Towards a Faster Web
High-performance Computing on the Browser

Matti Jokitulppo
Aalto University
School of Science
Email: matti.jokitulppo@aalto.fi

Olavi Haapala
Aalto University
School of Science
Email: olavi.haapala@aalto.fi

Mehrad Mohammadi
Aalto University
School of Science
Email: mehrad.mohammadi@aalto.fi

***Abstract*—The abstract goes here.**

I. Introduction

Until now, programmers have had little option but to use Javascript, or languages that compile down to it like Elm or PureScript when writing modern browser web applications applications. Furthermore, With the ever-increasing advent of mobile computing devices and hybrid mobile applications, fast browser-side code is more topical now than ever. However, the high-level, interpreted nature of Javascript makes performance difficult to achieve. Besides speed, modern Javascript frameworks and languages that compile down to it can also lead to large bloated applications that can take up many megabytes of bandwidth, due to Javascript being a text-based format.

Browser vendors like Firefox and Chrome have made countless attempts at optimizing the performance of JavaScript Recently, a low-level bytecode format for client-side code called WebAssembly has been standardized. WebAssembly promises to be faster to both execute and evaluate than ordinary Javascript, and also take less space. WebAssembly can also be written in various low-level languages like C, C++ or Rust. Besides WebAssembly, other means of writing performant client-side code in languages other than JavaScript includes asm.js, which is a subset of JS that C/C++ can be compiled down to, utilizing some well-supported performance optimizations of JavaScript that might be cumbersome to write by hand. The main idea behind asm.js is that its output can be directly converted into native code by browsers and their JavaScript runtimes, without going through an expensive interpretation process. In real-world usage scenarios, asm.js is often used as a fall-back for WebAssembly, if the end-user browser does not support it. We willll also look at doing calculations on the GPU with WebGL, since nowadays especially newer mobile devices contain quite capable GPUs, which are often quite underutilized in web applications. Naturally, GPUs require a totally different mindset when designing the actual algorithms since their operation differs fundamentally from CPUs, rendering them suboptimal for tasks such as graph traversal or sorting, since they cannot be parallelized easily.

Within this paper we benchmark various different algorithms implemented in plain JavaScript, the WebGL framework gpu.js, and C/C++ compiled to asm.js and WebAssembly. We then benchmark them on various different device and browser combinations. The algorithms we use are trivial and well understood, such that possible subtle differences in implementation won't affect the end result. The algorithms used are sorting an array, matrix multiplication and finding the nth Fibonacci number. As is apparent, none of these algorithms are themselves commonly used in the average web application, but they are used more as a proof-of-concept for heavy computing on the browser.

II. Experiment setup

As was previously stated, three different classic algorithms were chosen for performaning the benchmarks. They were calculating the nth number in the Fibonacci sequence, performing multiplication of two square matrices of equal sizing, and sorting a list of numbers. The matrix multiplication and list sorting were chosen due to the ease-of-scalability, since we could easily generate test data for them for arbitrary amounts of size. They also allowed us to perform benchmarks on different data types, since, for example, theoretically multiplying two 64-bit floating point numbers is much slower than doing the same for two 32-bit integers due to the way registers are laid out in modern CPUs. We theorized this would give WebAssembly a small benefit against asm.js and JavaScript, since both of them are constrained to using 64-bit floating point number types for arithmetic calculations, since it is the only number type available in JavaScript. It is worth mentioning that as specified in the EcmaScript standard [1] bitwise operators always return a 32-bit unsigned integer. This is one of the main tricks used by asm.js for potentially speeding up computation. The Fibonacci sequence was chosen because we wanted to test the possible differences between a recursive and iterative version of the algorithm, and we wanted to see how WebAssembly handles recursion compared to its alternatives, being a stack-machine based system.

For compiling C++ code to Javascript and WebAssembly, we used the Emscripten compiler project. As is stated on their website [1] Emscripten is an LLVM-based project that compiles C and C++ into highly-optimizable JavaScript in asm.js format. This lets you run C and C++ on the web at near-native speed, without plugins.". Setting up the necessary toolchain was quite painless, and after installing and setting up the compiler our

[1] http://kripken.github.io/emscripten-site/

C++ code could be compiled both to WebAssembly and asm.js with one command.

We used the popular version control hosting service GitHub for storing both build artifacts and code. GitHub also has basic file-hosting capabilities, allowing us to reach and run our benchmarks from the public internet without any additional hosting costs. Finally, we set up a simple Node.js API that we used to send our gathered measurements after every run. The results where then saved into a MongoDB database, where we could easily analyze them. An example of our data schema can be seen in listing 1. The example shows that on Chrome 62.0.3202 on Mac OS X, sorting an array of 50 floating point numbers took approximately 0.105 milliseconds.

Listing 1. BrowserStack testing code

```
{
        "_id" : ObjectId("5a1c3a5bcf7b3443d70c4b34"),
        "browser" : {
                "family" : "Chrome",
                "major" : "62",
                "minor" : "0",
                "patch" : "3202"
        },
        "device" : {
                "family" : "Other",
                "major" : "0",
                "minor" : "0",
                "patch" : "0"
        },
        "os" : {
                "family" : "Mac OS X",
                "major" : "10",
                "minor" : "13",
                "patch" : "1"
        },
        "type" : "JS",
        "data" : {
                "sorting" : 0.10500000000001819,
                "size" : 50,
                "type" : "JS"
        }
}
```

For running different benchmarks on combinations of various different devices and browsers, we set up an account on the popular browser testing service BrowserStack. BrowserStack is originally meant for writing and running end-to-end test cases for web applications, but for our purposes it fit our needs perfectly. BrowserStack offers a simple API for programmatically operating a given browser, and waiting for a certain condition to be fulfilled, so the core testing code was only 20 lines of JavaScript or so. The testing code can be in listing 2. In it, we simply iterate over the various browsers, our test cases and their various parameters and launch an asynchronous BrowserStack session for each of them. BrowserStack were also extremely generous in offering us their paid services for absolutely no cost, and we would very much like to extend our thanks for that.

Listing 2. BrowserStack testing code

```
for (const c of capabilitiesUsed) {
  for (const p of projects) {
    for (const params of p.parameters) {
      for (t of p.testsAvailable) {
        console.log(
          `Running test ${p.name} ${t} on browser ${
            c.browserName
          } with query parameters ${params}`
        );

        const url = `${baseUrl}/${p.name}/${t}/?${params}`;

        doRun(url, c);
      }
    }
  }
}
```

All in all, we chose to test on nine different browsers. For all nine, we used the latest available version on BrowserStack.

- Internet Explorer 11
- Safari on macOS
- Opera
- Edge
- Safari on iPhone
- Safari on iPad
- Android default WebView browser
- Firefox
- Chrome

III. Results

IV. Analysis

V. Conclusions

References

[1] ECMA International, *Standard ECMA-262 - ECMAScript Language Specification*, 8th ed., June 2017. [Online]. Available: https://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf