VIII Simpósio de Iniciação Científica 2025

"Do conhecimento acadêmico à transformação sustentável: inovação com validação científica"

BENCHMARK DE DESEMPENHO E PORTABILIDADE DO WEBASSEMBLY EM ALGORITMOS COMPUTACIONAIS

Vivian Leite Fragoso (IC)[1], Bruno Tardiole Kuehne (PQ)[1]
[1]*Universidade Federal de Itajubá.*

Palavras-chave: Algoritmos. Benchmark. Desempenho. Portabilidade. WebAssembly.

Introdução

O avanço de aplicações modernas em áreas como processamento numérico e Inteligência Artificial exige alto desempenho computacional, tradicionalmente obtido por meio de linguagens compiladas e arquiteturas otimizadas (MALLE et al., 2018). No entanto, a crescente diversidade de ambientes de execução — desde navegadores web até dispositivos embarcados — coloca a portabilidade como um requisito igualmente importante. Nesse cenário, torna-se imperativo avaliar alternativas que conciliem eficiência e flexibilidade *cross-platform*.

O JavaScript, embora seja a linguagem predominante na Web, enfrenta limitações inerentes em cenários que demandam intensivo processamento numérico, sendo até **6,5 vezes mais lento** e **4,45 vezes mais consumidor de energia** do que a linguagem C em aplicações típicas de benchmark (DE MACEDO et al., 2022). Ainda que tenha evoluído com máquinas virtuais modernas e compilação *Just-in-Time*, o JavaScript não foi concebido com foco em desempenho (DE MACEDO et al., 2022).

Como resposta a esses desafios, o WebAssembly (Wasm) emerge como alternativa portátil e eficiente (HAAS et al., 2017). Pesquisas iniciais, como a de Malle et al. (2018), logo identificaram seu potencial, observando que o Wasm conseguia atingir e, em alguns casos, até superar o desempenho de implementações nativas em certos cenários.

Estudos recentes avaliam o Wasm para Edge AI. Khelifa et al. (2024) constataram que runtimes standalone atingem baixa sobrecarga (~1.1x) em redes menores (ex: MobileNet), mas o desempenho degrada substantivamente em modelos maiores (ex: VGG), com análises concentradas em x86-64.

Diante desse contexto, esta pesquisa visa ampliar a compreensão sobre o desempenho e a portabilidade do WebAssembly por meio de uma análise experimental comparativa. O desempenho do Wasm é avaliado em relação ao JavaScript e ao código nativo em C++, utilizando um conjunto de algoritmos computacionalmente intensivos de processamento numérico e Machine Learning, selecionados para abranger diferentes perfis de carga computacional e permitir uma análise abrangente sob diversas condições.

Para tal, buscou-se:

1. Mensurar o desempenho em termos de tempo de execução e consumo de memória, variando os tamanhos de entrada;
2. Avaliar a portabilidade do WebAssembly mediante compilação em um dispositivo ARM (Apple M1);
3. Comparar diretamente as implementações em WebAssembly e JavaScript, identificando cenários de vantagem significativa;
4. Discutir limitações atuais e perspectivas futuras do WebAssembly, situando seu desempenho em relação ao código nativo.

A abordagem adotada foi experimental, com execuções controladas e repetidas em JS (Node.js e navegador), Wasm e C++ nativo. Utilizou-se o Emscripten, um compilador de código-fonte para WebAssembly que converte programas em C/C++ em módulos Wasm, permitindo sua execução em ambientes web e não-web (EMSCRIPTEN AUTHORS, s.d.), em dois modos (ccall/cwrap e standalone). Para garantir a confiabilidade dos dados, adotou-se geração determinística de dados, semente fixa e descarte de execuções iniciais para evitar viés de warm-up (DE MACEDO et al., 2022).

Este trabalho busca oferecer uma visão prática e fundamentada sobre a viabilidade do WebAssembly como alternativa de alto desempenho e portabilidade.

Metodologia

Etapa 1 — Experimentos em Node.js (JavaScript vs WebAssembly)

Foram realizados testes comparativos entre JavaScript (JS) e WebAssembly (Wasm) utilizando Node.js v24.4.1, executados em um computador com Apple M1 (8 núcleos, 8 GB RAM, macOS Sonoma).

As versões JS foram implementações puras dos algoritmos. Os módulos Wasm foram gerados a partir de código C++ com o compilador Emscripten (emcc), utilizando as opções -s WASM=1, -s MODULARIZE=1, -s EXPORT_NAME=... e -s ENVIRONMENT=node/browser, resultando em módulos não-standalone carregados via ccall e cwrap.

As métricas coletadas nesta etapa foram:

- Tempo de execução real (ms);
- Validação da saída (comparação entre JS e Wasm).

Etapa 2 — Experimentos no Navegador (JavaScript vs WebAssembly)

Nesta etapa, os mesmos algoritmos foram executados no Google Chrome, no mesmo dispositivo Apple M1. Os módulos Wasm utilizados foram os mesmos da etapa anterior, enquanto o código JS foi adaptado ao ambiente do navegador. Coletou-se tempo de execução e foi realizada a validação da saída.

Etapa 3 — Experimentos em dispositivo embarcado (C++ vs WebAssembly)

Foram comparadas implementações nativas em C++ com módulos WebAssembly standalone (WASI), gerados com a opção -s STANDALONE_WASM=1 no Emscripten. Esses módulos foram previamente validados em ambiente macOS (Apple M1) e, em seguida, transferidos para execução em uma TV Box com arquitetura ARM/Linux, utilizando o runtime Wasmtime, sem necessidade de recompilação. Os binários nativos em C++ foram compilados diretamente no dispositivo TV Box.

Além das métricas anteriores, esta etapa incluiu:

- Uso de memória residente (RSS, em KB) via GNU time;
- Código de saída de cada execução.

Algoritmos e Parâmetros

Foram utilizados quatro algoritmos com entradas em três tamanhos (small, medium, large):

- **Multiplicação de Matrizes (MatMul)**: 50×50, 500×500, 1000×1000;
- **Gradiente Descendente**: 100×10, 1000×100, 10000×1000 iterações/parâmetros;
- **Perceptron**: 50 mil, 200 mil e 1 milhão de épocas;
- **K-Means**: 4096, 16384 e 65536 pontos.

Todos os algoritmos foram implementados de forma determinística, com semente fixa para geração de números pseudoaleatórios, garantindo reprodutibilidade. As saídas foram validadas em todas as execuções antes da análise das métricas.

Protocolo Experimental

Nas etapas 1 e 2, cada algoritmo foi executado 55 vezes, com descarte das 5 primeiras execuções para mitigar efeitos de aquecimento, totalizando 50 execuções válidas por cenário.

Na etapa 3, realizada exclusivamente na TV Box, foram executadas 35 repetições por algoritmo, com descarte inicial de 5 execuções e posterior análise de 30 execuções válidas por configuração.

Análises estatísticas

Os dados coletados foram analisados estatisticamente com o objetivo de verificar diferenças significativas entre os **modos de execução** e os algoritmos. Para isso, aplicou-se uma **ANOVA multifatorial**, considerando três fatores: **modo de execução** (JavaScript, WebAssembly, C++ nativo), **algoritmo** (MatMul, Gradiente, Perceptron, K-Means) e **tamanho de entrada** (small, medium, large). Também foram calculados **intervalos de confiança de 95%** para as médias, permitindo avaliar a variabilidade e a confiabilidade dos resultados. A adoção dessas técnicas segue recomendações metodológicas para experimentos de desempenho em ciência da computação (GEORGES et al., 2007). Os testes foram realizados com auxílio das bibliotecas **SciPy** e **statsmodels** em Python, e os dados foram previamente organizados em **DataFrames** contendo os resultados individuais de cada execução.

Resultados e discussão

Para a primeira etapa a análise de variância multifatorial

(ANOVA) demonstrou efeitos estatisticamente significativos para todos os fatores considerados: modo de execução, algoritmo e tamanho da entrada, bem como todas as interações entre eles. Isso indica que o tempo de execução é influenciado de forma combinada por esses elementos. Complementarmente, os intervalos de confiança de 95% calculados para os tempos médios revelaram que, na maioria dos casos, as implementações em WebAssembly apresentaram desempenho superior ao JavaScript, com diferenças estatisticamente significativas e intervalos não sobrepostos.

	sum_sq	df	F	PR(>F)
C(engine)	2.543009e+06	1.0	1.089158e+06	0.0
C(algorithm)	1.790867e+07	3.0	2.556731e+06	0.0
C(size)	1.406839e+07	2.0	3.012710e+06	0.0
C(engine):C(algorithm)	3.811772e+06	3.0	5.441876e+05	0.0
C(engine):C(size)	3.322312e+06	2.0	7.114648e+05	0.0
C(algorithm):C(size)	2.567548e+07	6.0	1.832780e+06	0.0
C(engine):C(algorithm):C(size)	5.385737e+06	6.0	3.844474e+05	0.0
Residual	2.745771e+03	1176.0	NaN	NaN

Figura 1 – Tabela ANOVA Multifatorial Js vs Wasm (Node)

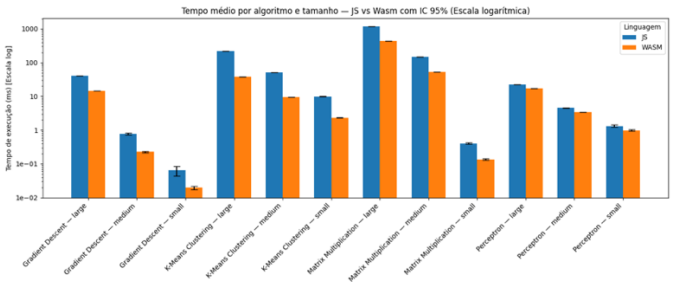

Figura 2 – Tempo médio por algoritmo e tamanho – Js vs Wasm (Node) com IC 95%

A Figura **2** apresenta os tempos médios de execução para os algoritmos testados, separados por modo de execução e tamanho de entrada. A escala logarítmica permite observar de forma clara os tempos extremamente baixos registrados nas entradas menores (ex: *Gradient Descent – small,* com valores abaixo de 0.1 ms). Em todos os casos, o WebAssembly apresentou desempenho superior ao JavaScript, com diferenças particularmente acentuadas em algoritmos com maior complexidade computacional, como K-Means e Multiplicação de matrizes. Os intervalos de confiança de 95% confirmam a robustez dos resultados, com baixíssima variabilidade entre execuções. Na segunda etapa do experimento, realizado no ambiente do navegador, os tempos médios de execução foram analisados com seus respectivos intervalos de confiança de 95%. Os resultados mostram que o WebAssembly apresenta desempenho superior ao JavaScript em todos os algoritmos e tamanhos de entrada. A diferença é mais acentuada para tamanhos maiores, evidenciando a escalabilidade superior do Wasm em cenários computacionalmente mais intensivos. A ANOVA multifatorial confirmou que os efeitos principais (*engine*, algoritmo e tamanho) e as interações entre *engine* × algoritmo e *engine* × tamanho são estatisticamente significativas, demonstrando que a escolha da *"engine"* impacta de forma relevante e variável conforme o contexto do algoritmo e a carga de entrada. Esses achados reforçam o potencial do Wasm para aplicações de alto desempenho em navegadores.

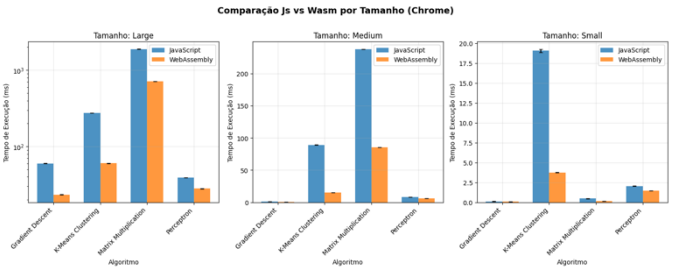

Figura 3 – Tempo médio por algoritmo e tamanho – Js vs Wasm (Chrome) com IC 95%

A Figura **3** apresenta os tempos médios de execução dos algoritmos avaliados no Chrome. Em todos os cenários testados, o WebAssembly demonstrou superioridade de desempenho frente ao JavaScript, com vantagens particularmente expressivas em algoritmos computacionalmente intensivos, como *K-Means Clustering* e *Matrix Multiplication*, onde chegou a ser até **4,5** vezes mais rápido. Os intervalos de confiança de 95%, corroboram a consistência dos resultados, apresentando variabilidade insignificante entre as execuções repetidas.

Na terceira etapa a análise dos tempos médios revelou que o WebAssembly é consistentemente mais lento que a implementação nativa em todos os cenários. A razão de overhead variou entre 1,8× a 20×, com média de aproximadamente 2,8×. O maior overhead real foi observado para o algoritmo Gradiente Descendente com entrada *medium*, onde o WebAssembly foi 20 vezes mais lento que a versão nativa. Os intervalos de confiança de 95% não se sobrepuseram entre os grupos wasm e native, reforçando a significância estatística das diferenças.

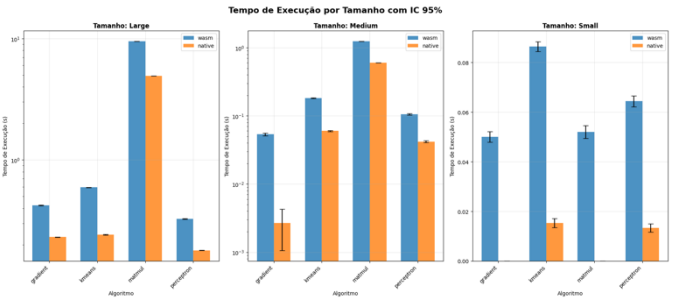

Figura 4 – Tempo médio por algoritmo e tamanho – Nativo vs Wasm (TV Box) com IC 95%.

Em relação à variabilidade, observou-se que o tempo de execução das versões nativas apresentou flutuações

maiores em cenários de entrada média, especialmente no algoritmo kmeans. No mesmo algoritmo, também foram registradas variações relevantes no tamanho pequeno, com desvio padrão próximo da média. As demais implementações, tanto em WebAssembly quanto em nativo, apresentaram amplitudes menores nos intervalos de confiança, com comportamento mais consistente. Em geral, a variabilidade foi mais perceptível nos tamanhos de entrada pequenos e médios, enquanto os cenários com entrada *large* apresentaram valores mais concentrados.

A ANOVA multifatorial apontou que todos os fatores e interações avaliados impactam significativamente o consumo de memória. O WebAssembly consumiu de 5 a 6 vezes mais memória que o código nativo em todos os cenários, com intervalos de confiança de 95% não sobrepostos, indicando diferenças robustas.

O algoritmo de multiplicação de matrizes teve o maior uso, chegando a 40.256 KB no WebAssembly (l*arge*), frente a 24.152 KB na versão nativa.

Conclusões

Os resultados demonstram que o **WebAssembly** supera consistentemente o **JavaScript** em desempenho tanto no Node.js quanto no navegador, especialmente em algoritmos computacionalmente intensivos como K-Means e Multiplicação de Matrizes, chegando a ser até **4,5** vezes mais rápido**.** No entanto, quando comparado ao código nativo em C++ executado em ambiente embarcado (ARM/Linux), o Wasm apresentou um *overhead* médio de **2,8×** em tempo de execução e consumiu **5 a 6 vezes** mais memória, evidenciando que sua portabilidade *cross-platform* tem um custo significativo em termos de eficiência. Ainda assim, confirmou-se sua capacidade de execução sem recompilação em ambientes distintos. Portanto, o WebAssembly consolida-se como uma alternativa viável para aplicações web de alto desempenho, embora a escolha entre portabilidade e desempenho máximo deva ser ponderada conforme os requisitos do projeto.

Para trabalhos futuros sugere-se a investigação do consumo energético em ambientes embarcados, a expansão dos experimentos para outras plataformas de *hardware* e *runtimes* de WebAssembly, bem como a exploração de extensões de otimização como *SIMD (Single Instruction, Multiple Data)* e suporte a *multithreading*, visando à mitigação do *overhead* de desempenho e à aproximação dos níveis de eficiência observados em implementações nativas.

Agradecimentos

Agradeço ao Professor Doutor Bruno Tardiole Kuehne pela orientação e pelas contribuições técnicas fundamentais ao desenvolvimento desta pesquisa. À Universidade Federal de Itajubá (UNIFEI), ao Instituto de Engenharia de Sistemas e Tecnologia da Informação (IESTI) e ao seu corpo docente, agradeço pelo suporte, pela formação sólida e pelo incentivo contínuo. Ao CNPq, sou grata pelo apoio financeiro que viabilizou este projeto. Estendo meu reconhecimento aos funcionários da UNIFEI e aos colegas de curso e laboratório pelo apoio e colaboração ao longo desta jornada.

Referências

DE MACEDO, João et al. **WebAssembly versus JavaScript: Energy and Runtime Performance**. In: INTERNATIONAL CONFERENCE ON ICT FOR SUSTAINABILITY – ICT4S, 2022, Lisboa. **Anais** […]. Lisboa: IEEE, 2022. p. 24–30. Disponível em: https://ieeexplore.ieee.org/document/9830108. Acesso em: 10 nov. 2024.

EMSCRIPTEN AUTHORS. **Emscripten: an LLVM-to-WebAssembly Compiler.** [S. l.], s.d. Disponível em: https://emscripten.org/. Acesso em: 18 ago. 2025.

GEORGES, Andy; BUYTAERT, Dries; EECKHOUT, Lieven. **Statistically Rigorous Java Performance Evaluation**. In: CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, SYSTEMS, LANGUAGES, AND APPLICATIONS – OOPSLA, 2007, Montréal. **Anais** […]. New York: ACM, 2007. p. 57–76. Disponível em: https://dri.es/files/oopsla07-georges.pdf. Acesso em: 23 mar. 2025.

HAAS, Andreas et al. **Bringing the web up to speed with WebAssembly.** In: ACM SIGPLAN CONFERENCE ON PROGRAMMING LANGUAGE DESIGN AND IMPLEMENTATION, 38., 2017, Barcelona. **Proceedings** […]. New York: ACM, 2017. p. 185-200. Disponível em: https://dl.acm.org/doi/pdf/10.1145/3062341.3062363. Acesso em: 28 mar. 2025.

KHELIFA, Saif eddine et al. **Case study of WebAssembly Runtimes for AI Applications on the Edge.** In: IEEE INTERNATIONAL CONFERENCE ON COMMUNICATIONS, 2024, Denver. Proceedings […]. New York: IEEE, 2024.Disponível em: https://ieeexplore.ieee.org/document/10449907. Acesso em: 1 jun 2025.

MALLE, Bernd et al. **The Need for Speed of AI Applications: Performance Comparison of Native vs. Browser-based Algorithm Implementations**. 2018. Disponível em: https://arxiv.org/abs/1802.03707. Acesso em: 5 jun. 2025.