Review Article *Open Access*

Journal of Pioneering Artificial Intelligence Research

ISSN: 3069-0846

DOI: doi.org/10.63721/25JPAIR0111

Performance Comparison of Web Assembly and JavaScript

Jakub Ciszewski[1*] and Kaja Myk[2]

Electrical Engineering WUT Applied Computer Science Warsaw, Poland

Citation: Jakub Ciszewski (2025) Performance Comparison of Web Assembly and JavaScript J of Pion Artf Research 1(2),1-7. WMJ/JPAIR-111

Abstract

JavaScript remains the dominant language for client-side scripting, while Web Assembly offers near-native execution speeds, making it a compelling choice for computationally intensive tasks. This study provides a comprehensive analysis of the performance differences between Web Assembly and JavaScript across various computing environments, including different browsers (Firefox, Chrome, Edge) and platforms (desktop and mobile).

To evaluate computational efficiency, we conducted a series of benchmark tests, including integer operations (Sieve of Er- Antisthenes, sorting algorithms), floating-point calculations (numerical integration, Monte Carlo method) and recursive computations (Fibonacci sequence, matrix multiplication). Additionally, we investigated the impact of Web Assembly on ma- chine learning workloads by utilizing minimalist implementations such as Tiny DNN for digit classification on the MNIST dataset. Our findings indicate that Web Assembly consistently outperforms JavaScript in CPU-bound tasks, particularly in integer operations and recursive computations. However, JavaScript's just-in-time (JIT) compilation allows it to remain competitive in some floating-point calculations.

One focus of our study was the application of Web Assembly in browser-based machine learning. We examined the performance of lightweight neural network implementations, emphasizing Web Assembly's ability to accelerate tensor computations directly in the browser. This capability is crucial for deploying AI models on the client side, reducing reliance on cloud-based services, improving privacy, and minimizing latency. Our analysis also explores Web Assembly's potential for real-time applications such as image classification, object detection, and natural language processing.

Beyond raw performance metrics, this study assesses the broader implications of Web Assembly's adoption in web development. One of its key advantages is its ability to support multiple programming languages, including Rust, C, and C++, allowing developers to leverage high-performance libraries within the browser environment. Additionally, Web Assembly's sandboxing mechanisms enhance security by isolating execution, reducing potential attack vectors compared to traditional JavaScript-based applications.

Despite its advantages, Web Assembly has limitations. Execution performance varies across browsers, and

its integration with JavaScript-based applications presents challenges due to data serialization overhead. Moreover, Web Assembly lacks direct access to the DOM, necessitating JavaScript as an intermediary for UI interactions.

As Web Assembly continues to evolve, its role in performance- critical web applications is expected to expand, particularly in fields such as cryptography, data processing, and real-time machine learning.

***Corresponding author:** Jakub Ciszewski, Electrical Engineering WUT Applied Computer Science Warsaw, Poland.

Submitted: 26.07.2025 **Accepted:** 29.07.2025 **Published:** 10.09.2025

Keywords: Web Assembly, JavaScript, Performance Comparison, Machine Learning, Convolutional Neural Networks

Introduction

Web Assembly (WASM) is a binary format that enables high-performance execution in web browsers, offering near- native speed for computationally intensive tasks. Unlike JavaScript, which is interpreted, Web Assembly is compiled from languages like C, C++, and Rust, providing faster execution for complex applications. While JavaScript remains the dominant client-side scripting language, Web Assembly has emerged as a strong alternative for performance-critical tasks, particularly in areas such as numerical simulations and machine learning [1-7].

The topic of performance comparison between JavaScript and Web Assembly has gained significant academic interest, with numerous studies exploring how these two technologies handle computationally intensive tasks in web applications. As JavaScript remains the dominant language for client-side scripting, researchers have sought to understand the performance benefits of Web Assembly, particularly for tasks that require near-native execution speeds [8].

A recent study investigates the potential for enhancing JavaScript performance in Internet of Things (IoT) environments through the use of Web Assembly. The authors con- ducted a series of experiments on a Raspberry Pi platform, comparing execution time, memory consumption, and energy usage across implementations in JavaScript, Web Assembly, and C. The results demonstrate that Web Assembly can significantly improve execution speed and reduce energy consumption in IoT applications, although this may come at the cost of a slight increase in memory usage in certain scenarios [10]. These findings suggest that Web Assembly offers a promising avenue for optimizing JavaScript-based workloads in resource-constrained IoT devices.

In a comprehensive evaluation of execution efficiency and energy consumption, the study investigates the comparative performance of Web Assembly (Was) and JavaScript (JS) within modern web browser environments. Employing a combination of microbenchmarks and real-world applications—including a game console emulator and a PDF viewer—the authors systematically analysed performance across major browsers such as Google Chrome, Microsoft Edge, and Mozilla Firefox. The findings reveal that Was delivers superior performance and reduced energy usage in practical application scenarios, although results from micro bench- marks exhibit greater variability [9]. This research underscores the viability of Wasim as a high-efficiency alternative to JS in web-based applications and establishes a methodological foundation for further empirical studies in this domain.

Weifang Wang investigates the performance of web applications enhanced with web Assembly in comparison to their JavaScript counterparts, focusing on execution time and memory usage across a variety of programs. The study reveals that while Chrome's Just-In-Time (JIT) optimization significantly accelerates JavaScript execution, it has little to no impact on Web

Assembly performance. Furthermore, the authors report that Web Assembly consumes substantially more memory than JavaScript, primarily due to differences in memory management models [11]. The authors express hope that these findings will contribute to the ongoing refinement of Web Assembly runtimes. Overall, this work advances the understanding of the practical performance benefits and trade-offs of Web Assembly relative to traditional JavaScript in the context of modern web applications.

This study aims to compare the performance of Web Assembly and JavaScript in executing various algorithms across different browsers (Firefox, Chrome, Edge, and Safari) and platforms (Linux and Windows). We evaluate execution times for integer, floating-point, and recursive computations, providing insights into the strengths and weaknesses of each approach for web-based applications. Furthermore, the study explores the practical application of Web Assembly in machine learning tasks, particularly focusing on its performance with the MNIST dataset.

Comparison between Web-Assembly and JavaScript for Mathematical Operations

Performance Testing

We conducted a series of performance tests on both Windows and Linux, evaluating the execution times of various numerical computations across different web browsers, including Safari, Chrome, Firefox, and Edge. The primary objective of these tests was to compare the efficiency of JavaScript and Web-Assembly implementations in handling computationally intensive tasks, measuring execution times in seconds. This approach allowed us to analyse the impact of both interpreted (JavaScript) and compiled (C-to-WASM) implementations on the execution speed of the algorithms.

The selected algorithms include:

- The Sieve of Eratosthenes, used for generating prime numbers,
- The Monte Carlo method, applied for estimating the value of Pi through random sampling,
- The Fibonacci sequence, calculated using recursive and iterative approaches,
- Numerical integration, a method for approximating the integral of a function.

It is important to note that Safari was not tested on Linux because the browser is not officially available on this operating system. This limitation prevented direct comparisons for Safari across both platforms.

To ensure the accuracy and fairness of our comparisons, all tests were performed using the same number of iterations for each algorithm. This approach guarantees that differences in execution time reflect the efficiency of the browsers and the underlying system optimizations rather than variations in workload.

The results of these tests are presented in the tables below, providing a clear overview of how different browsers and operating systems handle numerical computations.

Table 1: JavaScript Test Results on Linux in Seconds

Algorithm	Firefox	Chrome	Safari	Edge
The Monte Carlo method	6.289	11.409	X	13.222
The Sieve of Eratosthenes	13.011	13.333	X	15.101
The Fibonacci sequence	3.133	6.212	X	6.200
Numerical integration	5.092	8.743	X	9.111

Table 2: Web assembly Test Results on Linux in Seconds

Algorithm	Firefox	Chrome	Safari	Edge
The Monte Carlo method	6.209	10.902	X	11.238
The Sieve of Eratosthenes	6.001	12.021	X	12.029
The Fibonacci sequence	2.103	5.421	X	6.101
Numerical integration	2.121	4.781	X	5.140

Table 3: Web assembly Test Results on Windows in Seconds

Algorithm	Firefox	Chrome	Safari	Edge
The Monte Carlo method	8.202	6.401	12.202	6.303
The Sieve of Eratosthenes	15.101	14.109	13.208	15.093
The Fibonacci sequence	6.101	4.201	6.101	4.082
Numerical integration	8.009	6.987	8.302	7.503

Table 4: Webassembly Test Results on Windows in Seconds

Algorithm	Firefox	Chrome	Safari	Edge
The Monte Carlo method	7.989	13.555	12.161	6.331
The Sieve of Eratosthenes	13.101	14.004	12.984	13.101
The Fibonacci sequence	5.991	3.099	6.055	4.001
Numerical integration	7.778	6.559	8.333	7.123

Result Comparison

Test Results on Linux: On Linux, the best results were achieved with Firefox, followed by Chrome and Edge. Web-Assembly (WASM) was faster than JavaScript across the board.

Firefox generally performed better due to its highly optimized JavaScript engine and the efficient handling of Web-Assembly. It has long been known for its focus on performance improvements, particularly for web standards like Web-Assembly, and is often faster in executing complex computations. Chrome also showed good performance, benefiting from the V8 engine's strong optimization capabilities. However, Edge, while competitive, tended to lag behind, possibly due to less aggressive optimizations for Web-Assembly or differences in its JavaScript engine performance. Additionally, the varying levels of browser support for Web-Assembly optimizations and how each browser manages resources may have contributed to the observed differences in performance.

Test Results on Windows

On windows, we also achieved better results with Web-Assembly. The fastest browsers were Edge, Chrome, Safari, and Firefox, in that order.

Edge performed the best, likely due to its strong integration with the Windows operating system and optimizations made for performance, especially in handling Web Assembly. Chrome followed closely behind, benefiting from the V8 engine's advanced optimizations for both JavaScript and Web Assembly.

Safari, while generally optimized for performance on Apple devices, showed solid results but was slower compared to Edge and Chrome, possibly due to the optimizations being more focused on macOS. Firefox, although performing well, lagged slightly behind in this case, which could be attributed to differences in how its Web Assembly runtime and JavaScript engine were optimized for Windows. These variations can result from the different priorities and optimizations made by each browser's development team for specific platforms.

Comparison of JavaScript and Web assembly Performance in Machine Learning Tasks

Methodology

The aim of this chapter is to evaluate the practical application of Web Assembly (WASM) in comparison to JavaScript (JS) for machine learning (ML) tasks within web browsers. The study focuses on assessing how WASM performs in real- world image processing tasks, such as classifying the MNIST dataset. The Convents library was used to repeatedly load and process 8,000 images 100 times, simulating a typical ML workflow.

Data was collected using developer tools available in mod- ern web browsers, allowing for precise measurements of execution time and memory consumption across different platforms and browsers. The final results were calculated as the average value from the conducted tests, providing reliable insights into the performance of WASM compared to JS.

The primary goal of this study was to investigate the potential of Web Assembly as a solution for computationally intensive tasks in browser-based ML applications. Particular attention was given to how WASM can offer superior performance and reduced memory usage, especially in mobile environments and other resource-constrained platforms.

Execution Time Comparison

The test results clearly demonstrate the significant advantage of Web Assembly over JavaScript in terms of execution time. The speed-up achieved with Web Assembly varied depending on the platform and browser, but in every case, WASM significantly outperformed JavaScript. The highest speed-up was observed on mobile devices, particularly on the iPhone using Safari, where the performance improvement reached 30.03x. In contrast, the smallest speed-up was recorded on Windows, specifically using the Edge browser, with a gain of 17.81x.

Table 5: Execution Time and Speed-Up Comparison For JavaScript And Web assembly Across Different Platforms and Browsers.

Platform	Browser	JS (s)	WASM (s)	Speed-up
Android	Chrome	120.887	4.531	26.68
Android	Samsung	128.619	4.819	26.69
iPhone	Safari	69.700	2.321	30.03
iPhone	Chrome	69.004	2.332	29.59
Windows	Chrome	75.348	4.140	18.20
Windows	Edge	76.049	4.270	17.81
Windows	Firefox	124.741	4.603	27.10
Windows	Opera	76.061	4.170	18.24
MacBook	Safari	77.258	4.259	18.14

Memory Consumption Analysis

In addition to execution time, the average memory consumption during image processing was measured. JavaScript consumed an average of 33.2 MB, while Web Assembly used only 19.6 MB. These results suggest that Web-Assembly not only accelerates data processing but also provides more efficient memory management. This efficiency is particularly significant for mobile devices with limited resources, where conserving memory can significantly enhance both performance and user experience.

Table 6: Memory Usage Comparison Between JavaScript and Web assembly During Image Processing.

Metric	JS(MB)	WASM(MB)
Memory usage	33.2	19.6

Comparison with Native Execution

To provide further context, the execution time of the same ML task was measured on a native implementation running on a laptop equipped with an Intel Core i7 processor. The native execution time was 1.5 seconds, significantly outperforming both JS and WASM. However, while WASM remains slower than native execution, its performance is much closer to native than JavaScript, making it a viable alternative for high- performance web-based ML applications.

Results

The conducted study clearly indicates the superiority of Web Assembly over JavaScript in terms of performance and memory efficiency for ML applications in browsers. WASM enables several times faster execution of ML operations, which is crucial for applications requiring intensive computations. This advantage is particularly evident on mobile devices, where memory efficiency and execution speed are key factors for a positive user experience. The data was collected using built-in developer tools in browsers, ensuring precise measurements. Although native execution remains the fastest option, Web Assembly offers a compelling balance between performance and accessibility, making it the preferred technology for client-side ML applications.

Limitations of Web assembly in Dom Manipulation
Although Web Assembly provides significant performance improvements and allows code written in various languages to run on the web, it currently does not support direct manipulation of the Document Object Model (DOM). DOM APIs are only accessible through JavaScript, meaning that any Web Assembly-based application must rely on a" bridge" to JavaScript in order to interact with the page structure.

In practice, this means that if a Web Assembly module needs to modify the layout, add HTML elements, or respond to user events, it must call JavaScript functions that perform the necessary DOM operations. This adds overhead and complexity, as developers need to maintain an integration layer between WASM and JS.

Conclusions

The performance comparison conducted across both Linux and Windows platforms reveals significant differences between JavaScript and Web-Assembly in handling computationally intensive tasks. JavaScript, being an interpreted language, introduces overhead that affects its execution speed, especially for algorithms such as the Monte Carlo method, Sieve of Eratosthenes, Fibonacci sequence, and Numerical integration. On the other hand, Web Assembly offers a more efficient execution environment due to its closer alignment with machine code, leading to faster performance for complex computations. Additionally, when C code is compiled to Web-Assembly, the results show even greater improvements in performance.

While browser performance varies, with Firefox generally performing better than Chrome and Edge, the overall findings demonstrate that Web-Assembly (particularly when compiled from C) provides a more efficient solution for computationally demanding tasks across both Windows and Linux platforms. This comparison highlights the performance trade-offs and ad- vantages of each approach for executing intensive algorithms. In the context of machine learning (ML), particularly for tasks like the MNIST dataset classification, Web-Assembly offers significant practical benefits. Training models and per- forming inference on such tasks in the browser can be computationally expensive, especially when handling large datasets like MNIST. With the integration of Web-Assembly, ML operations benefit from faster execution times compared to JavaScript, making it a compelling choice for client-side ML applications. For example, in the case of image classification tasks, Web-Assembly allows for faster data processing, reduced latency, and improved overall user experience, especially on devices with limited computational power. This makes Web-Assembly a strong candidate for real-time ML applications running in browsers, where performance and efficiency are critical.

However, despite these performance advantages, Web-Assembly currently lacks the ability to directly manipulate the Document Object Model (DOM), which remains accessible only through JavaScript. As a result, applications that require interaction with the browser's interface must still rely on JavaScript as a bridge. This introduces additional complexity and may offset some performance gains in scenarios involving frequent or dynamic UI updates. Therefore, while Web-Assembly is a powerful tool for computational logic, it is best used in conjunction with JavaScript for full-featured web applications.

References

1. D Herrera, H Chen, E Lavoie and L Hendren (2018) Web-Assembly and JavaScript Challenge: Numerical Program Performance Using Mod- ern Browser Technologies and Devices https://www.sable.mcgill.ca/publications/techreports/2018-2/techrep.pdf.
2. J De Macedo, R Abreu, R Pereira and J Saraiva (2021) On the Runtime and Energy Performance of Web-Assembly: Is Web-Assembly Superior to JavaScript Yet? http://ieeexplore.ieee.org/document/9680302.
3. M Reiser, L Blaser (2017) Accelerate JavaScript Applications by Cross- Compiling to Web-Assembly, Proceedings of the ACM 10-17.
4. Y Yan, T Tu, L Zhao, Y Zhou and W Wang (2021) Understanding the Performance of Web-Assembly Applications, Proceedings of the ACM 533-549.
5. B R Mohan, Tushar (2022) Comparative Analysis of JavaScript and Web-Assembly in the Browser Environment https://ieeexplore.ieee.org/document/9929829/authors#authors.
6. J W Sunarto, A Quincy, F S Maheswari, Q D A Hafizh, M G Tjandrasubrata, et al. (2023) A Systematic Review of Web-Assembly vs. JavaScript Performance Comparison https://ieeexplore.ieee.org/document/10277917.
7. A Haas, A Rossberg, D Schuff, B Titzer, M Holman, et al. (2017) Bringing the Web Up to Speed with Web-Assembly,Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation 185-200.
8. A Jangda, B Powers, E D Berger and A Guha (2019) Not So Fast: Analysing the Performance of Web-Assembly vs. Native Code, Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC 19) 107-120.
9. R Lehmann, A Reinefeld (2020) Performance Comparison of Web-Assembly and JavaScript for Numerical Computations in the Browser, Proceedings of the 9th International Conference on Performance Evaluation and Benchmarking for Big Data 1-6.
10. F L Oliveira, J C B Mattos (2020) Analysis of Web Assembly as a Strategy to Improve JavaScript Performance on IoT Environments https://sol.sbc.org.br/index.php/sbesc_estendido/article/view/13102.
11. W Wang (2021) Empowering Web Applications with Web Assembly: Are We There Yet? https://ieeexplore.ieee.org/document/9678831.