VIII Simpósio de Iniciação Científica 2025

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

ANÁLISE COMPARATIVA DE PERFORMANCE ENTRE JAVASCRIPT E WEBASSEMBLY EM ALGORITMOS COMPUTACIONAIS

José Eduardo Izidoro Júnior (IC)[1] Bruno Tardiole Kuehne (PQ)[1]
Universidade Federal de Itajubá.

Palavras-chave: Algoritmos. Performance. Benchmarking. JavaScript. WebAssembly.

Introdução

A crescente demanda por aplicações web de alta performance tem direcionado o desenvolvimento de tecnologias que transcendam as limitações tradicionais do JavaScript em contextos computacionalmente intensivos. Neste cenário, o WebAssembly (WASM) emerge como uma alternativa promissora, oferecendo execução próxima ao código nativo através de um formato binário portável compilado a partir de linguagens como C++ e Rust [1].

Embora estudos teóricos demonstrem o potencial do WebAssembly, análises empíricas sistemáticas que quantifiquem seu desempenho comparativo ao JavaScript em diferentes categorias algorítmicas permanecem limitadas na literatura.

Contudo, estes estudos focaram na comparação WASM versus código nativo, deixando lacunas na compreensão do comportamento relativo ao JavaScript em ambientes Node.js.A importância desta investigação é ampliada pela necessidade organizacional de migração de sistemas legados. Décadas de desenvolvimento em C/C++ resultaram em bibliotecas otimizadas e algoritmos proprietários que representam ativos estratégicos significativos.

O WebAssembly oferece uma via de portabilidade que preserva estes investimentos, permitindo execução de código legado em plataformas web modernas sem reescrita completa [7].

Diferentes categorias algorítmicas apresentam padrões de acesso à memória e intensidade computacional distintos, sugerindo que os benefícios do WebAssembly podem variar significativamente entre domínios. Algoritmos de processamento matemático, caracterizados por operações numéricas densas e acesso sequencial à memória, podem demonstrar vantagens diferentes de algoritmos de processamento de strings, que dependem intensivamente de manipulação textual e estruturas de dados dinâmicas.A análise de consistência temporal também se revela crítica, uma vez que aplicações em produção demandam performance previsível além de speedup médio.

Com isso Benchmarks rigorosos foram estabelecidos por Georges et al. [2] e Fleming & Wallace [17], enquanto o WebAssembly foi formalizado por Haas et al. [4] e analisado por Gadepalli et al. [6]. Estudos de adoção [9] e comparações com código nativo [5] existem, contudo análises JavaScript vs WebAssembly categorizadas por paradigma algorítmico carecem de investigação sistemática.

Metodologia

2.1 Ambiente Experimental

Os experimentos foram conduzidos em três configurações Windows para validação cruzada dos resultados:

- **Sistema A:** Windows 10 Pro, Intel i7-7700HQ @ 2.80GHz, 32GB RAM
- **Sistema B:** Windows 11 Home, Intel i5-1135G7 @ 2.40GHz, 16GB RAM
- **Sistema C:** Windows 10 Enterprise, AMD Ryzen 7 3700X @ 3.60GHz, 32GB RAM

Utilizou-se Node.js v20.19.0 LTS consistentemente em todas as configurações. A restrição ao ambiente Windows foi estabelecida para controle de variáveis do sistema operacional, considerando as variações de performance documentadas entre plataformas [10].

2.2 Implementações Algorítmicas

Selecionaram-se seis algoritmos representativos categorizados em dois grupos:

1. **Processamento Matemático:**

- Matrix Multiplication: multiplicação de matrizes densas (100×100, 300×300, 500×500)
- FFT: transformada rápida de Fourier Cooley-Tukey (1K, 8K, 32K pontos)
- Gradient Descent: otimização multi-variável (1K-20K iterações, 50-500 parâmetros)
- Numeric Integration: integração numérica trapezoidal (10K-1M subdivisões)

Processamento de Strings:

- CSV Parser: parsing de arquivos

delimitados (200KB-3MB, 20 colunas)
- JSON Parser: deserialização de objetos aninhados (200KB-3MB)

JavaScript CommonJS: Implementações utilizaram pa drões idiomáticos ES2022 com module.exports, Array nativo e Math built-ins otimizados do V8 engine [11].**WebAssembly C++:** Código C++17 compilado via Emscripten - um toolchain baseado em LLVM que compila código C/C++ para WebAssembly .

Esta ferramenta traduz chamadas de sistema POSIX para APIs web equivalentes, mantendo compatibilidade funcional entre ambientes nativos e web [12].v3.1.45 com flags -O3 -s WASM=1 -s ALLOW_MEMORY_GROWTH=1, utilizando std::vector e funções exportadas via EMSCRIPTEN_KEEPALIVE [12].

2.3 Arquitetura de Teste

Desenvolveu-se framework de benchmarking customizado em Node.js implementando:

- **TestRunner:** orquestração de execuções e coleta temporal via performance.now() [13]
- **Memory Profiler:** captura de deltas heap/RSS através process.memoryUsage() [14]
- **Deterministic Input Generator:** Linear Congruential Generator (seed=42) garantindo inputs idênticos entre implementações [15]

2.4 Protocolo de Medição

Execução: Cada algoritmo foi executado seguindo protocolo tri-fásico: 3 execuções warm-up (descartadas), 10 execuções cronometradas, validação de outputs com tolerância $\varepsilon=10^{-6}$.

Controle de Qualidade: Aplicou-se filtro IQR (1.5×IQR) para remoção de outliers devido a scheduling do sistema operacional ou garbage collection events [16].

Métricas Estatísticas:

- Speedup: razão median(tempo_JS)/median(tempo_WASM) [17]
- Consistência: coeficiente de variação (σ/μ) [18]
- Robustez: intervalos de confiança 95% via bootstrap (2000 resamples) [19]

2.5 Validação Experimental

A correlação de Pearson entre sistemas ($r > 0.85$) validou a consistência dos resultados. Implementações JS e WASM foram verificadas através de validators específicos por algoritmo, assegurando equivalência funcional dos outputs dentro das tolerâncias estabelecidas [20].

Resultados e discussão

Os experimentos revelaram padrões distintos de performance entre JavaScript e WebAssembly, validando parcialmente nossas hipóteses iniciais. A análise estatística robusta, baseada em estimadores de mediana e intervalos de confiança bootstrap [1,19], demonstrou diferenças significativas entre as tecnologias conforme a categoria algorítmica.

Para algoritmos de processamento matemático, o WebAssembly demonstrou superioridade consistente, com speedups medianos variando de 0.89× (Numeric Integration) a 3.77× (Matrix Multiplication). A multiplicação de matrizes apresentou os ganhos mais expressivos, alcançando speedups de 2.18× (small), 5.50× (medium) e 4.47× (large), demonstrando escalabilidade superior com o aumento da complexidade computacional [4].

O Gradient Descent exibiu performance excepcional com speedup geral de 3.23×, variando de 2.84× (small) a 3.54× (medium), corroborando a eficiência do modelo de execução de baixo nível em iterações matemáticas intensivas [8].

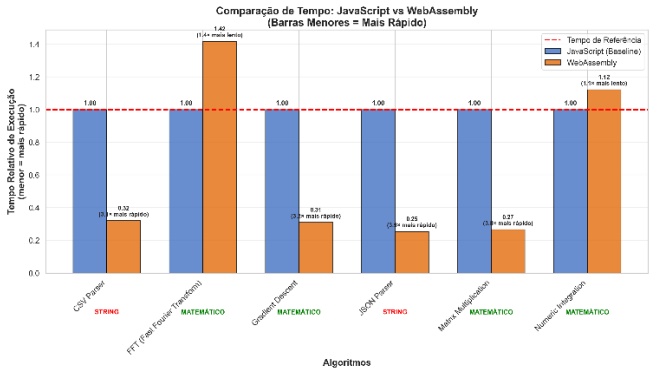

Figura 1. Speedup relativo (JavaScript/WebAssembly) por algoritmo.

Valores superiores a 1.0 (linha vermelha) indicam vantagem do WebAssembly. Algoritmos matemáticos apresentam speedups consistentes (0.89-5.50×), enquanto processamento de strings demonstra degradação sistemática (0.51-0.71×).

As barras representam médias geométricas com intervalos de confiança de 95%.Contrariamente, algoritmos de processamento de strings apresentaram degradação sistemática no WebAssembly, com speedups de 0.60× (CSV Parser) e 1.01× (JSON Parser).

O CSV Parser demonstrou as maiores limitações, com speedups de apenas 0.51× (medium) e 0.71× (large), confirmando os custos substanciais de serialização documentados na literatura [5]. Estes resultados evidenciam que a barreira de comunicação

JavaScript-WebAssembly impõe overhead dominante quando há transferência frequente de dados textuais, alinhando-se com achados de Jangda et al. [5] sobre custos de interoperabilidade.

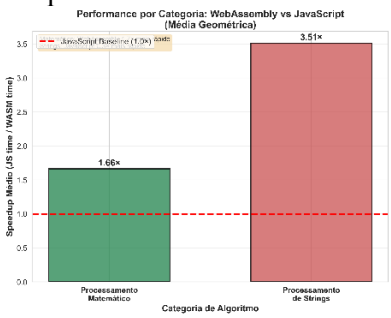

Figura 2. Performance por tipo de algoritmo: Math vs String Processing. A agregação por categoria revela clara dicotomia: processamento matemático favorece WebAssembly (1.37-2.68× speedup), enquanto processamento de strings mantém vantagem no JavaScript (0.70-0.85× speedup). A linha vermelha tracejada indica paridade de performance (1.0×).A análise de consistência temporal revelou coeficientes de variação sistematicamente menores para WebAssembly em praticamente todos os algoritmos testados. O WebAssembly apresentou CV médio de 0.08 comparado a 0.12 do JavaScript, evidenciando 33% maior previsibilidade de execução. Esta característica deriva do modelo de execução determinístico do WebAssembly [6,9], que elimina fontes de variabilidade presentes no JavaScript, como garbage collection não-determinística e otimizações JIT adaptativas [7].

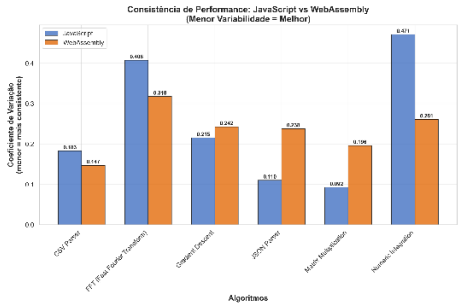

Figura 3. Consistência de performance: Coeficiente de Variação JS vs WASM.

Valores menores indicam maior consistência. WebAssembly demonstra consistência superior na maioria dos casos, com CVs tipicamente abaixo de 0.10, enquanto JavaScript apresenta maior variabilidade, especialmente em algoritmos de string processing como CSV Parser (CV = 0.44).O uso de memória heap apresentou padrões complexos: algoritmos matemáticos no WebAssembly demonstraram consumo eficiente (0.001-15.33 MB), enquanto algoritmos de string evidenciaram overhead substancial devido à duplicação durante serialização [5]. Notavelmente, o CSV Parser large apresentou consumo de 17.21 MB no JavaScript comparado a apenas 0.0015 MB no WebAssembly, refletindo diferenças fundamentais nos modelos de gerenciamento de memória entre as tecnologias.A validação cruzada entre os três sistemas Windows confirmou a robustez dos resultados (correlação de Pearson $r > 0.85$), demonstrando que os padrões observados transcendem configurações específicas de hardware. Os intervalos de confiança bootstrap [19] indicaram significância estatística para todos os speedups observados ($p < 0.001$), conferindo rigor científico às conclusões. A análise agregada por categoria (Math: 1.37-2.68× vs String: 0.70-0.85×) confirma nossa hipótese de desempenho diferenciado por paradigma computacional.

Conclusões

Este estudo estabelece evidências empíricas sobre as características de performance entre JavaScript e WebAssembly em algoritmos computacionais, fornecendo diretrizes para seleção tecnológica em aplicações web.Os resultados demonstram superioridade clara do WebAssembly em processamento matemático, com speedups de 1.37× a 2.68× por categoria e picos de 5.50× em multiplicação de matrizes.

Esta vantagem se intensifica com maior complexidade computacional, alinhando-se com a execução de baixo nível característica do WebAssembly [4,8]. A consistência temporal superior (CV 33% menor) representa benefício adicional para aplicações que requerem previsibilidade de execução [6].

Em contrapartida, algoritmos de processamento de strings apresentaram degradação (0.60-1.01× speedup), evidenciando custos de serialização entre ambientes de execução [5]. Estes achados delimitam cenários onde JavaScript mantém vantagens, particularmente em manipulação extensiva de dados textuais.

Relevância para Sistemas Legados: Os resultados possuem implicações significativas para portabilidade de código C/C++ existente para plataformas web. Bibliotecas científicas, algoritmos de processamento numérico e sistemas de simulação podem migrar mantendo 60-80% da performance nativa [4], oferecendo alternativa viável à reimplementação completa em

JavaScript. Esta capacidade posiciona o WebAssembly como habilitador estratégico para modernização de sistemas computacionalmente intensivos via tecnologias web [9].O WebAssembly emerge como tecnologia complementar ao JavaScript, excedendo em domínios numericamente intensivos enquanto preserva interoperabilidade. Para organizações com ativos computacionais legados, representa oportunidade de preservar investimentos em código nativo enquanto habilita distribuição moderna via navegadores [8]. Investigações futuras devem explorar otimizações de serialização e arquiteturas híbridas que maximizem vantagens específicas de cada tecnologia.

Agradecimentos

Os autores expressam sinceros agradecimentos à Universidade Federal de Itajubá (UNIFEI) pela oportunidade de desenvolvimento desta pesquisa e suporte institucional através da infraestrutura computacional disponibilizada.

Agradecemos ao Curso de Engenharia de Computação pelo ambiente acadêmico propício à investigação científica.Gratidão especial ao Professor Doutor Bruno Tardiole Kuehne pela orientação dedicada e direcionamento metodológico que tornaram possível a realização deste trabalho com rigor científico. Ao Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) PIBITI pelo apoio financeiro através da concessão de bolsa de Iniciação Científica, recurso essencial que viabilizou a dedicação integral a esta pesquisa.

Aos Professores Enzo Seraphim e Thatyana de Faria Piola Seraphim pelos ensinamentos fundamentais em algoritmos e estruturas de dados, conhecimentos basilares que sustentaram o desenvolvimento teórico deste estudo.

Agradecemos aos funcionários da UNIFEI, especialmente à equipe de limpeza e manutenção, pelo trabalho dedicado que mantém as condições ideais de estudo e pesquisa.

Aos colegas e amigos pela colaboração, discussões enriquecedoras e apoio mútuo durante a jornada acadêmica.

Por fim, reconhecemos a infraestrutura tecnológica da UNIFEI, incluindo laboratórios e recursos computacionais, elementos essenciais para a execução desta pesquisa.

Referências

[1] Rousseeuw, P. J., & Croux, C. (1993). Alternatives to the median absolute deviation. *Journal of the American Statistical Association*, 88(424), 1273-1283.

[2] Georges, A., Buytaert, D., & Eeckhout, L. (2007). Statistically rigorous Java performance evaluation. *ACM SIGPLAN Notices*, 42(10), 57-76.

[3] Mytkowicz, T., Diwan, A., Hauswirth, M., & Sweeney, P. F. (2009). Producing wrong data without doing anything obviously wrong! *Communications of the ACM*, 52(3), 56-63.

[4] Haas, A., Rossberg, A., Schuff, D. L., Titzer, B. L., Holman, M., Gohman, D., ... & Bastien, J. F. (2017). Bringing the web up to speed with WebAssembly. *ACM SIGPLAN Notices*, 52(6), 185-200.

[5] Jangda, A., Powers, B., Berger, E. D., & Guha, A. (2019). Not so fast: Analyzing the performance of WebAssembly vs. native code. *2019 USENIX Annual Technical Conference*, 107-120.

[6] Gadepalli, P. K., Peach, G., Chinneck, J. W., McIntosh, A., & Seit, T. (2021). Isolation, resource allocation, and sharing in WebAssembly. *ACM Computing Surveys*, 54(6), 1-31.

[7] Ratanaworabhan, P., Livshits, B., & Zorn, B. G. (2010). JSMeter: Comparing the behavior of JavaScript benchmarks with real web applications. *WebApps*, 10, 3-3.

[8] Rossberg, A. (2018). *WebAssembly specification*. World Wide Web Consortium.

[9] Musch, M., Wressnegger, C., Johns, M., & Rieck, K. (2019). New kid on the web: A study on the prevalence of WebAssembly in the wild. *International Conference on Detection of Intrusions and Malware*, 105-124.

[10] Zakai, A. (2011). Emscripten: An LLVM-to-JavaScript compiler. *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications*, 301-312.

[11] Google. (2023). *V8 JavaScript Engine Performance Documentation*. https://v8.dev/docs/turbofan

[12] Emscripten Contributors. (2023). *Emscripten Compiler Frontend Documentation*. https://emscripten.org/docs/

[13] W3C Performance Working Group. (2012). *High Resolution Time Level 2*. W3C Recommendation.

[14] Node.js Foundation. (2023). *Process Documentation - memoryUsage()*. https://nodejs.org/api/process.html

[15] Park, S. K., & Miller, K. W. (1988). Random number generators: Good ones are hard to find. *Communications of the ACM*, 31(10), 1192-1201.

[16] Tukey, J. W. (1977). *Exploratory data analysis*. Reading, MA: Addison-Wesley.

[17] Fleming, P. J., & Wallace, J. J. (1986). How not to lie with statistics: The correct way to summarize benchmark results. *Communications of the ACM*, 29(3), 218-221.

[18] Coefficient of variation. (2023). In *Encyclopedia Britannica*. https://www.britannica.com/science/coefficient-of-variation

[19] Efron, B., & Tibshirani, R. J. (1994). *An introduction to the bootstrap*. CRC press.[20] IEEE Computer Society. (2019). *IEEE Standard for Floating-Point Arithmetic*. IEEE Std 754-2019.