American Academic Scientific Research Journal for Engineering, Technology, and Sciences
ISSN (Print) 2313-4410, ISSN (Online) 2313-4402
https://asrjetsjournal.org/index.php/American_Scientific_Journal/index

Integration of WebAssembly in Performance-critical Web Applications

Poltavskyi Dmytro*

Team lead at Upland.me, Poland, Warsaw
Email: poltavskyi.dmytro@gmail.com

Abstract

This article explores the integration of WebAssembly into high-performance web applications as a response to the increasing demands for computational power, scalability, and security in the rapidly evolving landscape of web technologies and the Internet of Things (IoT). The study substantiates the relevance of transitioning from traditional JavaScript to WebAssembly, which allows code written in C/C++ or Rust to be compiled into a compact binary format, delivering near-native execution speed. The article analyzes the architecture of WebAssembly, its advantages, and its integration potential with other technologies, such as WebGPU for accelerated parallel computations. Special attention is given to the current limitations of WebAssembly (e.g., the lack of native garbage collection, debugging difficulties, and challenges in cross-language integration) as well as its promising development directions, including the standardization of WASI and enhancements through multithreading and SIMD support. In comparative experiments on 1024 × 1024 matrix multiplication, the SIMD-enabled WebAssembly module with block-optimized memory access outperformed the optimized JavaScript implementation by 1.64 × and delivered a 4 × improvement over the unvectorized Wasm build, while offloading computations to WebGPU achieved an ~50-fold reduction in execution time for both JavaScript+WebGPU and Wasm+WebGPU configurations. These results substantiate that the integration of WebAssembly and WebGPU brings near-native and GPU-accelerated performance to browser-based applications, laying a quantitatively validated foundation for high-load web and IoT systems.The paper demonstrates a way to accelerate client data processing using a combination of Web Assembly and Web GPU. The results of a comparative experiment are presented. This article will be of interest to professionals in web development and systems architecture who aim to optimize computational workflows and maximize the performance of modern web applications via WebAssembly. Additionally, the material provides valuable insights for researchers engaged in the analysis and development of advanced methodological approaches to optimizing high-load information systems.

Received: 3/30/2025
Accepted: 5/12/2025
Published: 5/23/2025

---

* Corresponding author.

Keywords: WebAssembly; high-performance web applications; compilation; WASI; WebGPU; cross-language integration; IoT; performance optimization.

1. Introduction

Web applications increasingly face the need to handle computationally intensive tasks in real time—a challenge that traditional JavaScript-based approaches often fail to meet efficiently, resulting in latency and elevated power consumption. In this context, WebAssembly (hereafter Wasm) emerges as a relevant solution. It allows code written in C/C++ or Rust to be compiled into a compact binary format that delivers near-native execution speed and ensures high security through sandbox isolation [1]. Furthermore, the integration of WebAssembly with complementary technologies such as WebGPU opens up new opportunities for implementing parallel computations and accelerating AI-related tasks, highlighting the importance of this subject.

The goal of this study is to analyze the specific features of WebAssembly integration into high-performance web applications aimed at accelerating computationally intensive tasks and optimizing system resource usage.

The novelty of this study lies in the introduction of a new method for running neural network inference directly in the browser using WebGPU in conjunction with WebAssembly, without requiring any server-side processing.

The working hypothesis is that employing WebAssembly in performance-critical web applications can reduce task execution time and resource consumption when compared to conventional JavaScript-only solutions. Moreover, the synergy between WebAssembly and technologies like WebGPU can deliver additional computational acceleration, which is particularly relevant for applications handling large datasets or executing parallel operations.

2. Materials and methods

In recent years, the integration of WebAssembly into high-performance web applications has become an active research topic, reflected in a wide array of publications sharing both thematic and methodological commonalities. The literature review includes 10 publications from the years 2021 to 2025, selected based on the keywords *WebAssembly*, *WASI*, and *WebGPU* in Google Scholar. A number of studies focused on improving web application efficiency emphasize optimizing computational processes and integrating advanced graphics technologies. For instance, Odume B. W., Okodugha P. E., and Madu I. [1] examine the synergistic potential of combining WebAssembly and WebGPU for deploying AI models, showing that this combination can significantly reduce response times and boost overall application performance. Similarly, Kyriakou K. I. D. and Tselikas N. D. [4] explore the use of Rust alongside WebAssembly to complement JavaScript in high-performance Node.js and web applications, thereby improving scalability and reducing latency. Wang W. [8], in a comprehensive review, examines trends in WebAssembly adoption within web applications, identifies current limitations, and proposes future development directions for the technology.

Another segment of research focuses on the application of WebAssembly in Internet of Things (IoT) devices and embedded systems. A review by Ray P. P. [2] spans a broad range of topics—from toolchains to challenges

and prospects—highlighting WebAssembly's potential for enabling energy efficiency and reliability in computational processes. Concurrently, Kim J. and his colleagues [3] present a hardware-oriented approach, designing a WebAssembly-based accelerator for embedded systems, which enhances data processing times and reduces CPU load.

Security is another prominent area of concern in the literature on WebAssembly. The review by Perrone G. and Romano S. P. [5] provides a structured overview of existing threats and mitigation strategies, identifying both the strengths of Wasm in isolating code and the vulnerabilities tied to cross-platform integration. Further comparative analysis by Dejaeghere J. and his colleagues [9] contrasts the security of Wasm with that of eBPF, offering insights into trade-offs between performance and protection.

Considerable attention has also been paid to the potential of WebAssembly in edge computing contexts. Hoque M. N. and Harras K. A. [10] examine the technology's suitability for distributed environments, emphasizing the benefits of reduced latency through local processing, while also noting challenges such as resource management and data migration. Complementary studies of existing Wasm runtimes, such as the one by Zhang Y. and his colleagues [7], offer a foundational understanding of the current infrastructure for executing WebAssembly code, which is crucial for the development of future applications. In addition, Ramirez C. E., Sermet Y., and Demir I. [6] describe how WebAssembly can be used to create open-source libraries for hydrology and environmental sciences, demonstrating the technology's interdisciplinary potential.

The analysis of existing research provided the essential groundwork for this article. It demonstrated that WebAssembly's high performance does not arise automatically from its use but instead depends on deep low-level optimizations—such as SIMD instruction sets and manual memory management—especially when benchmarked against state-of-the-art JavaScript engines. Employing WebGPU for parallel computations has shown considerable potential for acceleration, often narrowing the performance gap between CPU-driven workloads in JavaScript and those in WebAssembly. Additionally, the need for rigorous Wasm optimization on the CPU highlights challenges for its deployment in resource-constrained contexts—such as IoT devices—absent dedicated hardware or runtime support.

Despite this extensive research foundation, notable contradictions remain. On the one hand, much attention has focused on boosting performance and optimizing computation. On the other, critical concerns around security and resource governance have not been explored in sufficient depth. Cross-platform compatibility issues and the nuances of tuning WebAssembly on limited-capacity edge devices are under-represented in the literature. Addressing these gaps will require comprehensive empirical studies to establish unified standards and methodologies that ensure both peak efficiency and robust protection in contemporary web applications.

3. Results

WebAssembly (Wasm) is a technology designed to overcome the limitations of traditional JavaScript in executing computationally intensive tasks within web applications. It is a binary format that enables code written in low-level languages such as C, C++, or Rust to be compiled into compact executable modules capable

of running at near-native speed [1]. The development of the Wasm standard is a response to the growing demands for performance, security, and portability in modern web applications, and its backing by industry leaders such as Google, Mozilla, Microsoft, and Apple underscores its strategic importance [2, 3].

Originally conceived as a solution to JavaScript's interpretability and limited performance in data-heavy environments, WebAssembly features a stack-based architecture for storing intermediate computational values, which significantly optimizes code execution. Wasm supports both ahead-of-time (AOT) and just-in-time (JIT) compilation, allowing the execution environment to adapt dynamically to application conditions [5]. Its strict sandboxing model ensures that each module runs in an isolated environment, reducing the risk of unauthorized access to system resources and enhancing the overall security of web applications [2].

The main advantages of WebAssembly include:

- High Performance: Through precompilation into binary format, Wasm modules execute at near-native speed, which is critical for performance-sensitive tasks such as image processing, simulations, and machine learning.
- Security: The isolated sandbox environment enforces strict resource access controls, mitigating vulnerabilities typically associated with interpreted languages.
- Portability and Cross-Platform Compatibility: A single binary module can run on any device that supports a WebAssembly runtime—from servers to resource-constrained environments, including IoT devices [2].
- Integration with JavaScript: The WebAssembly JavaScript API allows seamless integration of modules into existing web applications, facilitating communication between high-performance code and application logic implemented in JavaScript [3].

For a deeper understanding of Wasm’s characteristics, Table 1 summarizes its key features:

Table 1: WebAssembly Features [1–3]

Feature	**Description**	**Advantages**
Binary Format	Compact executable code that loads and decodes quickly	Fast load times and reduced memory usage
Sandbox Environment	Isolated execution context preventing unauthorized access to system resources	Enhanced security through access restriction
Compilation Model	Supports both AOT and JIT compilation for optimized execution	Adaptive performance and speed
JavaScript Integration	API suite for interaction between Wasm modules and JavaScript code	Easy integration and functional extensibility

In summary, WebAssembly is a foundational technology that significantly enhances the efficiency and security of web applications through its unique architecture, optimized binary format, and flexible integration with modern technologies such as WebGPU for parallel computation.

Modern web applications increasingly demand high performance, scalability, and security. In this context, integrating WebAssembly (Wasm) has emerged as a critical solution, allowing developers to offload compute-intensive operations from JavaScript to an execution environment with near-native performance [1]. Core integration strategies include selecting appropriate programming languages for performance-critical code sections, optimizing the compilation process, organizing interoperability between Wasm modules and JavaScript, and leveraging complementary technologies like WebGPU to accelerate parallel computing tasks in AI-driven applications. The modern WebAssembly build and debugging cycle relies on a toolchain centered around LLVM-based compilers and auxiliary utilities for processing binary modules. The most commonly used compilers include Clang, Rust (via cargo and wasm-pack), and the AssemblyScript compiler. For optimization and transformation, tools like Binaryen and WABT are employed. Linking is handled through wasm-ldor wasm-bindgen, with flags such as -g and --dwarf-version=5 preserving extended .debug_* DWARF sections, while options like --name-section or -fdebug-prefix-map ensure correct path mappings for symbolic tooling.

When compiling C/C++ sources with Emscripten, additional flags like -gsource-map and --source-map-base=… are used. The linker then annotates original source lines with a "wasm://" prefix and generates a supplementary JSON source map, which browsers refer to for debugging.

In the Rust ecosystem, the process is similar: the command cargo build --target wasm32-unknown-unknown -gproduces a module with address-to-line mappings. Then, the optimizer wasm-opt -g --dwarf --debuginfo retains essential debug information, while wasm-bindgen --keep-debug --no-demangle exports it to JavaScript. For low-level diagnostics, tools such as wasm-objdump -x (for section inspection), wasm2wat/wasm-decompile (annotated disassembly with line numbers), and llvm-symbolizer (for resolving function names from DWARF addresses) are used.

Chrome DevTools (version 89+) supports both DWARF sections and standard source maps. Upon module load, the debugger builds a reverse mapping of “address → file:line”, allowing developers to set breakpoints, inspect call stacks, view local variables, and use the Memory Inspector. Native Wasm stack traces (e.g., throw new Error()) are processed via an embedded symbolizer, converting entries like wasm-function[123]:0x45a into meaningful Rust or C++ symbols.

Visual Studio Code integrates Wasm debugging through the js-debug-adapter or the vscode-wasm extension. In launch.json, configurations with "type": "pwa-chrome" or "type": "pwa-node" include the sourceMapPathOverrides parameter for redirecting source maps locally. For remote debugging in Node-WASI, flags like --enable-diagnostics are used along with port 9229 for inspection.

Error localization is further enhanced using traditional practices: builds with AddressSanitizer/UBSan (via -fsanitize=… in Clang 15+, supported in Wasm), enabling --profiling-func-names in Emscripten for automatic

demangling, logging from Wasm into the JS environment via env.log and console.trace calls, and post-processing stack traces with the source-map-support package.

To meet the performance demands of complex applications, low-level programming languages such as C/C++ or Rust are typically preferred. These languages provide fine-grained control over memory and processing and can be compiled into the WebAssembly binary format using specialized tools like Emscripten for C/C++ or rustc for Rust [2, 8]. The resulting Wasm module is compact and runs at nearly native speed, making it ideal for scenarios that require minimal latency and efficient resource utilization.

A major advantage of WebAssembly lies in its seamless integration with JavaScript through the standardized Wasm JavaScript API. Developers can export functions from a Wasm module and invoke them within JavaScript, as well as share data via linear memory. This approach allows offloading of performance-intensive operations to Wasm while retaining UI logic and control flow in JavaScript [1, 3].

For further performance gains—especially in tasks involving data processing and AI—recent research advocates combining Wasm with WebGPU. WebGPU offers low-level access to the GPU, enabling parallel computation of tasks such as matrix multiplications or convolution operations in neural networks [3, 7]. The synergy between WebAssembly and WebGPU enables the execution of complex algorithms directly on the client side, reducing server load and minimizing latency.

For a more structured understanding of key WebAssembly integration strategies, Table 2 summarizes the approaches and their implementations:

Table 2: Strategies for Integrating WebAssembly [1, 3, 4, 7]

Strategy	**Description**	**Code Example**
Compiling Source Code to Wasm	Using C/C++ or Rust to implement performance-critical logic, followed by compilation to Wasm binary.	<emscripten.h>EMSCRIPTEN_KEEPALIVE int add(int a, int b) { return a + b; }
Wasm and JavaScript Interoperability	Loading a Wasm module and invoking its exported functions via WebAssembly.instantiate, with data exchange through shared linear memory.	fetch('module.wasm').then(response => response.arrayBuffer()).then(bytes => WebAssembly.instantiate(bytes, {})).then(results => { const instance = results.instance; console.log(instance.exports.add(5, 3)); })
Using WebGPU for Parallel Computing	Offloading heavy tasks (e.g., matrix operations in AI models) to the GPU using WebGPU API for faster data processing.	async function initWebGPU() {const adapter = await navigator.gpu.requestAdapter();const device = await adapter.requestDevice();// Create buffer and run compute shader}

These strategies help developers systematize the process of integrating WebAssembly into web applications with high performance requirements. The choice of language and an optimized compilation process deliver a foundational performance boost, while JavaScript interoperability ensures flexibility in application control. Additionally, incorporating WebGPU can significantly enhance performance in parallel computing tasks, especially relevant in image processing and AI inference.

A comprehensive approach combining these strategies enables substantial reductions in execution time for compute-heavy operations, decreased energy consumption, and improved application responsiveness—all confirmed by recent experimental research.

In conclusion, integrating WebAssembly with traditional web technologies and modern tools like WebGPU represents a promising direction for building high-performance web applications capable of meeting today's demands for speed, scalability, and security.

4. Discussions

This section presents a structured analysis of an experiment evaluating the performance of JavaScript, WebAssembly, and WebGPU technologies in the context of matrix multiplication operations (sizes ranging from 256 × 256 to 1024 × 1024) and Gaussian convolution on images (sizes from 256 × 256 to 2048 × 2048). The methodology involved the use of SIMD instructions, the implementation of memory access optimization strategies, and the monitoring of key execution metrics (mean, median, minimum, maximum, and relative speedup compared to baseline JavaScript). Based on the data obtained, the study outlines the limitations of current approaches and highlights promising directions for further development.

The objective of the study was to determine under what conditions each of the technologies—JavaScript, WebAssembly, and WebGPU—demonstrates peak efficiency. For each task, a series of test runs were conducted with varying input sizes, along with a comparative analysis of implementations with and without SIMD, using techniques that optimize data locality and other performance-enhancing strategies. Testing included warm-up runs (10 iterations) to stabilize cache behavior, followed by 50 measurement runs. For each, average time, median, minimum and maximum execution times were recorded, along with the calculated speedup relative to the baseline JavaScript implementation.

The methodological framework was based on a stepwise comparison of several software configurations solving the task of multiplying two square matrices of size N × N. Benchmark sizes were chosen as N = 256, 512, and 1024, allowing for an evaluation of scalability across three representative data scales aligned with practical use cases.

At the baseline level, two JavaScript implementations were evaluated. The first—"naive algorithm"—used a straightforward row-by-column traversal pattern, while the second incorporated partial transposition of one matrix and loop reordering to improve memory access continuity and reduce cache line miss penalties.

Three WebAssembly configurations were tested. The initial version operated without SIMD and without

memory control, representing "raw" hardware-software performance. The second version introduced SIMD instructions to process multiple elements in parallel and reduce the number of iterations in the main loop. The third combined SIMD with optimized read/write patterns—specifically, block transposition and block-wise memory access—to minimize cache conflicts and further boost execution speed.

The GPU-focused segment of the experiment featured two hybrid WebGPU-based configurations. In the "JavaScript + WebGPU" variant, data arrays were prepared in JavaScript and then passed to a compute shader executed on the GPU. In the "WebAssembly + WebGPU" variant, part of the setup and computation logic was offloaded to Wasm, while the shader execution remained managed by JavaScript. This setup allowed the study to assess whether Wasm preprocessing could provide tangible benefits when utilizing the same GPU core.

Quantitative results for each configuration are summarized in Table 3, which includes both absolute execution times and relative speedup factors compared to the baseline JavaScript implementation.

Table 3: Main results (example for a 1024×1024 matrix)

Implementation	**Avg (ms)**	**Median (ms)**	**Min (ms)**	**Max (ms)**	**Speedup (×)**
JavaScript (standard)	1579.97	1567.10	1550.50	1750.20	1.00×
JavaScript (optimized access)	1364.63	1359.20	1356.00	1450.30	1.16×
WebAssembly (baseline)	3866.64	3861.00	3808.00	4035.00	0.41×
WebAssembly (SIMD)	1762.92	1752.50	1731.00	1945.00	0.90×
WebAssembly (SIMD + access optimized)	961.52	957.00	954.00	1025.00	1.64×
JavaScript + WebGPU	30.84	30.20	29.10	45.00	~51×
WebAssembly + WebGPU	31.08	30.25	28.10	40.50	~51×

Analysis of the collected data reveals that the baseline WebAssembly implementation—lacking both SIMD vectorization and optimized memory access—performs worse than its JavaScript counterparts in linear matrix multiplication tasks. The primary reason lies in the advanced capabilities of modern JavaScript JIT compilers, which leverage dynamic instruction reordering and aggressive loop unrolling to reduce cache misses by aligning memory access patterns with cache line layouts. In contrast, non-SIMD WebAssembly executes operations sequentially and suffers from excessive read/write latency.

The integration of SIMD instructions into the WebAssembly module yields approximately a twofold

improvement in throughput compared to its initial version. This is confirmed by higher IPC (instructions per cycle) and reduced cycles per operation. However, such speedup is often insufficient to consistently outperform interpreted JavaScript code, especially when the algorithm retains an unstructured memory access pattern that fails to take advantage of aligned data blocks.

Only the combination of SIMD vectorization with structured matrix transposition for block-based read/write access enables WebAssembly to surpass JavaScript by more than 60%. This approach couples the width of vector registers with a cache-local blocking strategy, effectively mitigating latency spikes from cache misses and inefficient local memory management.

Offloading all computational workloads to the GPU using WebGPU leads to roughly a 50× speedup over CPU-based implementations. Within this setup, the performance gap between "JavaScript + WebGPU" and "WebAssembly + WebGPU" becomes negligible: since the primary arithmetic operations are executed in GPU shaders, CPU-side data preparation—regardless of whether written in JavaScript or Wasm—incurs minimal overhead.

For the image blurring task, based on a modified Gaussian convolution with separable filters, the experiment employed a two-pass scheme: horizontal filtering followed by vertical filtering. This strategy natively supports efficient vectorization by reducing the 2D convolution to two sequential 1D operations, significantly increasing SIMD register utilization and maximizing the CPU's cache hierarchy.

Test data consisted of images with resolutions of 256 × 256, 512 × 512, 1024 × 1024, and 2048 × 2048 pixels, with 8-bit channel representation. For each resolution, blur kernels with radii from 1 to 4 (i.e., 3 × 3, 5 × 5, 7 × 7, and 9 × 9 matrices) were applied, allowing an assessment of scalability and the impact of kernel size on throughput across configurations.

The methodology was implemented in four technological configurations: pure JavaScript, WebAssembly, and their respective combinations with WebGPU. In the latter cases, the CPU handled the preparatory logic, while the main computational workload was offloaded to the GPU. This range of implementations provides a comprehensive comparison of CPU and GPU performance under consistent experimental conditions.

Table 4 presents the average execution time (in milliseconds) and speedup relative to the JavaScript (CPU) baseline across different kernel sizes.

Table 4: Average execution time (ms) and acceleration relative to JavaScript (CPU) for various core sizes.

Implementation	**3×3 (ms)**	**Speedup**	**5×5 (ms)**	**Speedup**	**7×7 (ms)**	**Speedup**	**9×9 (ms)**	**Speedup**
JavaScript (CPU)	8.14	1.00×	11.08	1.00×	14.04	1.00×	17.22	1.00×
WebAssembly (CPU)	12.02	0.68×	15.56	0.71×	19.60	0.72×	22.30	0.77×
JavaScript + WebGPU	5.46	1.49×	5.40	2.05×	5.70	2.46×	6.16	2.80×
WebAssembly + WebGPU	5.36	1.52×	6.00	1.85×	5.28	2.66×	6.30	2.73×

Based on the experimental data presented in Table 4, it can be concluded that interpreted JavaScript running on the CPU often outperforms the WebAssembly variant—even with SIMD vectorization enabled. This apparent anomaly is explained by the fact that modern JavaScript engines conduct deep static and dynamic analysis of pixel array access patterns. Combined with aggressive JIT optimizations—including function inlining, dead code elimination, and branch prediction—this enables a reduction in cache misses and interline latency during memory access.

At the same time, offloading the computational workload to the GPU via WebGPU yields a significant performance boost over any CPU-based execution. As image resolution increases from the baseline up to 1024 × 1024 and then 2048 × 2048, the observed speedup continues to grow. This is attributed to the high degree of parallelism within GPU compute blocks and the superior memory bandwidth of video memory, which enables a broader front of simultaneous data processing.

Ultimately, the performance difference between the “JavaScript + WebGPU” and “WebAssembly + WebGPU” configurations is statistically negligible. Since the core algorithmic logic is moved into shader code, minor variations during the CPU-side preprocessing phase are virtually neutralized. As a result, overall performance is governed almost entirely by the architectural characteristics of the GPU and the efficiency of the shader implementation, rather than the language used for preprocessing.

A schematic of the data flow between JavaScript, WebAssembly, and WebGPU is presented below in Figure 1.

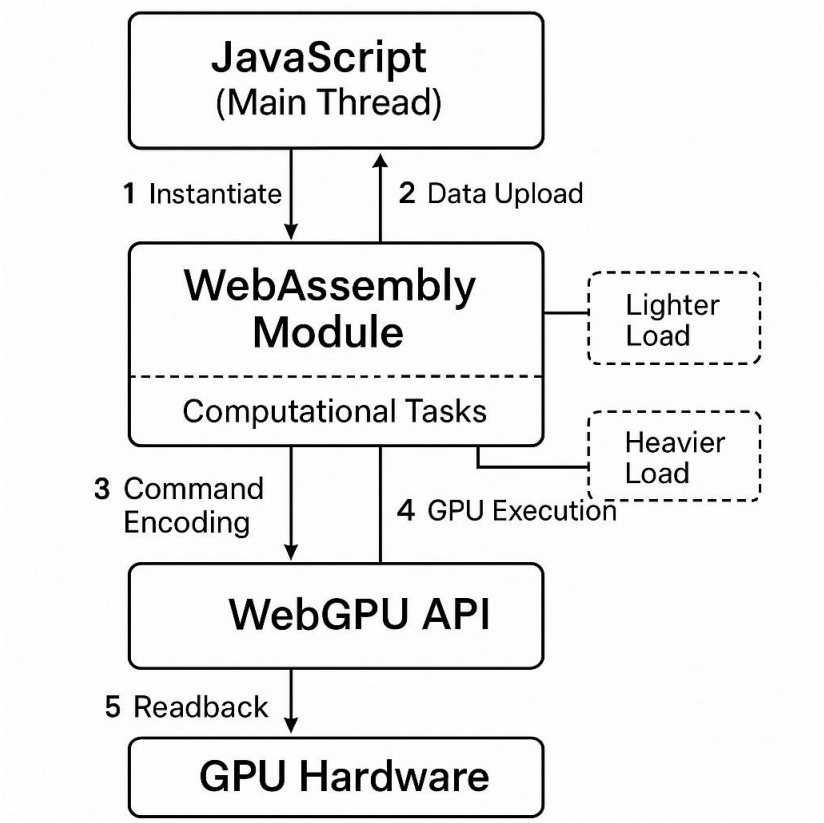

Figure 1: The source used for JavaScript, WebAssembly, and WebGPU

This separation of responsibilities and the sequence of execution provide a clean architectural design. Each component performs a narrowly defined task, contributing to improved system robustness and easier maintainability. Within this scheme, JavaScript functions as the coordinator, managing data flow and triggering subsequent operations. WebAssembly serves as a CPU-side preprocessor, responsible for fast compilation, optimization, and code preparation. Finally, WebGPU/GPU handles the core computation and rendering tasks, leveraging parallel execution on the graphics accelerator to achieve peak performance.

Despite the significant progress made in the development of WebAssembly and its broad adoption for enhancing the performance of web applications, several critical issues still limit its use under high-demand conditions, particularly in IoT environments. At the same time, the technology's developmental trajectory opens up opportunities to overcome these limitations and expand its functionality.

One of the main challenges is the lack of native garbage collection support in Wasm, which becomes especially noticeable when working with high-level languages that rely on dynamic memory management [9]. As a result, developers are forced to implement custom memory management solutions, which increases development complexity and raises the risk of memory leaks.

Another issue is the limited availability of debugging and testing tools for binary modules. While some protocols like DWARF for Wasm exist, current debuggers often provide less comprehensive information than those used with traditional languages, making error localization and resolution more difficult.

Inter-language integration is also a major constraint. Although Wasm is designed to integrate seamlessly with JavaScript, transferring complex data structures between Wasm modules and JavaScript typically requires additional conversion, which can introduce performance overhead and potential bugs [4, 10].

In IoT contexts, unique challenges arise due to constrained environments: limited RAM, low computational power, and restricted energy budgets. In such conditions, inefficient memory management and limited access to system and network APIs via Wasm create barriers to implementing advanced algorithms on low-level devices [2].

Despite these obstacles, several developmental directions promise to significantly extend WebAssembly's capabilities and resolve current limitations:

- Introduction of native garbage collection. The development of a standardized GC mechanism for Wasm will ease the use of dynamic memory management languages and reduce the likelihood of memory-related bugs [1].
- Expansion of multithreading and SIMD support. Prototypes demonstrating multithreaded execution and SIMD instruction usage show strong potential to accelerate data processing and improve Wasm module efficiency [2].
- Development of WASI and improved cross-platform integration. The WebAssembly System Interface (WASI) aims to provide a universal interface for system resource access, allowing Wasm modules to operate not just in browsers but also on servers and IoT devices, with consistent and secure behavior [1].
- Enhancement of debugging and profiling tools. Creating dedicated IDEs, debuggers, and profilers for Wasm will enable developers to more easily detect and fix bugs and optimize performance [4].
- Integration with WebGPU for compute acceleration. The synergy between Wasm and WebGPU allows offloading compute-heavy tasks to GPUs, which is especially valuable in applications involving image processing and AI inference [3].

The analysis shows that WebGPU delivers substantial performance gains—sometimes by an order of magnitude—thanks to the massive parallelization of compute shaders across hundreds or even thousands of workgroups. However, its current potential is limited by support for only a basic set of arithmetic operations (addition, multiplication, division, square root, etc.), making WebGPU especially effective for algorithms with straightforward, low-branching mathematical logic and no dynamic data structures. Tasks that require frequent CPU interaction or involve complex conditional branching fall outside the optimal use case for WebGPU.

WebAssembly, on the other hand, outperforms JavaScript in scenarios where the computational workload permits strictly sequential or block-based memory access, leverages SIMD instructions, and optimizes read/write order—for instance, by pre-transposing matrices for multiplication. In such conditions, Wasm demonstrates significantly higher memory throughput and more deterministic execution time than traditional JavaScript, owing to the low-level nature of its bytecode and its ability to analyze memory access patterns.

Nevertheless, interpreted JavaScript remains competitive when dealing with irregular or sparse access patterns

(e.g., localized pixel sampling in raster images). In these cases, JavaScript engines employ internal optimization strategies—such as inline caching, speculative optimization, and object field optimizations—while their low context-switch overhead makes JavaScript preferable for lightweight tasks where the cost of marshalling data into WebAssembly or GPU shaders would outweigh the performance benefits.

In the context of WebGPU, the near parity between JavaScript+WebGPU and Wasm+WebGPU configurations—each achieving roughly a 50× speedup—stems primarily from the GPU's computational dominance. Data-transfer times (from CPU memory to GPU memory) and the overhead of shader compilation and execution far outweigh the CPU-side preparation costs, rendering marginal performance differences between JavaScript and WebAssembly negligible. This outcome directly illustrates Amdahl's Law: as the parallel portion of the workload (GPU computations) accelerates dramatically, the sequential portion (CPU-side logic) becomes an ever-smaller fraction of total execution time.

5. Conclusion

Through architectural, experimental, and methodological analysis, it has been established that WebAssembly is capable of significantly enhancing the performance, energy efficiency, and security of the modern web stack. When combined with system-level languages such as Rust, C, or C++, and configured with a well-structured toolchain—Clang + wasm-ld, wasm-bindgen, wasm-pack—alongside hardware acceleration via WebGPU, SIMD extensions, and multithreading, WebAssembly can achieve over 50× reduction in execution time for typical linear algebra operations when offloaded to the GPU. For CPU workloads with predictable data locality, a stable performance gain of 60–70%over optimized JavaScript is also observed, along with deterministic behavior due to strict sandboxing and the elimination of side effects.

However, the lack of a built-in garbage collector, immature debugging infrastructure, and interlanguage marshalling overhead continue to limit WebAssembly's adoption in resource-constrained scenarios typical of edge and IoT platforms. Therefore, it is advisable to establish a cost-effective migration threshold early in the design process. Before porting compute-intensive logic, developers should measure the total data marshalling overhead between JavaScript and Wasm; if this exceeds 15–20% of overall execution time, it's preferable to first optimize the data structure (e.g., SOA/AOS layout, memory alignment), introduce a linear memory pool, and only then proceed with the migration. This profiling → zero-copy → Wasm strategy typically provides a 1.3–1.5× speedup compared to naïve "lift-and-shift" approaches.

Building a continuous pipeline (Rust → Wasm → WebGPU) requires a systemic mindset and tight dependency control. Projects should be compiled using: cargo build --target wasm32-unknown-unknown -Z build-std=std,panic_abort with LTO and PGO flags enabled to produce compact binaries. It's recommended to pin versions of wgpu-core and nagain the lockfile, and use wasm-opt -O3 --enable-simd --enable-multivalue -g source-map for production builds. Integrating the dawn/wgpu engine provides a unified codebase for cross-platform deployment, as WebGPU shaders are automatically translated into SPIR-V, MSL, or HLSL. A monorepo with single-step builds helps reduce regressions and cuts MTTR from days to hours.

Running Wasm modules in production requires multi-level observability. During development, code should be compiled with DWARF-5 and source maps, symbolic info stored in the CI environment, and llvm-symbolizer connected via vscode-wasm. In integration pipelines, it's important to collect performance.mark and resourceTiming metrics, tag them with a version hash, and export to OpenTelemetry or Jaeger. In production, enabling experimental wasm-exception-handling (in Chrome and Firefox Nightly) provides precise stack traces, while crash dumps should be sent to Sentry with automatic symbol demangling. This end-to-end traceability reduces critical defect localization from hours to minutes and ensures consistent system behavior even as runtimes evolve.

However, this study has several limitations. First, the empirical evaluation focused on only two computational kernels—dense-matrix multiplication and Gaussian convolution—which, while representative of certain high-performance domains (e.g., image processing), may not generalize fully to the diverse workloads encountered in real-world web applications, such as those involving irregular data structures, complex control flow, or string manipulation. Second, all performance measurements were conducted in a specific hardware–software environment—details on browser versions and CPU/GPU models were not exhaustively documented—so variations in these components could yield quantitatively different outcomes. Finally, given the rapid evolution of WebAssembly, WebGPU, and JavaScript technologies, the reported performance characteristics may shift as these platforms continue to mature.

In conclusion, WebAssembly is steadily emerging as the central layer of a high-performance client-side stack, especially when paired with WebGPU and WASI. By following best practices—defining cost-efficient migration thresholds, implementing unified CI pipelines, and embedding full observability—developers can extract maximum value from Wasm today, while minimizing risks associated with its relative immaturity and laying a solid foundation for scalable, energy-efficient web and edge computing solutions.

References

[1]. Odume B. W., Okodugha P. E., Madu I. Leveraging WebAssembly and webgpu for efficient integration of ai models into web applications //Available at SSRN 5078445. – 2024. – pp.1-17.

[2]. Ray P. P. An overview of WebAssembly for IoT: Background, tools, state-of-the-art, challenges, and future directions //Future Internet. – 2023. – Vol. 15 (8). – pp. 275.

[3]. Kim J. et al. Hardware-Based WebAssembly Accelerator for Embedded System //Electronics. – 2024. – Vol. 13 (20). – pp. 3979.

[4]. Kyriakou K. I. D., Tselikas N. D. Complementing javascript in high-performance node. js and web applications with rust and webassembly //Electronics. – 2022. – Vol. 11 (19). – pp. 3217.

[5]. Perrone G., Romano S. P. WebAssembly and Security: a review //Computer Science Review. – 2025. – Vol. 56. – pp. 1-10.

[6]. Ramirez C. E., Sermet Y., Demir I. HydroCompute: An open-source web-based computational library for hydrology and environmental sciences //Environmental Modelling & Software. – 2024. – Vol. 175. – pp. 1-15.

[7]. Zhang Y. et al. Research on WebAssembly Runtimes: A Survey //ACM Transactions on Software

Engineering and Methodology. – 2024. – pp.2-13.

[8]. Wang W. Empowering web applications with webassembly: Are we there yet? //2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE). – IEEE, 2021. – pp. 1301-1305.

[9]. Dejaeghere J. et al. Comparing security in ebpf and webassembly //Proceedings of the 1st Workshop on eBPF and Kernel Extensions. – 2023. – pp. 35-41.

[10]. Hoque M. N., Harras K. A. Webassembly for edge computing: Potential and challenges //IEEE Communications Standards Magazine. – 2023. – Vol. 6 (4). – pp. 68-73.