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 byte code format for client-side code called WebAssembly [1] 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 run times, 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 will 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 sub-optimal 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.

As an example, the classic recursive algorithm for finding the nth Fibonacci number, written in C++, can be seen in listing 1, The corresponding WebAssembly byte code and asm.js output, both reassembled into a human-readable format can be seen in listing 2. The main unit of both compilation, distribution and evaluation of WebAssembly is the module, which can be seen on line 1 of the compiled output. As is apparent, WebAssembly byte code is a stack-machine based abstract syntax tree of sorts. Of course, this format is not meant to be written by hand. The actual function definition for our Fibonacci function starts on line 5. As was specified in the C++ code, the function takes in a 32-bit integer and returns a 32-bit integer. On line 8, we perform the *i.32.ge_s* instruction to check if the variable on the call stack, which is the integer *n* in the C++ code, is less than or equal to one. In the WebAssembly code, the parameter *n* is located at the top of the stack. If *n* is less than two, we simply get the local variable from the stack at point *$0* and return it. Otherwise, we call the *i32.add* function on the the output of two additional calls to our Fibonacci function.

The asm.js compiled output is actually extremely similar to what one would write in regular, everyday JavaScript. The only two performance optimizations made by the compiler in this particular piece of code is the casting of all the variables into 32-bit unsigned integers by performing the bit wise OR operation on them with 0, and using addition also when decrementing numbers from each other.

From the point of view of the programmer, using both the WebAssembly and the asm.js code is quite similar. One simply loads the build artifacts from the compiler using a regular HTML5-compliant script tag, which sets a global *Module* object which can be used to call functions defined in WASM/asm.js from regular JavaScript. One important distinction is that neither WASM nor asm.js handle dynamic memory, both being stack allocated. All memory has to be allocated beforehand on the JavaScript side as Uint8Arrays of the needed size, which then

Listing 1. Nth Fibonacci (C++)

```
int fibonacci(int n) {
  if (n <= 1) {
    return n;
  }
  else {
    return fibonacci(n - 1) +
        fibonacci(n - 2);
  }
}
```

Listing 2. Nth Fibonacci (WASM)

```
(module
 (table 0 anyfunc)
 (memory $0 1)
 (export "memory" (memory $0))
 (export "_Z9fibonaccii" (func $_Z9fibonaccii))
 (func $_Z9fibonaccii (; 0 ;) (param $0 i32) (
     result i32)
  (block $label$0
   (br_if $label$0
    (i32.ge_s
     (get_local $0)
     (i32.const 2)
    )
   )
   (return
    (get_local $0)
   )
  )
  (i32.add
   (call $_Z9fibonaccii
    (i32.add
     (get_local $0)
     (i32.const -1)
    )
   )
   (call $_Z9fibonaccii
    (i32.add
     (get_local $0)
     (i32.const -2)
    )
   )
  )
 )
)
```

Listing 3. Nth Fibonacci (asm.js)

```
function fibonacci(a) {
    a = a | 0;
    var b = 0;
    if ((a | 0) < 2) return a | 0;
    else {
        b = oa(a + -1 | 0) | 0;
        return (oa(a + -2 | 0) |
            0) + b | 0;
    }
    return 0;
}
```

needs to be passed in as a pointer to the WASM/asm.js module. This can be somewhat unwieldy and verbose, especially when dealing with pointers of pointers.

II. Experiment setup

As was previously stated, three different classic algorithms were chosen for performing 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 [2] bit wise 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-optimized 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 tool chain was quite painless, and after installing and setting up the compiler our C++ code could be compiled both to WebAssembly and asm.js with one command. Unfortunately, after compiling the actual documentation on how to load, call and use functions using WebAssembly was somewhat lacking.

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. Each of our benchmarks took in its configurations such as the requested data type and number of elements as query parameters embedded straight into the URL, for ease-of-usage. 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 4. 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. A screenshot of the BrowserStack testing dashboard can be seen in figure 1

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

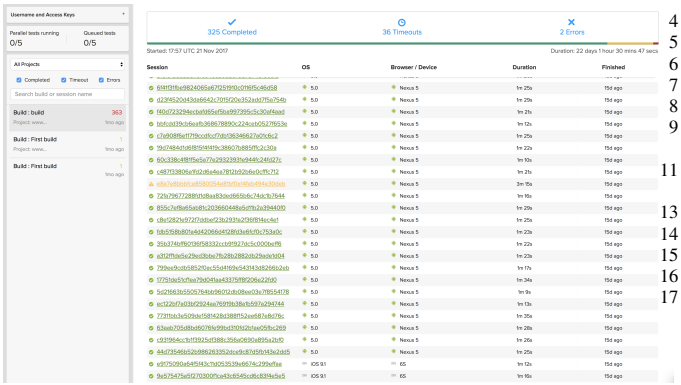

Fig. 1. Screenshot of BrowserStack dashboard

Listing 4. Measurement data example

```
{
  "_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 5. 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 5. 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 & Conclusions

For all the browsers measured, WebAssembly was indeed by far the fastest method for all three benchmarks on all browsers. However, for the sake of fairness for WebAssembly and asm.js we included not only the actual time spent on computations, but also the time spent on passing data and dynamically managing memory. This overhead made WASM and asm.js somewhat slower than native JavaScript on very trivial amounts of data.

Asm.js came in second when it came to pure speed. On some browsers, mostly Firefox, almost beating WebAssembly in some use-cases. It makes that asm.js would be the fastest of Firefox, both technologies being made by the Mozilla corporation. In some ways it feels like asm.js is already somewhat outdated technology currently. We see no reason one should use it if WebAssembly is a viable option. Asm.js can still be used as a fallback for older browsers, however, so it retains it's usefulness to that extent.

Using gpu.js to offload the easily parallelized matrix multiplication to the GPU instead of the CPU was an interesting experiment. However, as is to be expected there was a large overhead in moving data from the CPU to the GPU pipeline, and then back again. To make matters worse, WebGL has no easy way of passing simple numerical data back and forth like with regular OpenGL and the CUDA toolkit. Instead, numerical data has to be first converted into a 2D texture which must be loaded into WebGL before any operations can be performed on it, and then it must be converted back before it can be used on the web application side. The debugging tools were also very lackluster for gpu.js, and we would not consider it production-ready in its current state. It was an interesting experience none the less.

WebAssembly has recently gained support in all the latest browsers [2], and we predict that it will continue to grow as the ecosystem and tooling for it matures. During our tests, we ran into some issues of lackluster documentation, but that will not be an issue once adaption rate increases. Besides the performance implications, we feel that the breakthrough factor that will make WebAssembly succeed in the long run is the fact that one no longer has to write their web applications in JavaScript, but can choose from any number of other languages without the inevitable legacy baggage that comes with JavaScript, since the EcmaScript standard has to stay backwards compliant for the unforeseeable future. Currently, the biggest hurdle for WebAssembly is the lack of DOM operations and the lack of a standardized garbage collector, but those are currently being worked on.

REFERENCES

[1] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the web up to speed with webassembly," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2017. New York, NY, USA: ACM, 2017, pp. 185–200. [Online]. Available: http://doi.acm.org/10.1145/3062341.3062363

[2] 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

[2]https://blog.mozilla.org/blog/2017/11/13/webassembly-in-browsers/