WebAssembly for browser-based scientific computing: neural simulation and Xiangqi (chinese chess)

Martin D. Pham

University of Toronto, Toronto, ON, Canada
martindopham@gmail.com

Abstract. We present two applications of WebAssembly for browser-based scientific computing. We first present a comparison of the Brian library extension Brian2WASM to a Rust-based spiking neural network implementation compiled to WebAssembly. We then present a WebAssembly module for hyperdimensional computing on the Xiangqi (Chinese chess) board game. Hyperdimensional computing, sometimes called vector symbolic architectures, is a framework for representing data as algebraic operations in a high dimensional vector spaces. We demonstrate preliminary results for board state encoding from Forsyth–Edwards notation and discuss next steps for implementing a vector symbolic game engine/artificial player. Together, these two examples demonstrate how WebAssembly can be deployed on the browser for client-side offloading of scientific computing.

Keywords: WebAssembly · Spiking Neural Network · Hyperdimensional Computing.

1 Introduction and motivation

Browser-based computing is the use of client-side resources, such as the CPU and memory, to perform computations within a web browser. This approach leverages the capabilities of modern web technologies to enable high-performance computing tasks without the need for server-side processing. Recent work has used browser-based tools for drug discovery [**?**] and for visualizing large datasets [**?**].

WebAssembly (Wasm) is a binary instruction format that allows code written in high-level languages to be compiled and run in web browsers at near-native speed [**?**]. It provides a safe and efficient way to execute code on the client side, making it an ideal candidate for scientific computing tasks [**?**]. Wasm has been used in high-performance computing (HPC) applications [**?**] and code-generation [**?**] for big data processing [**?**].

The remainder of the paper provides background on WebAssembly and the application domains, presents the SNN and HDC methods, reports the results of both experiments, and concludes with directions for future work.

1.1 WebAssembly

WebAssembly is designed to be a portable compilation target for programming languages, enabling high-performance applications on the web [**?**]. It is supported by all major browsers and provides a secure execution environment through a sandboxed model [**?**]. Wasm runtimes are responsible for executing WebAssembly code and providing the necessary abstractions for interacting with the host environment. Recent surveys by [**?**] and [**?**] provide a comprehensive state of the art overview of WebAssembly runtimes, including their architecture, performance characteristics, and use cases. The work of [**?**] further surveys the security aspects of WebAssembly, highlighting potential vulnerabilities and mitigation strategies. When compiled from both C++ and Java, Wasm has shown promise in improving performance and resource utilization on the Raspberry Pi [**?**]. Additionally, Rust [**?**] may be compiled to Wasm, making it an appealing choice for high-performance web applications. Such results suggest Wasm may be used for similar gains in browser-based computing, although there is still a need for more research in this area to fully understand the trade-offs and best practices for using WebAssembly in scientific computing applications.

1.2 Spiking Neural Networks

Spiking neural networks are a class of artificial neural networks that explicitly model temporal dynamics of neural activity, making them more biologically plausible than traditional artificial neural networks [?]. They are particularly well-suited for tasks involving time-varying signals and have been applied in various domains, including robotics, sensory processing, and neuromorphic computing. Neural activity is modeled as discrete events, or spikes, which occur at specific points in time and are driven by individual neuronal models [?]. This event-driven approach allows for more efficient processing of temporal information compared to traditional methods. However, training SNNs can be challenging due to the non-differentiable nature of spike events. Frameworks such as Brian use a code-generation strategy to initially represent the network and simulation definition as an abstract syntax tree before generating numerical solvers that may be used at runtime [?].

The temporally sparse representation of activity allows for low resource utilization, making SNNs a promising approach for online [?] and edge computing applications [?] [?] when combined with appropriate hardware. Spike-based models are of particular interest for their applications in neuromorphic computing on non-Von Neumann machines [?], however there is still many open problems regarding SNN dynamics and compilations. While many SNN simulators exist [?], there is still a need for more efficient and scalable solutions to fully leverage the potential of SNNs in practical applications and on traditional hardware environments such as client-side browsers.

1.3 Hyperdimensional Computing

Hyperdimensional computing, also known interchangeably as vector symbolic architectures (VSA), is a computational framework that represents data as high-dimensional vectors, typically with thousands of dimensions [?]. These hyperdimensional vectors are usually drawn from random projections, and with appropriate binding, bundling and other vector symbolic operations, they can be algebraically manipulated to represent complex data structures and relationships [?]. The work of [?] provides some theoretical framing of HDCs, and the work of [?] provides a foundation for a category theoretic understanding of VSAs. A comprehensive pair of surveys was done by [?] and [?] to summarize the state of the art in models and applications of HDCs, with a useful comparison across different VSAs done by [?].

Although libraries such as TorchHD [?] provide a platform to implement different VSAs, it remains a challenge to create efficient implementations that can be deployed in web browsers for client-side computing. An important approach is the one taken by Nengo [?], which uses the Neural Engineering Framework (NEF) [?] to implement a VSA called the Semantic Pointer Architecture (SPA) [?], a variation of Holographic Reduced Representations (HRR) [?] which uses SNNs. The Nengo software provides a frontend application based on a Python web server, which interprets the model and streams results back as they are computed.

1.4 Xiangqi (Chinese Chess)

Computer Chinese chess [?] is the computational study and implementation of the board game Xiangqi [?]. Strategies include pruning [?] and search-based algorithms [?], coevolutionary approaches [?], and more recently deep [?] and reinforcement learning [?]. Computer vision and robotic applications have have been successfully deployed using convolutional neural networks [?]. A recent perspective report by [?] provides an overview of the advancements and challenges in the field.

2 Methods

2.1 Brian2 vs Rust-based Simulators

Brian2Wasm [?] is a library extension for the Brian simulator which targets WebAssembly for compilation. Brian2 provides equation-based model specification with automatic code genera-

tion, including compiled backends; its standalone C++ mode can use threaded execution with suitable toolchains, while Brian2Wasm deploys generated kernels in the browser. The neuron model simulated is the Ornstein-Uhlenbeck process as seen in Figure 1, where y is the state variable, τ is the time constant, σ is the noise intensity, and ξ is a Gaussian white noise term. We simulate a neural population of 5000 neurons for 10 seconds of simulation time. We evaluate wall clock time for Brian-based and Rust-based implementations of this process across four configurations in Figure 3: Brian native, Brian2Wasm in browser, Rust native, and Rust+Wasm in browser. Native solvers use compiled/optimized builds, while browser solvers use WebAssembly. WebAssembly supports SIMD and thread-level parallelism via shared memory and atomics, but due to browser deployment constraints (cross-origin isolation), the reported browser runs use a single-threaded configuration.

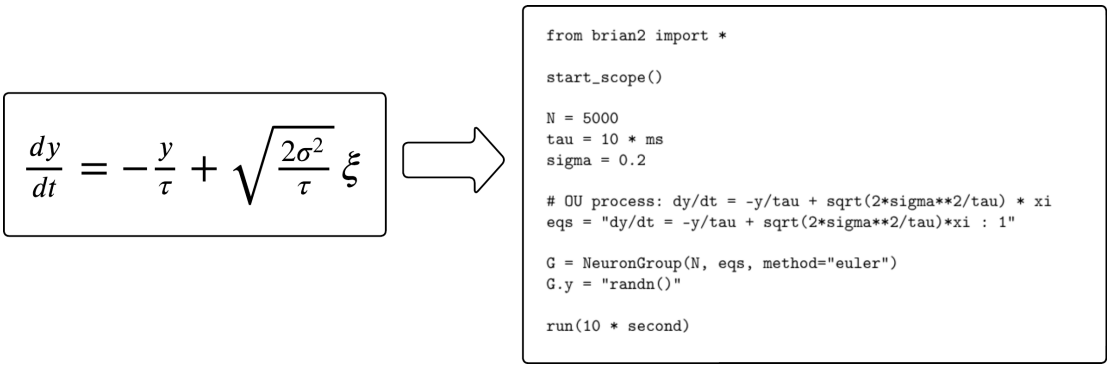

Fig. 1. Brian2 code snippet for the Ornstein-Uhlenbeck process, which can be compiled to WebAssembly for browser deployment.

Compared directly, vanilla Brian2 (native backend) prioritizes speed and parallel tuning, while Brian2Wasm prioritizes deployability and reproducibility in the browser. Vanilla Brian2 can use generated C++ and host-side threading to maximize throughput, whereas Brian2Wasm runs inside the browser sandbox, where threading depends on deployment constraints and can add runtime overhead. Relative to Brian, a Rust implementation requires explicit numerical integration logic, random-noise handling, memory layout choices, and separate native and Wasm build/tooling pipelines. This increases implementation effort but gives tighter low-level control over data structures and optimization, whereas Brian offers faster model iteration through high-level equations and automatic solver/code generation. In practice, optimization emphasis shifts by deployment: vanilla Brian favors solver/method selection and generated-backend tuning, Brian2Wasm favors browser-facing constraints (payload size, startup cost, and runtime compatibility), Rust native favors architecture-specific compiler/vectorization and memory-layout tuning, and Rust+Wasm favors Wasm-targeted size/performance tradeoffs (for example SIMD enablement, bindgen boundary minimization, and reduced host-call overhead). We intentionally deferred basic low-level optimizations (manual SIMD kernels, aggressive loop fusion/unrolling, and multi-threaded browser execution) in this comparison to preserve a like-for-like baseline across all four columns; computations are primarily IEEE-754 double precision as provided by Brian/Python and Rust defaults, and explicit vectorized operations were not hand-implemented beyond compiler/runtime auto-vectorization when available.

2.2 Xiangqi Hyperdimensional Encoding

Using a modified version of the Forsyth–Edwards Notation (FEN) [**?**], we encode the state of a Xiangqi board as a hyperdimensional vector as seen in Figure 2. The FEN string representing the board state is given by:

`rnbakabnr/9/1c5c1/p1p1p1p1p/9/9/P1P1P1P1P/1C5C1/9/RNBAKABNR`

where rows are separated by /, lowercase letters represent red pieces, uppercase letters represent black pieces, and numbers represent empty squares. By using a role-filler construction, we bind each piece to its position on the board and bundle all piece-position pairs into a single hyperdimensional vector representing the entire board state.

Let $\mathcal{T} = \{\text{RED}, \text{BLACK}\}$ be team symbols, $\mathcal{P} = \{K, A, B, N, R, C, P\}$ be piece symbols, and let X and Y be base row and column hypervectors. Let Π denote a fixed permutation operator, so that $\Pi^i(X)$ and $\Pi^j(Y)$ encode coordinate-dependent row/column roles. With binding operator $\otimes$ and bundling operator $\oplus$, each occupied square contributes

$$v_{i,j} = (t_{i,j} \otimes p_{i,j}) \otimes \left(\Pi^i(X) \otimes \Pi^j(Y)\right),$$

and the board representation is

$$V_{\text{board}} = \bigoplus_{(i,j)\in\Omega} v_{i,j},$$

where Ω is the set of occupied coordinates. To query the piece at a location (i, j), we unbind position and decode by similarity over the team-piece dictionary:

$$\tilde{u}_{i,j} = V_{\text{board}} \otimes \left(\Pi^i(X) \otimes \Pi^j(Y)\right)^{-1}, \qquad (\hat{t}, \hat{p}) = \arg\max_{t\in\mathcal{T},\, p\in\mathcal{P}} \text{sim}(\tilde{u}_{i,j},\, t \otimes p).$$

Conversely, to query location from a team-piece token (t, p), we compute

$$\tilde{\ell}_{t,p} = V_{\text{board}} \otimes (t \otimes p)^{-1}$$

and select the coordinate pair (i, j) with maximal similarity to $\left(\Pi^i(X) \otimes \Pi^j(Y)\right)$.

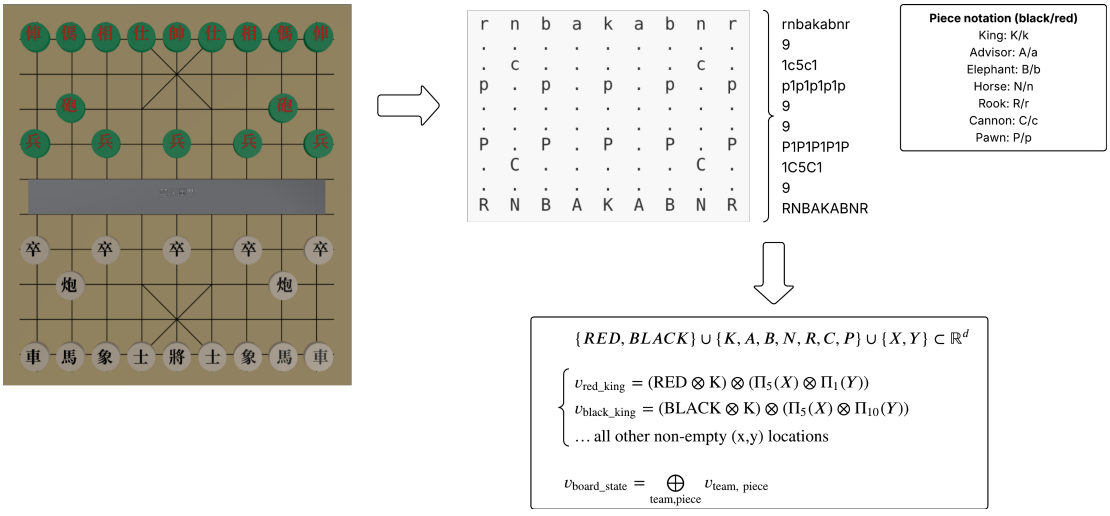

Fig. 2. The board state is first converted into a text string (FEN) and then encoded into a hyperdimensional vector using a location and piece-based role-filler construction.

3 Results and Discussion

3.1 Comparing Neural Solvers

Both browser-based and native implementations were evaluated over 100 runs. Browser-based simulations took longer than native ones, but the Rust-based Wasm implementation remained competitive despite the lack of additional optimization. Figure 3 shows the performance comparison of Brian2WASM and Rust-based SNN implementations for simulating the Ornstein-Uhlenbeck process.

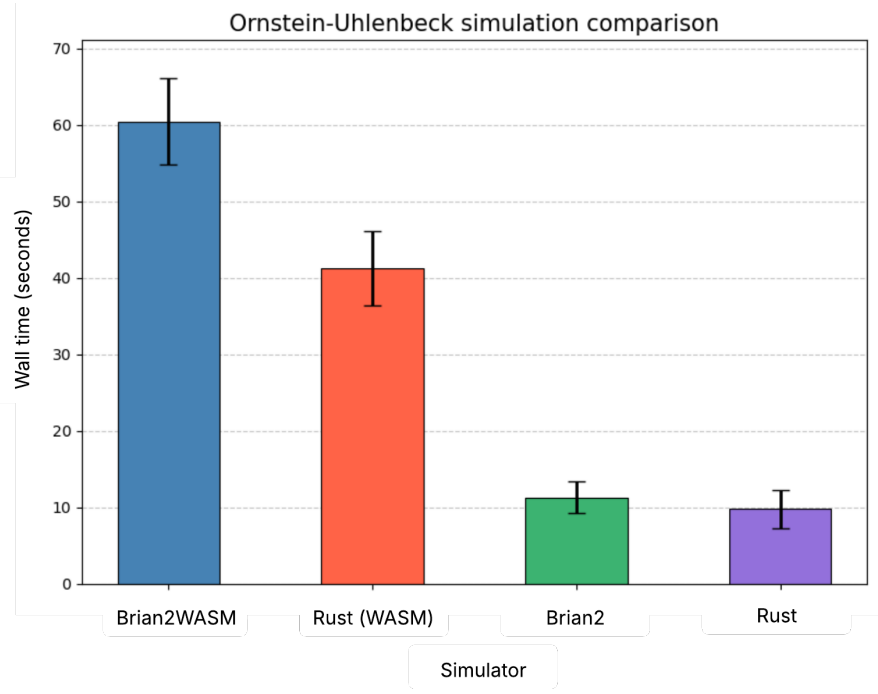

Fig. 3. Performance comparison of Brian2WASM and Rust-based SNN implementations for simulating the Ornstein-Uhlenbeck process over 100 runs. All simulations were run on a System76 Lemur Pro with 56Gb of RAM. Browser based simulations were run in Firefox. In all four columns, the computational core is compiled code (generated Brian kernels or Rust release binaries/Wasm modules), not interpreted numerical loops.

3.2 Xiangqi Querying

The computed distributed hypervector encoding the board state was queried against all 90 possible piece positions on the board. Over 100 runs, the encoding performed poorly, obtaining an average accuracy of only 24%($\pm$7%) in correctly identifying the piece at a given position. Many errors arose from misidentifying empty squares as occupied, likely because the encoding does not include an explicit representation for empty positions. The role-filler construction used for position vectors may also limit recall accuracy relative to alternative position encodings.

4 Conclusion and Future Work

We presented two WebAssembly-based case studies to evaluate Rust as a practical alternative to Brian2WASM for browser deployment of scientific workloads. For the SNN study, browser execution incurred a clear overhead relative to native runs (up to about 6$\times$ in our setting), but the Rust-based Wasm implementation remained functionally correct and competitive with Brian2WASM for client-side use. These results support browser-native deployment as a useful option when reducing server-side setup and scaling burden is important for shared simulation applications. For Xiangqi, the low query accuracy should be interpreted as a limitation of the current HDC encoding design rather than a refutation of the browser/Wasm approach, motivating future work on richer empty-square representations and alternative position-composition schemes.

Disclosure of Interests. The author has no competing interests to declare that are relevant to the content of this article.