Optimizing Web-Based Educational Simulations: Integrating Web Assembly to Improve Performance

A Technical Report Submitted to the Department of Computer Science

Presented to the Faculty of the School of Engineering and Applied Science

University of Virginia • Charlottesville, Virginia

In Partial Fulfillment of the Requirements for the Degree

Bachelor of Science, School of Engineering

Kevin Sandoval

Fall 2024

On my honor as a University Student, I have neither given nor received unauthorized aid on this assignment as defined by the Honor Guidelines for Thesis-Related Assignments

Briana Morrison, Department of Computer Science

Optimizing Web-Based Educational Simulations: Integrating Web Assembly to Improve Performance

CS4991 Capstone Report, 2024

Kevin Sandoval
Computer Science
The University of Virginia
School of Engineering and Applied Science
Charlottesville, Virginia USA
ucy5mh@virginia.edu

ABSTRACT

Many web-based educational simulations face performance bottlenecks, particularly on low-end machines, due to high central processing unit (CPU) usage. I propose integrating efficient algorithms and low-level optimization techniques into these simulations using Web Assembly (WASM) to reduce CPU load. The solution will employ techniques such as efficient memory management, single instruction multiple data (SIMD) instructions, and efficient matrix multiplication to increase the performance of CPU-intensive calculations. In addition to WASM, other environments will be tested for comparison, including asm.js, vanilla JS, and React. Metrics that will be tested include CPU usage, scripting time, and code timing. This project is expected to show that WASM is the fastest and least CPU-intensive environment. Future work can optimize graphical rendering in education simulations, a potential CPU-heavy operation that can be sped up with frameworks like WebGL.

1. INTRODUCTION

Many visual-based educational simulations are available online that help visualize concepts across various fields. When these simulations become more intensive, these visualizations become inaccessible to many people without the high-end hardware to efficiently run these simulations.

To illustrate the computational challenges present in some of these simulations, the performance of an electromagnet simulation was analyzed [1]. The Google Chrome browser allows CPU throttling to simulate a lower-end machine with different throttling amounts, which can roughly be compared to different CPUs. For example, with an M2 Apple Silicon chip, a 6x throttle amount roughly is comparable to a 2015 Intel Core i5 dual-core chip. The performance of the simulation under 20x CPU throttling resulted in a drastic performance degradation (measured as code scripting time and frame rate).

One aspect universal to nearly all web-based apps is the JS engines on which the simulation code runs. To address JS engine limitations, a binary instruction format known as WebAssembly (WASM) was introduced. According to the WebAssembly website (2024), this format seeks to execute code at native speed by leveraging more hardware-level capabilities [2]. I propose integrating WASM into intensive educational simulations and comparing their performances to simulations that do not use WASM.

2. RELATED WORKS

Haas, et al. (2017) evaluated the performance of WASM compared to asm.js, a subset of JS. Results indicated that, on average, WASM

was 33.7% faster than asm.js [3]. A limitation of this design is the nature of the tests performed, which only made comparisons between asm.js and WASM and between WASM and native code. My project will address this by performing tests on vanilla JS, React, asm.js, WASM, and native code.

Yan et al. (2021) evaluated the performance of WASM under various environments, including compiler-generated WASM and WASM used in real-world applications [4]. When 41 C programs were compiled to WASM and JS, WASM consistently performed better. Similar results were seen when comparing real-world applications (Long.js, Hyphenopoly.js, and FFmpeg). The performance evaluation approach proposed in this paper evaluated many small programs representing various use cases of WASM; however, most web applications are not simple programs or contain one single algorithm. My project will address this drawback by analyzing the performance of using WASM throughout large-scale web-based simulations.

3. PROPOSED DESIGN

A web-based educational simulation will be implemented in different environments. Starting from the C++ language environment, a physics-based simulation will be written and translated to other environments, including WASM. The different environments will be tested under various conditions to highlight performance differences.

3.1 Environments Tested

This project tests five different environments: WASM, JS, React, asm.js, and native code (which is compiled C++ code). In my project, environments are a combination of a programming language and how the code is executed in the browser (aside from native code). These environments differ in aspects such as automatic optimizations, computer hardware utilizations, and language features.

The JS environment is vanilla JS, meaning no use of a framework. A framework, alongside JS, will be tested since frameworks often enhance performance through built-in optimizations. React will be tested because its use of simple components to build complex systems is well-suited for building physics-based simulations.

Although WASM is not a language, it enables code to run at near-native speeds in the web browser by compiling C++ code into a binary format. asm.js compiles C++ code to JS while attempting to retain as many C++ features and optimizations as possible. Native code will directly compile C++ code and run the code outside of a web browser for comparison.

3.2 Implementation of Simulation

A physics simulation will be implemented in C++ that will simulate electric fields, simple motors, or some other physics system. To implement the chosen physics system in C++, essential components, such as forces, vectors, and fields, will be implemented as C++ classes. These classes will be translated into the equivalent version in JS and React. Then, C++ functions will be written that simulate the interactions of those components. C++ language optimization features/functions will be used as much as possible, such as SIMD instructions (which can run multiple math operations simultaneously instead of just one). After the initial simulation code is written in C++, the code will be manually translated, automatically translated, or compiled to the five different environments.

3.2.1 JS and React

A manual translation of the C++ code to vanilla JS and React will be done. This will require meticulously translating the C++ classes implementing the various physics elements into the JS equivalent (JS classes) and the React equivalent (TypeScript (TS)

classes or React components). The C++ functions modeling the physics interactions will be translated into JS/TS. This will involve a loss of some C++ language features.

3.2.2 WASM, asm.js, and Native Code

asm.js automatically compiles C++ code into JS code. WASM compiles C++ code into a binary format, which can be run in the browser. Finally, C++ can be compiled into machine code using a compiler such as Clang. The machine code can then be run natively on a machine, independent of a browser.

3.3 Performance Metrics

Various simulation metrics in each environment will be measured, including CPU usage, scripting time, and code timing in milliseconds. CPU usage measures the amount of the system's CPU used as a percentage. Scripting time, a web browser feature, measures how long individual JS functions take to run. Finally, code timing measures execution time and allows for more accurate measurements and control over what is timed by defining timing points within the code.

3.4 Data Collection Methods

Google Chrome's Task Manager, which displays the CPU usage percentage of each tab, will be used to measure CPU usage across the JS, React, asm.js, and WASM environments. I will use my computer's task manager, Activity Monitor, to measure the CPU usage for the native code. Throughout multiple points in the simulation, the CPU usage will be measured at a 5-second time interval and later averaged. Google Chrome's performance measurement tool will measure the entire simulation over different time intervals (10 seconds, 30 seconds, and 1 minute) to measure the scripting time. This is not available for the native code. Finally, for all environments, after the simulation is initialized, the time will be measured at the point the simulation starts, the simulation will run for a set number of iterations (between 10,000 and 100,000 iterations), and then the time will be measured at the end of those iterations.

Each environment's measurements will be collected multiple times and averaged to attain more accurate measurements. Using the results from this data collection process, the performance of all five environments will be compared to determine the best-performing environment.

4. ANTICIPATED RESULTS

Based on previous research, I anticipate that WASM will perform the best in CPU usage, scripting time, and code timing compared to the other web-based environments because WASM is designed to run code at near-native speeds. In addition, compiled C++ code can take advantage of low-level C++ optimizations such as SIMD instructions and manual memory management. This anticipated result aligns with the findings of Haas et al. (2017) and Yan et al. (2021), where WASM consistently performed better in smaller-scaled web applications.

I anticipate that Vanilla JS, React, and asm.js will not perform as well as WASM because of the lack of C++ low-level features. I anticipate potential overhead when calling C++ functions in the WASM environment, either from the compilation of C++ code or from invoking C++ functions within JS. However, over thousands of iterations, the C++ optimizations will likely reduce execution time for each function call, offsetting the overhead cost. I anticipate that the native code will perform the best overall since it can leverage the entire hardware of the system, but this is at the cost of not being portable to a web-based environment without first being compiled into a format that can be run in a web browser. Finally, I anticipate that asm.js will

perform better than vanilla JS, while React will perform better than asm.js.

5. CONCLUSION

Through the anticipated outcomes of this project, WASM is expected to be the most optimal environment for large-scale web projects. Hence, current and future educational simulations will likely benefit from speed and CPU usage improvements using WASM. While many speed improvements come from using C++ language features, developers of these simulations should consider C++'s steep learning curve. Additionally, developers should consider the greater chance of introducing software bugs by using C++'s more complex features, such as manual memory management.

Despite the difficulty associated with coding in C++, I believe implementing educational simulations should be written in C++ to take advantage of WASM's benefits. By reducing CPU usage, computers with less processing power can run these simulations. Thus, these simulations can be accessible to more students, especially considering many students can only afford low-end computers. Students from diverse backgrounds often learn complicated concepts more easily through visual demonstrations. By incorporating WASM into these simulations, more students can engage their visual learning processes to understand challenging topics better, likely enhancing educational outcomes.

6. FUTURE WORK

Most simulation programs contain an update and render loop. In the update step, the next state of the simulation is calculated. In the render step, that state is drawn to the web browser for the user to see. This project focused on the update step, but future work may focus on finding the most optimal rendering environment for these simulations. This is important because the rendering step often takes a lot of CPU usage, especially with more complicated systems with many moving objects to render. Many rendering environments can be tested, including basic HTML and CSS elements, HTML Canvas, Three.js, React, and WebGL.

Similarly to WASM, WebGL uses more of a computer's hardware, specifically the graphics processing unit (GPU). In addition to WebGL, frameworks like Three.js provide a library of functions that simplify WebGL usage, albeit at the cost of function call overhead. Combining WASM with WebGL or Three.js could significantly improve CPU efficiency and speed in educational simulations.

REFERENCES

[1] Magnets and Electromagnets. Retrieved November 15, 2024 from https://phet.colorado.edu/sims/html/magnets-and-electromagnets/latest/magnets-and-electromagnets_en.html

[2] Webassembly. Retrieved September 27, 2024 from https://webassembly.org/

[3] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and Jf Bastien. 2017. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 14, 2017. ACM, Barcelona Spain, 185–200.

[4] Yutian Yan, Tengfei Tu, Lijian Zhao, Yuchen Zhou, and Weihang Wang. 2021. Understanding the performance of webassembly applications. In *Proceedings of the 21st ACM Internet Measurement Conference* (*IMC '21*), November 02, 2021. Association for Computing Machinery, New York, NY, USA, 533–549.