browser developers to make significant improvements in Javascript compilers and engines over the years [4, 5]. Modern browser engines make use of JIT (Just-in-time) compilers [6, 7] to achieve optimizations while executing Javascript. In 2014, the performance of these engines was recorded to be close to 1.5-2 times that of native code [8]. The continuous evolution and improvement of these engines has made it possible for developers to build more complex applications in Javascript. Browsers are being used for computationally intensive tasks like numerical computing, cryptocurrency mining, and even machine learning and AI applications [9, 10]. The web has also become a target for graphics and game development over the last two decades due to the advancements made in browser engines [11–13]. Since providing users with a seamless user experience in any web application is crucial, research has been done to analyze and benchmark the performance of different graphics rendering techniques [14–17]. Additional efforts have been made to improve the performance of intensive Javascript applications and to avoid common pitfalls [18, 19]. A new specification that allowed Javascript as a compilation target for low level languages like C and C++ called asm.js emerged [20]. It is intended to have performance characteristics closer to native code. This was possible because it was a strict subset of Javascript, which allowed for ahead of time optimizations. Despite the performance improvements, asm.js inherits various Javascript issues like inconsistent performance, and overhead from parsing and

compiling the source. This is because asm.js needs to be transpiled to Javascript using a source-to-source compiler to run on the web.

2.2 Webassembly

All these efforts eventually led to the development of Webassembly which serves as a compilation target for typed languages like C/C++ and Rust with the intention of providing near native speeds [2]. Currently, the Emscripten toolchain built using LLVM [21, 22] compiles C and C++ to Webassembly, essentially enabling developers to run C and C++ in browsers. Binaryen is another compiler that allows developers to compile Assemblyscript to Webassembly [3, 23]. Similar compilers exist to cross compile Javascript or Typescript to Webassembly [24]. Webassembly is already faster and smaller than asm.js, and will only improve with features like multi-threading and SIMD support in development [25–27].

2.3 Benchmarks

To properly understand the performance benefits of using Webassembly, existing or new benchmarks will have to be utilized. Octane, Kraken, and Speedometer are suites provided by browser vendors to test the performance of browser engines [28–30]. Octane is no longer being maintained. In addition to Speedometer, the Webkit team has also created JetStream2 [30], a suite that also contains some benchmark tests on Webassembly. JSBench [31] suggests that benchmarks may not accurately measure the performance of real world applications and instead introduces a technique to generate benchmarks based on popular websites that match the behavior of real world applications. This was one of the major reasons why the Chrome V8 team retired octane, as micro-benchmarks were not a good representation of how browsers will perform while running modern web applications [28]. The Ostrich benchmark suite developed in the Sable lab at McGill University contains a set of 12 numerical computing problems [32] implemented in Javascript and C. Another benchmark suite was developed [33] using the Ostrich suite to compare the performance of Webassmebly compiled from C and Javascript. So while there are benchmarks for Webassmebly compiled from

other languages, the number of benchmarks comparing the performance of Webassembly compiled from Assemblyscript and Javascript are limited. That will be the focus of this paper.

CHAPTER 3. Methodology

To determine the process for the study, certain research questions need to be asked. Also, as this paper focuses on performance evaluation, specific benchmarks need to be selected to run experiments. The benchmarks, experimental platform, and execution process are described in the sections below.

3.1 Research Questions

Since Assemblyscript is relatively new, the performance benchmarks are fairly limited. This paper is trying to answer the following research questions:

- Does Assemblyscript provide performance benefits over Javascript?
- Are there cases where Assemblyscript is slower than Javascript?

Answering these questions would allow developers to make data driven decisions when choosing to use Assemblyscript. Assemblyscript can be used to alleviate bottlenecks in programs where it is the computationally better option and can be avoided in cases where regular Javascript outperforms it.

3.2 Benchmark Selection

Selecting the correct set of benchmarks is crucial as the results of this paper heavily depend on them. Since Assemblyscript is still in its early stages, any Javascript doesn't automatically become faster if it's implemented in Assemblyscript. As Webassembly runs in a sandbox execution environment [2], there is an overhead to exchanging data between Javascript and Webassembly. If the program uses too many managed objects that require memory management and garbage collection, Webassembly may not perform well as it isn't mature enough. Similarly, the current

version of Webassembly isn't a great fit for extensive DOM manipulation. On the other hand, Webassembly is really great for computationally intensive tasks [3]. This means that the selected benchmarks for this paper must meet certain criteria:

1. The benchmarks must be computationally intensive, allowing us to draw effective conclusions from the comparisons made with Javascript.
2. The benchmarks should produce correct results.
3. The benchmarks should should be reasonably complex. We do not want to micro-benchmark as that has its own disadvantages [28].
4. The benchmarks should be diverse in its applicability. We do not want to focus only on a narrow set of problems.

Based on the above criteria, the Thirteen Dwarf categories for numerical computing by a team at Berkeley [34] met the requirements. They identified 13 different categories of numerical computing problems that will be crucial to the field of computing in the upcoming decade. The Ostrich benchmark suite [32] was created based on those categories and implemented a computation problem for 12 of those in Javascript and a few other languages. For the purposes of this paper, 5 of those 12 problems were picked and are described below.

3.2.1 Breadth First Search

This algorithm falls into the category of Graph Traversal according to the Thirteen Dwarfs. This is simply a breadth first search on a randomly generated graph.

3.2.2 Fast Fourier Transform

This algorithm falls into the category of Spectral Methods according to the Thirteen Dwarfs. Spectral methods are a set of problems where the data is generally in the frequency domain, instead of the space or time domain [34]. In this problem, the FFT is applied to a randomly generated data set.

3.2.3 Lower Upper Decomposition

This algorithm falls in the category of Dense Linear Algebra according to the Thirteen Dwarfs. A Dense matrix is a matrix that has a high number of non-zero elements. Lower Upper Decomposition is performed on a randomly generated dense matrix.

3.2.4 Sparse Matrix Vector Multiplication

This algorithm falls into the category of Sparse Linear Algebra according to the Thirteen Dwarfs. A Sparse matrix is a matrix that has a high number of zeroes. Matrix multiplication is performed on a randomly generated sparse matrix and a randomly generated vector. A CSR (compressed sparse row) is used to represent the sparse matrix.

3.2.5 Page Rank

This algorithm falls into the category of Map Reduce according to the Thirteen Dwarfs. Map Reduce includes the set of problems where computation can be performed independently on multiple parts of the data and the results will be aggregated in the end.

3.3 Experimental Platform

3.3.1 Setup

The benchmarks were tested on a Lenovo T460 with an Intel(R) Core(TM) i7-6600U CPU @ 2.60GHz running Ubuntu 18.04.4 LTS. The test machine had 24 GiB of Memory and no dedicated GPU. The browsers selected for executing the benchmarks were Google Chrome 79.0.3945.117 and Mozilla Firefox 75.0.

3.3.2 Implementation

The implementation of the 5 problems described above existed for Javascript in the Ostrich benchmark suite. The code was updated to conform to modern Javascript standards. The same problems were implemented from scratch in Assemblyscript. Despite trying to keep the code as

identical as possible, there were certain parts that had to be implemented in a different way due to the current limitations of Webassembly. Since Closures are still in the proposal stages for Webassembly, any Javascript function that made use of closures had to be implemented to function correctly without using it in Assemblyscript. As Assemblyscript is a strict subset of Typescript, all the variables and objects needed to be strictly typed. Assemblyscript has a set of primitive types like i32 and f64. This is one of the reasons why Webassembly benefits in terms of performance as the types do not have to be resolved at runtime. Instead, the compiler will produce optimized code ahead of time.

The code implemented in Assemblyscript is compiled to Webassembly and these files are fetched and instantiated as modules in Javascript. The code was compiled with flags prioritizing speed over size as performance is the focus for this study. The exported Webassembly functions can then be invoked just like any other Javascript function. It is not straightforward to pass data between Webassembly modules and Javascript. Primitive types like numbers are supported, but strings and custom objects are not supported out of the box. The data has to be manually written to memory buffers and read on the other side. So to keep it reasonably simple, only primitive data is exchanged with the Webassemlby functions through function arguments and return values. To calculate the execution time for the program, a timestamp function was needed. The Web API includes a Performance interface that contains a function to return a high resolution timestamp. As Webassembly runs in a sandbox execution environment, it does not have direct access to Web API's and Javascript's in-built functions. Instead, it is possible to pass in functions during Webassembly module instantiation. This was done to access functions like console.log and performance.now. The runtime for the benchmark is computed within Assemblyscript itself and returned as a result. An alternative is to obtain the timestamps before and after each Webassembly function is invoked and calculate the difference. The problem with this approach is that it also includes the time taken to make the function call and exchange data between Javascript and Webassembly. This used to be slow in the past but has improved significantly [35]. The the cost of interoperability can be useful when choosing to make the trade-off between using Webassembly or Javasctipt because if

an application is making thousands of calls and cost for each call is much higher than the time to execute the function itself, then Javascript would be the better option. Even so, for the purpose of this paper we are focusing primarily on the execution time in the context of Webassembly and are not worried about the cost of interoperability.

3.3.3 Execution

Generally, it may not be sufficient to run the benchmark just once. Browsers will perform different optimizations based on the number of times the benchmark has been executed. The first iteration usually takes the longest time because browsers perform various optimizations for subsequent runs. Also, garbage collection happens periodically, which leads to slower runtimes for certain iterations. While in many use cases functions may only need to be executed once, it is still beneficial to know how the runtime changes across multiple iterations. The Jetstream 2 benchmark suite recommends 120 iterations [30]. In addition, the average case runtime is calculated. The average case tells us if most executions will run smoothly. The graphed data can be viewed within the browser itself in the benchmark suite. This suite makes use of Chartjs to visualize and display the result of the tests.

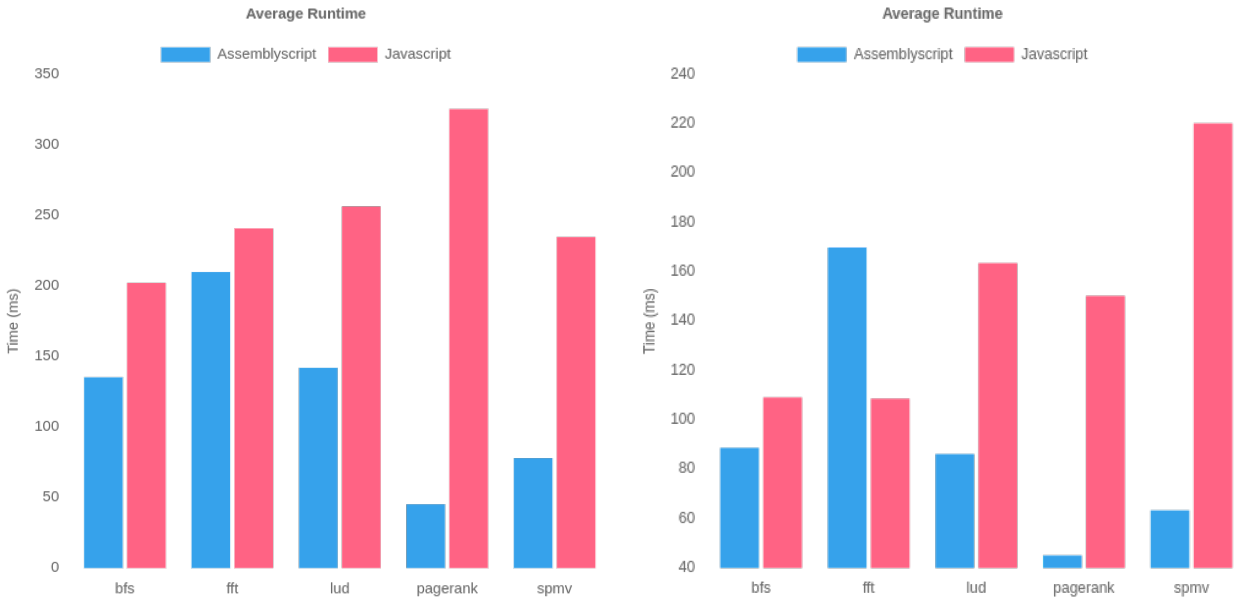
CHAPTER 4. Results

In the experiments that were performed, the performance of Assemblyscript is compared with Javascript across two major browsers - Google Chrome and Mozilla Firefox. Below, the first section discusses the results of the experiments and the second section discusses the insights from this study.

4.1 Javascript vs Assemblyscript

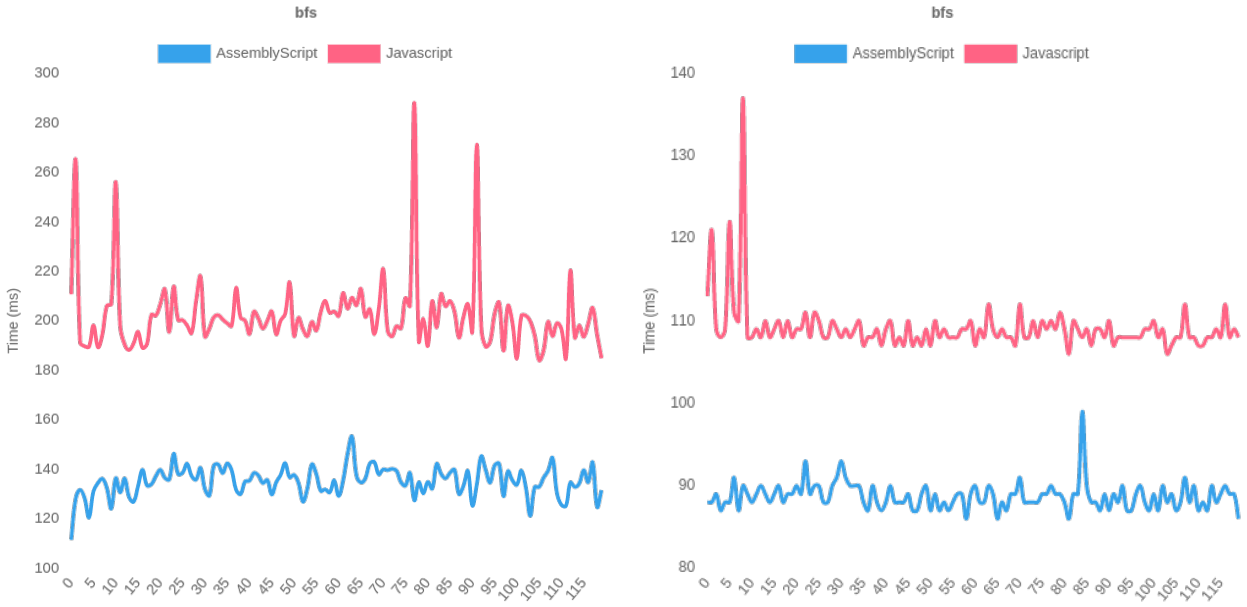
We will start by trying to answer RQ1, which is the major focus of this study. In 4.1a, we notice that Assemblyscript performs better than Javascript across all tests in Chrome. For *bfs*, *fft*, and *lud*, Assemblyscript is 1.5x, 1.14x, and 1.8x faster respectively. On the other hand, *pagerank* and *spmv* display more considerable improvements in performance. Assemblyscript performs 7.2x and 3x faster than Javascript for *pagerank* and *spmv*. We observe similar results on Firefox in 4.1b with the exception of *fft*. The speedup for *bfs* is slightly less on Assemblyscript, being only 1.2x faster. Javascript performs a lot better than in Chrome for *fft*, since it's 1.6x faster than Assemblyscript. Almost similar to Chrome, *lud* performs better on Assemblyscript with a speedup of 1.9x, while *pagerank* and *spmv* have speedups of 3.3x and 3.5x respectively. In general, Firefox appears to perform better or about the same in comparison to Chrome for all cases.

In 4.2a and 4.2b, we observe the performance of *bfs* over 120 iterations on Chrome and Firefox. While there are a lot of fluctuations in the performance, Assemblyscript performs better on every iteration. The occasional spikes in performance could be a result of garbage collection [30]. From 4.3 we notice that *fft* performs better in Assemblyscript on Chrome, while it is slower on Firefox. Despite being faster on average on Firefox, Javascript displays enormous spikes in the runtime. On the other hand, Assemblyscript performs consistently. Similarly, Assemblyscript performs really well over 120 iterations for *lud*, *pagerank*, and *spmv*. We observe consistent performance for Assemblyscript on those tests as well, while the runtimes for Javascript fluctuate considerably. One



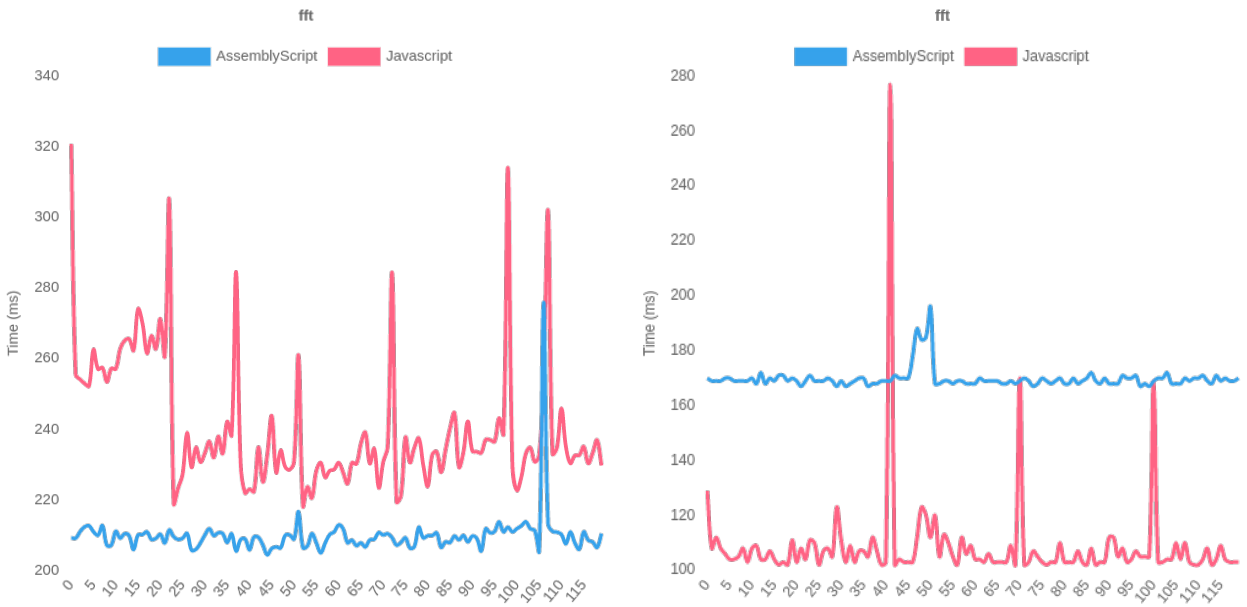
(a) Chrome (b) Firefox

Figure 4.1: Average Runtime



(a) Chrome (b) Firefox

Figure 4.2: Breadth First Search



(a) Chrome (b) Firefox

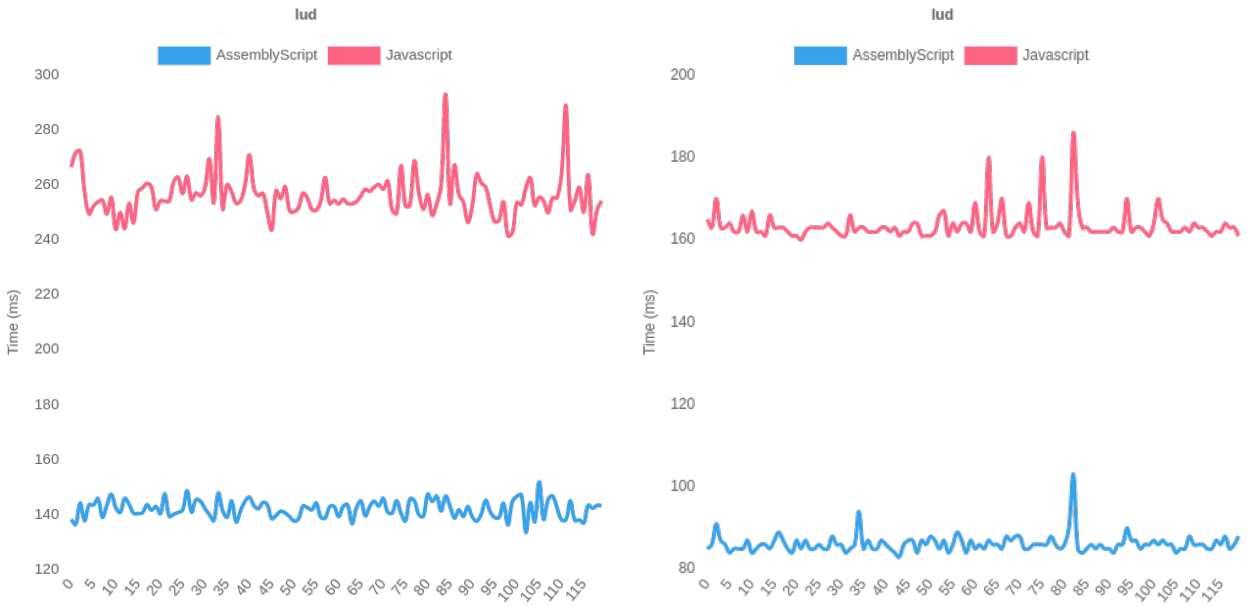
Figure 4.3: Fast Fourier Transform

of the things that stand out, for *pagerank*, the Assemblyscript runtimes are very consistent and the graph is almost a straight line with minimal fluctuations. Also, on both browsers in 4.6, *spmv* seems to perform well in Javascript (still slower than Assemblyscript) on the first iteration, but then drastically drops in performance for the remaining iterations.

4.2 Key Insights

The main observation from this study was that Assemblyscript performs better than Javascript in general across all the benchmarks except for Fast Fourier Transform on Firefox. The slowdown in this case is possibly because of reference counting overhead. Assemblyscript makes use of Reference Counting to manage memory [36]. In the *fft* implementation, a matrix transpose needs to be performed which requires matrix traversal. When accessing an element of a 2D matrix inside a doubly nested loop, the reference to the *i'th* row and *j'th* column is obtained and then released. In each subsequent *j'th* iteration, a new reference is obtained, used, and then released. The overhead in obtaining and releasing the reference for each iteration of the nested loop significantly impacts

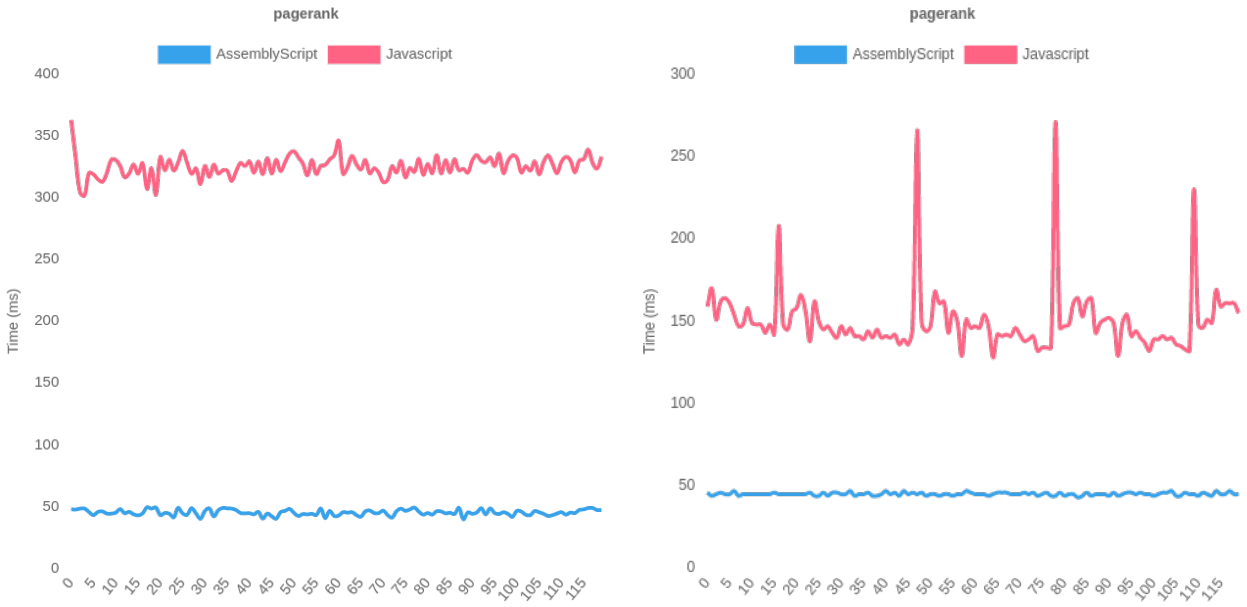
the performance on large arrays. This answers RQ2 as there are times when Assemblyscript may not be the best option. Ideally, the reference to *i'th* row should be maintained for every iteration of *j* and only released at a later time. A quick solution for this would be to maintain a row reference outside the inner loop. But in the case of a matrix transpose, column major traversal needs to be performed, so caching the *j'th* row outside the inner loop would not be possible. A solution that would work would be to represent the matrix using a 1-D array. This would prevent the program from being slowed down due to reference counting. It wasn't done for the purposes of this paper in order demonstrate that idiomatic Typescript may not always be faster in Assemblyscript. Once Garbage Collection is introduced in Webassembly and Assembblyscript, this specific issue would quite likely be solved.



(a) Chrome (b) Firefox

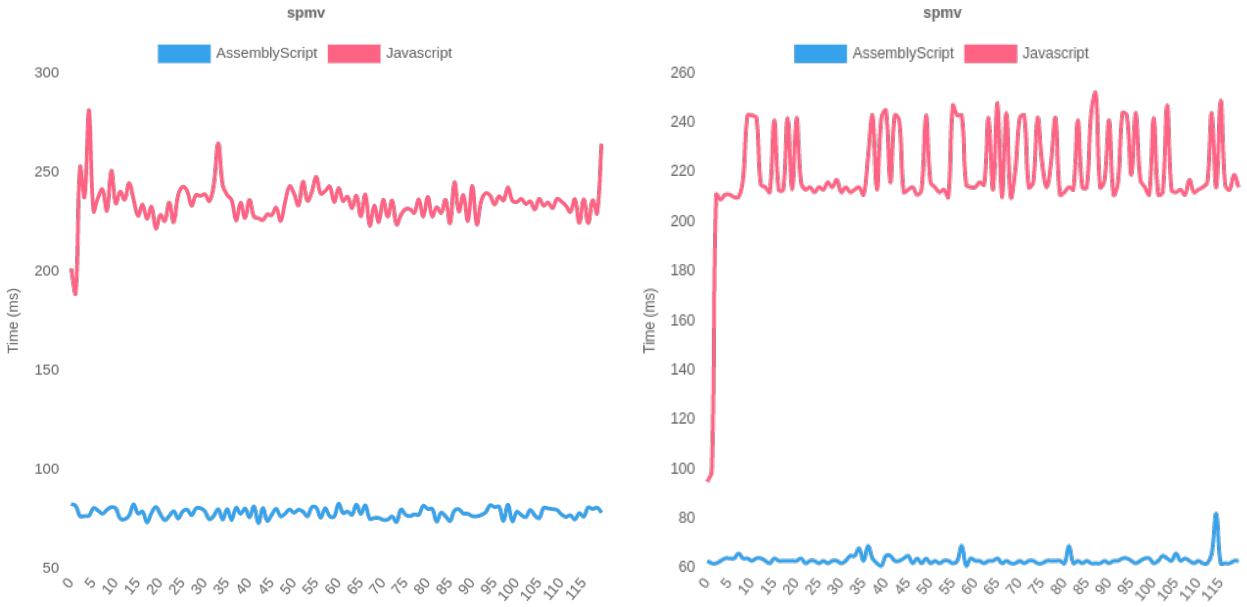
Figure 4.4: Lower Upper Decomposition

For most results across Javascript and Assemblyscript, the first iteration is quite slow. This is more noticeable in Javascript. After the initial iteration the performance becomes a lot better with the exception of occasional spikes. The speedup is a result of browser optimizations. All browser engines consist of an interpreter and an optimizing compiler [37]. After the Javascript is parsed and



(a) Chrome (b) Firefox

Figure 4.5: Page Rank



(a) Chrome (b) Firefox

Figure 4.6: Sparse Matrix Vector Multiplication

an Abstract syntax tree is generated, the interpreter comes into play. An interpreter generates byte code and executes the Javascript. When a function is executed a few times, it becomes 'hot' and the optimizing compiler kicks in and attempts to optimize the code based on various assumptions. This is why we see an improvement in the performance over multiple iterations. When the assumptions are incorrect, the flow is passed back to the interpreter and this is called a 'de-opt' or a deoptimization. The fluctuation in the results can be attributed to the optimizations and deoptimizations that occur in the engine. In addition, factors like garbage collection and other processes running in the browser can possibly affect the consistency of the results. Despite having a similar 2 stage pipeline which includes a baseline compiler and an optimizing compiler [38], we notice that Webassembly is fairly consistent in comparison to Javascript. This is because of features like strict types and lower level architecture allows the compiler to make strong guarantees and avoids getting deoptimized. One anomaly in the results was that for *spmv* in Javascript, on the first iteration it performs well but then the performance takes a significant hit. The reason for this behavior is not entirely clear. One possible explanation could be that initially the browser doesn't have other tasks and processes running in the background, and the speedup from the optimizing compiler is negligible in comparison to the overhead of those background processes on subsequent runs.

CHAPTER 5. Conclusion and Future Work

Even though browser engines are relatively fast today, they lag behind in comparison to native code. The advent of Webassembly made it possible to execute typed languages such as Rust, C++, and Assemblyscript on the web to achieve near native speeds. Since Assemblyscript is fairly new, this paper presented a set of benchmarks to test the performance of Assemblyscript in comparison to Javascript. This study showed that Assemblyscript does in fact provide performance benefits in most cases. It also showed that idiomatic Typescript code may not always be fast in Assemblyscript. But these are mostly limitations due to lack of maturity. As Webassembly starts to support features like SIMD, multi-threading, and Garbage Collection, and tooling infrastructure like Binayen improves, Assemblyscript will only become more sophisticated and easy to use. Aside from performance benefits, this paper showed that Assemblyscript (or Webassembly) is more predictable in its performance since it doesn't fall off the fast path frequently and get deoptimized by the browser engine like Javascript.

Possible future work includes extending the benchmark suite to include all of the 12 dwarf problems and popular graphics benchmarking tests translated to Assemblyscript. In addition, other languages like Rust and C++ can be tested to demonstrate their performance in comparison to Assemblyscript. Even though this paper focused primarily on performance, the benchmark suite can be extended to have tests for file size to help developers make educated decisions.

BIBLIOGRAPHY

[1] C. Severance, "Javascript: Designing a language in 10 days," *Computer*, vol. 45, no. 02, pp. 7–8, Feb. 2012.

[2] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the web up to speed with webassembly," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2017. New York, NY, USA: ACM, 2017, pp. 185–200. [Online]. Available: http://doi.acm.org/10.1145/3062341.3062363

[3] "Assemblyscript," Sep. 2017. [Online]. Available: https://github.com/AssemblyScript

[4] "How v8 measures real-world performance," Dec. 2016. [Online]. Available: https://v8.dev/blog/real-world-performance

[5] "Launching ignition and turbofan," May 2017. [Online]. Available: https://v8project.blogspot.ca/2017/05/launching-ignition-and-turbofan.html

[6] L. Clark, "A crash course in just-in-time (jit) compilers." [Online]. Available: https://hacks.mozilla.org/2017/02/a-crash-course-in-just-in-time-jit-compilers/

[7] A. Gal, B. Eich, M. Shaver, D. J. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz, "Trace-based just-in-time type specialization for dynamic languages," in *PLDI '09*, 2009.

[8] F. Khan, V. Foley-Bourgon, S. Kathrotia, E. Lavoie, and L. Hendren, "Using javascript and webcl for numerical computations: A comparative study of native and web technologies," in *Proceedings of the 10th ACM Symposium on Dynamic Languages*, ser. DLS '14. New York, NY, USA: ACM, 2014, pp. 91–102. [Online]. Available: http://doi.acm.org/10.1145/2661088.2661090

[9] J. Rüth, T. Zimmermann, K. Wolsing, and O. Hohlfeld, "Digging into browser-based crypto mining," *Proceedings of the Internet Measurement Conference 2018 on - IMC '18*, 2018. [Online]. Available: http://dx.doi.org/10.1145/3278532.3278539

[10] B. Malle, N. Giuliani, P. Kieseberg, and A. Holzinger, "The need for speed of ai applications: Performance comparison of native vs. browser-based algorithm implementations," *ArXiv*, vol. abs/1802.03707, 2018.

[11] A. Evans, M. Romeo, A. Bahrehmand, J. Agenjo, and J. Blat, "3d graphics on the web: A survey," *Computers & Graphics*, vol. 41, pp. 43–61, 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0097849314000260

[12] A. L. Ahire, A. Evans, and J. Blat, "Animation on the web: a survey," in *Web3D*, 2015.

[13] H. Levkowitz and C. Kelleher, "Cloud and mobile web-based graphics and visualization," in *2012 25th SIBGRAPI Conference on Graphics, Patterns and Images Tutorials*, Aug. 2012, pp. 21–35.

[14] J. Echterhoff, "Benchmarking unity performance in webgl – unity blog," Oct. 2014. [Online]. Available: https://blogs.unity3d.com/2014/10/07/benchmarking-unity-performance-in-webgl

[15] ——, "Updated webgl benchmark results – unity blog," Dec. 2015. [Online]. Available: https://blogs.unity3d.com/2015/12/15/updated-webgl-benchmark-results

[16] T. Horak, U. Kister, and R. Dachselt, "Comparing rendering performance of common web technologies for large graphs," in *Poster Program of the 2018 IEEE VIS Conference*, ser. VIS '18, 2018.

[17] R. C. Hoetzlein, "Graphics performance in rich internet applications," *IEEE Computer Graphics and Applications*, vol. 32, no. 5, pp. 98–104, Sep. 2012.

[18] M. Selakovic and M. Pradel, "Performance issues and optimizations in javascript: An empirical study," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, May 2016, pp. 61–72.

[19] P. Jensen, "Simd in javascript via c + + and emscripten," in *Workshop on Programming Models for SIMD/Vector Processing*, 2015.

[20] A. Z. David Herman, Wagner Luke, "asm.js," Aug. 2014. [Online]. Available: http://asmjs.org

[21] A. Zakai, "Emscripten: An llvm-to-javascript compiler," in *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, ser. OOPSLA '11. New York, NY, USA: ACM, 2011, pp. 301–312. [Online]. Available: http://doi.acm.org/10.1145/2048147.2048224

[22] C. Lattner and V. Adve, "Llvm: a compilation framework for lifelong program analysis transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, Mar. 2004, pp. 75–86.

[23] Webassembly, "Binaryen," Mar. 2016. [Online]. Available: https://github.com/WebAssembly/binaryen

[24] M. Reiser and L. Bläser, "Accelerate javascript applications by cross-compiling to webassembly," in *Proceedings of the 9th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*, ser. VMIL 2017. New York, NY, USA: ACM, 2017, pp. 10–17. [Online]. Available: http://doi.acm.org/10.1145/3141871.3141873

[25] "Webassembly is here! – unity blog." [Online]. Available: https://blogs.unity3d.com/2018/08/15/webassembly-is-here/

[26] "Webassembly load times and performance – unity blog." [Online]. Available: https://blogs.unity3d.com/2018/09/17/webassembly-load-times-and-performance/

[27] A. Zakai and Firefox, "Why webassembly is faster than asm.js." [Online]. Available: https://hacks.mozilla.org/2017/03/why-webassembly-is-faster-than-asm-js/

[28] "Retiring octane," Apr. 2017. [Online]. Available: https://v8.dev/blog/retiring-octane

[29] Mozilla, "Kraken javascript benchmark," Nov. 2010. [Online]. Available: http://krakenbenchmark.mozilla.org

[30] "Browserbench," 2014. [Online]. Available: http://browserbench.org

[31] G. Richards, A. Gal, B. Eich, and J. Vitek, "Automated construction of javascript benchmarks," in *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '11. New York, NY, USA: ACM, 2011, pp. 677–694. [Online]. Available: http://doi.acm.org/10.1145/2048066.2048119

[32] F. Khan, V. Foley-Bourgon, S. Kathrotia, and E. Lavoie, "Ostrich benchmark suite." [Online]. Available: https://github.com/Sable/Ostrich

[33] D. Herrera, H. Chen, E. Lavoie, and L. Hendren, "Numerical computing on the web: Benchmarking for the future," in *Proceedings of the 14th ACM SIGPLAN International Symposium on Dynamic Languages*, ser. DLS 2018. New York, NY, USA: ACM, 2018, pp. 88–100. [Online]. Available: http://doi.acm.org/10.1145/3276945.3276968

[34] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams *et al.*, "The landscape of parallel computing research: A view from berkeley," University of California, Berkeley, Tech. Rep., 2006.

[35] L. Clark, "Calls between javascript and webassembly are finally fast." [Online]. Available: https://hacks.mozilla.org/2018/10/calls-between-javascript-and-webassembly-are-finally-fast

[36] D. F. Bacon, C. R. Attanasio, S. E. Smith, and H. B. Lee, "A pure reference counting garbage collector," 2007.

[37] "Javascript engine fundamentals: Shapes and inline caches," Jun. 2018. [Online]. Available: https://mathiasbynens.be/notes/shapes-ics

[38] "Liftoff: a new baseline compiler for webassembly in v8," Aug. 2018. [Online]. Available: https://v8.dev/blog/liftoff