McGill University
School of Computer Science
Sable Research Group

Using JavaScript and WebCL for Numerical Computations: A Comparative Study of Native and Web Technologies

Sable Technical Report No. sable-2014-06

Faiz Khan Vincent Foley-Bourgon Sujay Kathrotia Erick Lavoie Laurie Hendren

08-Jun-2014

Contents

List of Figures

List of Tables

Abstract

From its modest beginnings as a tool to validate forms, JavaScript is now an industrial-strength language used to power online applications such as spreadsheets, IDEs, image editors and even 3D games. Since all modern web browsers support JavaScript, it provides a medium that is both easy to distribute for developers and easy to access for users. This paper provides empirical data to answer the question: *Is JavaScript suitable for numerical computations?* By measuring and comparing the runtime performance of benchmarks representative of a wide variety of scientific applications, we show that for sequential JavaScript is within a factor of 2 of native code. Parallel code using WebCL shows speed improvements of up to 2.28 over JavaScript for the majority of the benchmarks.

1 Introduction

JavaScript was developed and released by Netscape Communications in 1995 to allow scripting in their browser[43]. By enabling embedded computations, simple tasks such as validating forms and changing the content of HTML elements could be performed in the browser without any interaction with the server. Since that time web applications have become increasingly more sophisticated and compute-intensive. To support this trend, JavaScript technology has improved to include sophisticated virtual machines and JIT compilers[18, 29], and associated technologies like WebCL[15], which provide GPU support.

Web-based JavaScript technologies provide numerous advantages to both application developers and end-users. Due to the increased standardization and support of JavaScript by all modern web browsers[7], application developers have a simple and portable mechanism for distributing their applications. Users can easily run these applications because they are likely to have modern web browsers installed on their desktops, laptops and mobile devices.

Although one might normally associate web-based computing with e-commerce applications or social networking, we want to explore if JavaScript technology is ready for a wide variety of compute-intensive numerical applications. If so, then JavaScript could be a reasonable way to develop and distribute numerical applications, either through developing JavaScript applications directly, or by using JavaScript as a target in compilers for languages such as MATLAB or R. End-users would have easy access to such applications through the wide variety of computers and mobile devices at their disposal.

In addition to examining sequential performance, we also want to evaluate the performance of parallel code using WebCL, a JavaScript binding for OpenCL.

Our work has three main contributions: (1) the development of Ostrich, an extended benchmark suite which covers a wide spectrum of numerical applications; (2) an examination of the sequential performance of JavaScript on different browsers and platforms; and (3) an examination of parallel performance using WebCL. We elaborate on each of these contributions below.

Ostrich, an extended benchmark suite: Colella and a subsequent team at Berkeley have identified important common patterns of numerical computation[16, 24], and suites such as OpenDwarfs [27] and Rodinia[22] have been developed to evaluate more conventional sequential and parallel languages. We have built upon this work to develop a benchmark suite called Ostrich, which can be used to evaluate these applications on sequential JavaScript and for most benchmarks we also provide a parallel version using WebCL.

Sequential Performance Study: We study three variants of JavaScript code: hand-written using regular arrays, hand-written using typed arrays, and compiler-generated asm.js. We compare them among themselves and give an assessment of their respective performance. We also compare hand-written JavaScript using typed arrays and asm.js with native C code and show that they are both competitive for numerical computation. Indeed, our results show that for our benchmark suite, on average, JavaScript's performance is within a factor of 2 of C. We also identify the elements of C and JavaScript that can skew the results.

Parallel Performance Study: We study the performance of parallel computation by executing our benchmarks with the WebCL and OpenCL technologies. We compare the speedups these technologies offer over their respective sequential version. We show that for a majority of benchmarks can benefit from parallel code in WebCL. We determine there is an incongruence between the two frameworks, however the problem is due to technical details in the current implementation of WebCL. The parallel tests are performed both on multi-core CPUs and on GPUs.

The rest of this paper is organized as follows: Section 2 gives a brief introduction about different technologies, including JavaScript, asm.js, and WebCL. Section 3 introduces Ostrich, our new extended benchmark suite. Section 4 describes the experimental methodology used for our performance studies, and the machine configurations used. Section 5 and Section 6 respectively discuss the performance results of sequential and parallel versions of the benchmarks. Section 7 cites the related works in the area of JavaScript performance measurement and Section 8 concludes the paper.

2 Background

In this section we provide some key background on the technologies we are using for our experiments. Our study targets the JavaScript language from a sequential perspective and a parallel perspective. On the sequential side we analyze regular JavaScript as well as an important subset of the language called asm.js. We employ a compiler that targets this subset called Emscripten. We use native C and C++ code as a baseline for our measurements. On the parallel side we compare framework for JavaScript called WebCL to its native counterpart, OpenCL. In the remainder of this section we elaborate on these technologies.[1]

2.1 JavaScript

JavaScript[7] is a high-level, multi-paradigm, language that was originally intended for browser side scripting. Recently, the browser has become a target for non-traditional applications such as document and spreadsheet editing, graphics, games, and other multimedia. This has been supported by new technologies such as HTML5[6], CSS3[2], and WebGL[14] that interface through JavaScript. Browser engines have evolved and moved from interpreters to JIT compilers. The language has also introduced byte array buffers to provide an interface for dealing with raw binary data. Typed arrays for integer and floating point numbers have also been built on top of these buffers. Since JavaScript is the only language available on the browser, many other mainstream languages such

[1]Readers familiar with these JavaScript technologies may skip this section.

as Java[5], Python[12] and Ruby[10] have compilers that target it. However, the high-level nature of the language, such as dynamic typing or prototype objects, brings the performance of heavy applications written in or compiled to JavaScript under question. Our goal is to determine the suitability of JavaScript and browser engines to support numerical applications.

2.2 Asm.js

Mozilla has developed a subset of JavaScript called asm.js[1] as a compilation target for native applications written in languages such as C and C++. It is composed of type annotations, specific program structure and low-level instruction like code. The primary advantage of programs written in this subset is that they can be directly compiled to assembly code thereby bringing near native performance to the browser. Currently, Firefox is the only browser that has an ahead-of-time compiler that compiles directly to assembly code. However, since asm.js is a subset of JavaScript, it can be executed in all browsers that support JavaScript. We are interested in determining the performance gap between asm.js code and native C and C++ code. We are also interested in the the gap between asm.js in different browsers as well as the gap between asm.js and regular JavaScript code.

2.3 Emscripten

Emscripten[45] is a compiler that translates LLVM bytecode to asm.js. Any language that is compiled to LLVM, such as C and C++ with clang, can then be used by Emscripten to generate asm.js code. Emscripten has been used successfully to compile large software such as the Unreal 4 game engine, the Bullet physics library and the Qt GUI framework[4]. The success of Emscripten makes it a suitable compiler for our investigation with asm.js.

2.4 OpenCL

OpenCL[11] is a standard by the Khronos Group describing a framework for heterogeneous parallel computing. It abstracts GPUs, multi-core CPUs and Digital Signal Processors into a single programming model. The code is expressed through a C-like language for a single thread execution, called a kernel, with added constructs such as barriers for synchronization. The data is transferred from the host to the device with the required parameters and then executed on the device. Due to the heterogeneous nature of OpenCL, computations can execute in parallel on multiple devices. As a result the highly parallel nature of devices like GPUs can be harnessed through a more general purpose C-like expression. Since it is a general framework for heterogeneous devices we use it as a native code baseline for our parallel analysis.

2.5 WebCL

WebCL[15] is a proposed standard to bring OpenCL to the web. An implementation is available from Nokia as an extension for Firefox. WebCL exposes the OpenCL API to JavaScript with certain restrictions in order to maintain security. The same kernels written for OpenCL may be used with WebCL. It is the first standard that allows for heterogeneous parallel computing in a

browser exposing CPUs, GPUs and DSPs. We use it as our target for parallel code in JavaScript because it is the web equivalent of OpenCL.

3 Ostrich - an extended benchmark suite

Although many JavaScript benchmark suites have been contributed in the past both by researchers[40] and web browser vendors[8, 9, 38], they are not necessarily representative of the upcoming classes of numerical application that could be used by scientists and engineers on the web. Notably, numerical applications are often very compute-intensive, they should leverage the parallel capabilities of modern CPUs and GPUs, and achieve as much sequential performance as possible. At the moment, different proposals for exposing parallelism to web developers [15, 31] and for increasing the speed of sequential JavaScript [1] are being actively developed. There is now a need for introducing a new benchmark suite that is tailored to numerical computations. This will enable the wider web community to assess the current performance state of web technologies and entice web browser vendors to address the bottlenecks that prevent web technologies from being fully competitive with native ones.

Selecting benchmarks is an important, yet delicate task; the benchmarks should have certain desirable properties that we list below.

1. The benchmarks should be *representative* of the core computations found in actual numerical programs;
2. The benchmarks should have sufficient *breadth* in the classes of applications that they represent to ensure that the results are generally applicable;
3. The benchmarks should produce *correct* results;
4. The results of a given benchmark written in multiple languages should be *comparable*.

The Thirteen Dwarfs, as described in Section 3.1, is a classification in thirteen groups of numerical algorithms, and are representative of most numerical tasks. By selecting from as many of these dwarfs as possible, we ensure that the benchmarks are both representative (1) and cover a wide range of applications (2). By reusing the benchmarks from existing suites such as Rodinia[22] and OpenDwarfs[27], we profit from the previous work that went into ensuring they were correct (3). Finally, we ensure that results are comparable (4) across languages by using common types and programming techniques across all implementations.

In the remainder of this section we provide the background concerning the computational patterns that we used, existing benchmark suites, and then we provide details of our new extended Ostrich benchmark suite.

3.1 Thirteen Dwarfs

Colella identified seven algorithmic patterns that occur in numerical computing that are important for the coming decade[24]. These patterns are referred to as dwarfs. At a high level, a dwarf can be thought of as an abstraction that groups together related but different computational models.

An example of a dwarf is Dense Linear Algebra that reflects dense matrix and vector computations and algorithms such as LU decomposition and matrix multiplication. Asanovic et al. expanded the the concept to algorithms in Parallel Benchmark Suites, Machine Learning, Databases, Computer Graphics and Games. They identified an additional six dwarfs that cover the majority of common algorithms in these domains. As a result, a benchmark suite that implements a majority of these dwarfs can be considered representative of a broad range of numerical computing applications. We apply this philosophy in our Ostrich benchmark suite.

3.2 Rodinia and OpenDwarfs

Rodinia[22] and OpenDwarfs[27] are two high performance benchmark computing suites for heterogeneous architectures. Both suites offer benchmarks that are classified by dwarf categories. Rodinia offers implementations of benchmarks in CUDA, OpenMP and OpenCL and OpenDwarfs has implementations in OpenCL. Rodinia is the most mature suite, but does not cover all possible dwarfs. We drew a set of benchmarks from Rodinia and the remainder from the younger OpenDwarfs suite. The map-reduce dwarf did not have an implementation in either suite, so we provide our own.

3.3 Ostrich

Our benchmark suite, Ostrich, contains 12 sequential benchmarks and 10 parallel benchmarks covering 12 dwarf categories. All but two benchmarks have five implementations: C, JavaScript, asm.js, OpenCL and WebCL.

The OpenCL implementations for the majority of benchmarks originate from the Rodinia and OpenDwarfs suites. The C and C++ code is extracted from OpenMP code when available, otherwise we provide our own implementation based on the OpenCL code. The C and OpenCL benchmarks were adapted to match the constraints of JavaScript running in the browser. When benchmarks read data from a file, we incorporated data generation into the benchmark since JavaScript running in the browser cannot access the file system. We tested various input sizes across many configurations to find values that would work in all browsers. We avoided some non-deterministic behaviors in browsers when programs start to run against the memory limitations of the virtual machines. We supplied a custom random number generator, an adaptation of the Octane benchmark suite[9] PRNG, to ensure consistent results across languages. We ensured that all the benchmarks give the same result across all languages.

In addition, we ported the benchmarks to JavaScript and WebCL by hand. The JavaScript code is written as a direct translation of the C and C++ source code, using typed arrays wherever possible. We compile the C and C++ source code to asm.js code using Emscripten. We re-use the OpenCL kernels for WebCL and mirror the C and C++ wrapper code in JavaScript.

We created an infrastructure to automatically run the experiments in a repeatable manner.

The list below describes the different dwarfs and Table I describes the specific benchmarks we have selected, and which implementations were used from an existing suite (=), created by the authors (+) and modified by the authors ($\sim$).

Benchmark	Dwarf	Provenance	Description	C	JS	OpenCL	WebCL
back-prop	Unstructured grid	Rodinia	a machine-learning algorithm that trains the weights of connecting nodes on a layered neural network	=	+	=	+
bfs	Graph traversal	Rodinia	a breadth-first search algorithm that assigns to each node of a randomly generated graph its distance from a source node	=	+	=	+
crc	Combinatorial logic	OpenDwarfs	an error-detecting code which is designed to detect errors caused by network transmission or any other accidental error	+	+	=	+
fft	Spectral methods	OpenDwarfs	the Fast Fourier Transform (FFT) function is applied to a random data set	+	+	=	
hmm	Graphical models	OpenDwarfs	a forward-backward algorithm to find the unknown parameters of a hidden Markov model	+	+	=	+
lavamd	N-body methods	Rodinia	an algorithm to calculate particle potential and relocation due to mutual forces between particles within a large 3D space	=	+	=	
lud	Dense linear algebra	Rodinia	a LU decomposition is performed on a randomly-generated matrix	∼	+	=	+
nqueens	Branch and bound	OpenDwarfs	an algorithm to compute the number of ways to put down n queens on an $n \times n$ chess board where no queens are attacking each other	+	+	=	+
nw	Dynamic programming	Rodinia	an algorithm to compute the optimal alignment of two protein sequences	∼	+	=	+
page-rank	Map reduce	Authors	the algorithm famously used by Google Search to measure the popularity of a web site	+	+	+	+
spmv	Sparse linear algebra	OpenDwarfs	an algorithm to multiply a randomly-generated sparse matrix with a randomly generated vector	+	+	=	+
srad	Structured grid	Rodinia	a diffusion method for ultrasonic and radar imaging applications based on partial differential equations	=	+	=	+

Table I: Benchmarks: provenance, description and implementation details. + indicates a new implementation, = means the algorithm was used without modification, ∼ indicates that some changes were made to the algorithm.

Dense Linear Algebra: Dense matrix and vector operations typically found in BLAS implementations belong to this dwarf. Examples: matrix multiplication, LU decomposition.

Sparse Linear Algebra: Sparse Matrices have a large number of zeros. These matrices are stored in special formats in order to improve space and time efficiency. Operations on these special formats belong to the Sparse Linear Algebra dwarf. Examples: sparse matrix multiplication.

Spectral Methods: Data is often transformed from the time or space domain into to the spectral domain. Operations involving these transformations describe the Spectral Methods dwarf. Examples: FFT.

N-body methods: Computations that deal with the interaction between multiple discrete points fit into the N-body dwarf. Examples: lavamd.

Structured grids: Data is often organized in regular structured multi-dimensional grids. Computations that proceed as a sequence of upgrade steps describe this dwarf. Examples: image manipulation.

Unstructured grids: Data can also be organized in irregular grids or meshes. In this dwarf Operations involve a grid element being updated from neighbors within an irregular grid. Examples: neural networks.

Map Reduce: Computations that run multiple independent trials and aggregate the result at the end describe this dwarf. Examples: Page Rank, distributed search.

Combinatorial logic: This dwarf covers computations that involve a large number of simple operations that exploit bit-level parallelism. Examples: hashing, cryptography.

Graph traversal: This dwarf typically involves the traversal of a number of objects and examination of their characteristics. Examples: depth-first search, breadth-first search.

Dynamic programming: Typical dynamic programming involves solving large problems by using the solutions of subproblems. Examples: DNA alignment, query optimization.

Backtrack and Branch-and-bound: Certain problems involve large search spaces. In many cases searching the entire space is infeasible. One technique is to calculate bounds on subregions in the space in order to identify areas that can be ignored. Applications involving this technique fit in to the Backtrack and Branch-and-bound category. Examples: Integer programming.

Graphical models: Computations that run on top of a graph that has random variables as nodes and probabilities as edges fit into this dwarf. Examples: Hidden Markov models.

The source code, scripts, and instructions for repeating our experiments are publicly available in a GitHub repository at `http://github.com/Sable/Ostrich/wiki`.

4 Methodology

A web browser adds another layer of virtualization between a program and the machine hardware, in addition to the operating system. The browser safely executes code from untrusted sources

alongside trusted programs on the same machine and provides a programming interface that is portable across many devices and hardware implementations. That extra layer can potentially lead to major performance degradations compared to running the same program natively. Much industrial effort in web browsers goes into ensuring the performance penalty is minimized to provide a maximum amount of hardware performance to the program.

In this paper, we quantify the overhead of web over native technologies for numerical programs using our Ostrich benchmark suite and we study the performance impact of some JavaScript language features. In this section, we identify relevant research questions and group them into sequential and parallel sets. Answers to the sequential set will provide web developers of numerical applications and compiler writers for numerical languages that target JavaScript with information on the performance currently available on the latest versions of major web browsers. Answers to the parallel set will provide a glimpse of the upcoming performance of general-purpose parallel computing using WebCL, which is still at the prototype phase and currently only supported by the Firefox browser. They should help its current proponents to focus their efforts on minimizing the performance overhead of providing additional security guarantees compared to OpenCL.

4.1 Research questions

For the sequential experiments, we study the performance of hand-written JavaScript, hand-written JavaScript using typed arrays, and machine generated asm.js code and we report how they fare against C. Typed arrays provide a mechanism for accessing raw binary data much more efficiently than with JavaScript strings. In addition to using typed arrays, asm.js code introduces specialized JavaScript operations that helps the JIT compiler to better perform type-inference and it performs manual memory management. In a sense, it is a C-like subset that provides a bytecode-equivalent target for compilers. The following sequential questions therefore help both quantify the overhead of sequential code and quantify the performance impact of some key JavaScript features for numerical computation:

- *SQ1*: Is the performance of JavaScript competitive with C?
- *SQ2*: Do typed arrays improve the performance of JavaScript code?
- *SQ3*: Does the asm.js subset offer performance improvements over hand-written JavaScript?

WebCL is the first general-purpose and low-level parallel programming language that is proposed for the web platform and has the potential to provide performance near what can be achieved with the native OpenCL technology. For the parallel experiments, we are interested in questions regarding how WebCL compares to sequential JavaScript and if WebCL's gains compare to the speedups obtained in OpenCL:

- *PQ1*: Does WebCL provide performance improvement versus sequential JavaScript?
- *PQ2*: Does WebCL provide performance improvements for JavaScript which are congruent with the performance improvements of OpenCL versus C?

In the following sections, we describe the configurations of machines and browsers we used to answer the research questions and provide details on measurements methods for repeatability.

4.2 Experimental set-up

To represent the machine that a typical web user owns, we use two consumer-grade machines, a desktop and a laptop, to perform the sequential benchmarks; the relevant hardware and software specs are detailed in Table II. For the parallel benchmarks, we use two desktop machines with different GPUs that our group uses as compute servers; their specs are detailed in Table III.

	Desktop	MacBook Air
CPU	Intel Core i7, 3.20GHz × 12	Intel Core i7, 1.8GHz × 2
Cache	12 MiB	4 MiB
Memory	16 GiB	4 GiB
OS	Ubuntu 12.04 LTS	Mac OS X 10.8.5
GCC	4.6.4	llvm-gcc 4.2
Emscripten	1.12.0	1.12.0

Table II: Specifications of the sequential machines.

The benchmarks for the sequential experiments come in four distinct configurations listed below.

- Native code compiled from C with *gcc -O3*;
- asm.js code compiled from C with *emscripten -O2* and executed in Chrome, Firefox, and Safari;
- Hand-written JavaScript using regular arrays and executed in Chrome, Firefox, and Safari;
- Hand-written JavaScript using typed arrays and executed in Chrome, Firefox, and Safari.

	Tiger	Lion
CPU	Intel Core i7 930, 2.80GHz × 8	Intel Core i7-3820, 3.60GHz × 8
Cache	8 MiB	10 MiB
GPU	NVIDIA Tesla C2050	AMD Radeon HD 7970
Memory	6 GiB	16 GiB
OS	Ubuntu 12.04 LTS	Ubuntu 12.04 LTS
GCC	4.6.3	gcc 4.6.3

Table III: Specifications of the parallel machines.

For the parallel part, the benchmarks come in two configurations:

- Native code with an OpenCL kernel;
- JavaScript code with an OpenCL kernel made available through WebCL.

All our tests are performed against version 1.0.0 of the Ostrich benchmark suite.

4.3 Measurements

The sequential, native code version compiled from C serves as the performance baseline for the sequential and parallel experiments. In the sequential case, we are interested in how JavaScript fares against typical numerical code which is most often written in languages that compile to native code. In the parallel case, we are interested in the speed-up that a parallel implementation can provide against a sequential one.

For the sequential experiments, we test hand-written JavaScript and asm.js code generated by compiling C code with Emscripten. The code is run on Firefox 29 and Chrome 35 on Linux and in Firefox 29, Chrome 35, and Safari 6.1.4 on MacOS X. All executions use only a single core on both machines.

For the parallel experiments, we compare WebCL code running in Firefox against native OpenCL code. Since Firefox is the only browser that currently has support for WebCL, it is the only JavaScript engine that we test.

Every benchmark configuration is executed 10 times on each machine and we compute the arithmetic mean of the execution times. We calculate the execution time ratio against the C version and report the geometric mean of the ratios. The execution time covers only the core algorithm of the benchmark, not any initialization or finalization processing. The execution times are measured with *gettimeofday* for C, *Date.now* for asm.js and Safari, and with the high-resolution *performance.now* for JavaScript in Firefox and Chrome.

5 Sequential Results

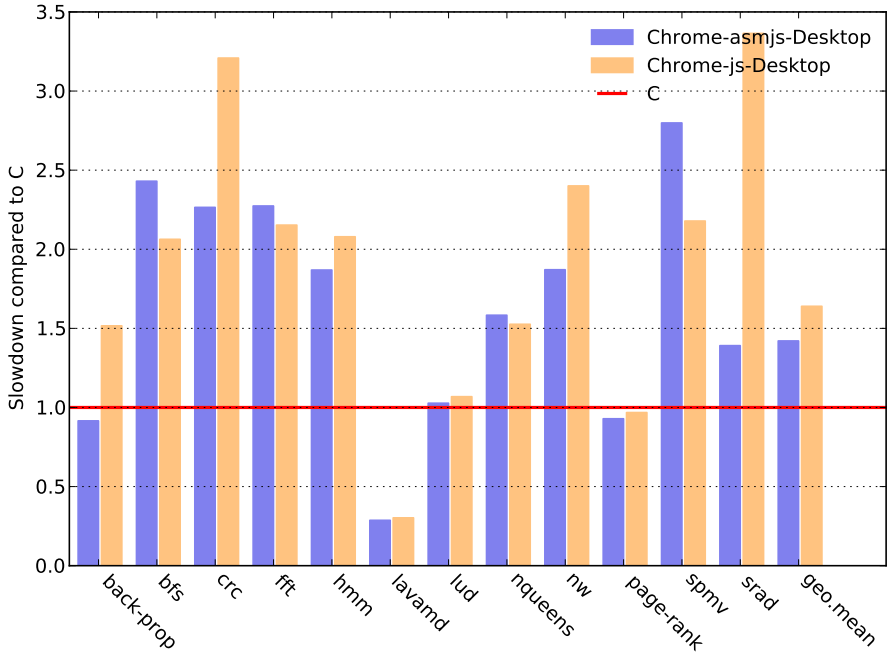
In this section, we compare the performance of different JavaScript variants between themselves and against C to answer the *SQ1*, *SQ2*, and *SQ3*.

5.1 JavaScript and asm.js vs C

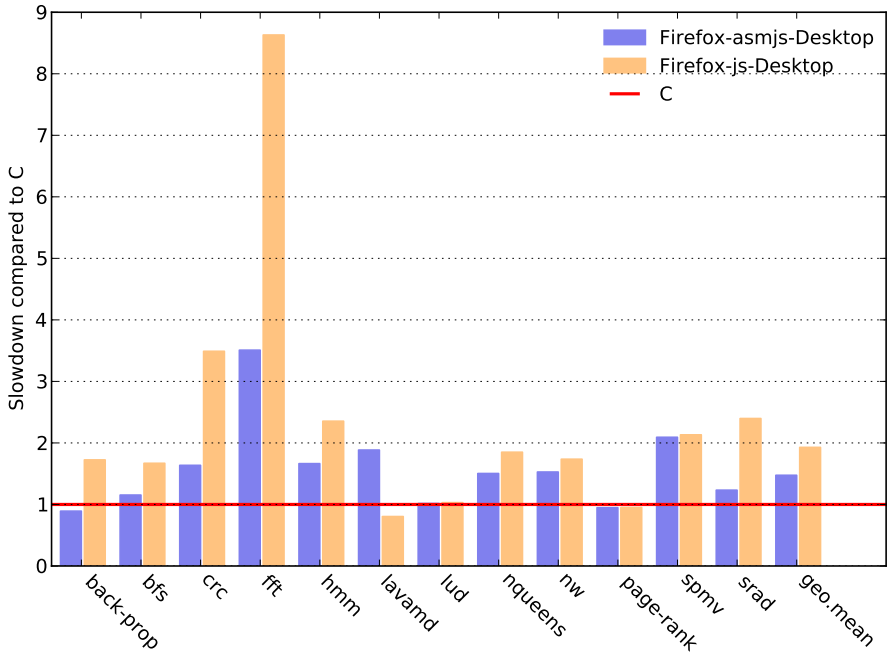
Let us start by tackling *SQ1* by comparing the performance of JavaScript and asm.js against C. This question is of particular interest because in order to extract the maximum performance out of the hardware, a large number of numerical programs are written in C, C++, and other low-level languages that compile to efficient native code.

The three contestants in this bout are hand-written JavaScript using typed arrays, asm.js generated by compiling C code with Emscripten, and C code. We start by looking at the results in Figure 1 obtained on our desktop machine using two popular browsers for Linux, Chrome and Firefox.

In Figure 1(a) for Chrome, we observe that 13 out of 24 results have a slowdown below 2x, and the slowest JavaScript program (*srad*) is only 3.36x slower than C. The most interesting benchmark, however, is definitely *lavamd* which is three times faster than C with both asm.js and JavaScript! Comparing the execution profiles of the benchmark reveals that the culprit for JavaScript's surprising performance is a call to the exponentiation function in a deeply nested loop body. An article on Google Developers explains that a faster exp() function was implemented in V8, and that it is faster than the one in the standard system libraries [13]. On average, the slowdowns with respect to C for JavaScript and asm.js are 1.6 and 1.4 respectively.



(a) Chrome's JavaScript and asm.js ratios compared to C



(b) Firefox's JavaScript and asm.js ratios compared to C

Figure 1: Slowdown of JavaScript and asm.js to C on Desktop (lower is better)

In Figure 1(b) for Firefox, 17 results show a slowdown below 2x and only one asm.js benchmark shows a slowdown significantly higher than 2. In 4 benchmarks, at least one of the JavaScript version is faster or nearly equal to C. Firefox's slowest result, *fft* is explained in the next section. On average, the slowdowns with respect to C for JavaScript and asm.js are 1.9 and 1.5 respectively.

With these results in mind, let us now see if the less powerful MacBook Air laptop gives similar ratios in Chrome and Firefox. We also look at the performance of Safari, the browser included with OS X.

In Figure 2(a) for Chrome, the slowdown of *crc* sticks out like a sore thumb. Profiling the results using Chrome shows that most of the *crc* benchmark time is spent in the core part of the calculation of the error correction code, ruling out the occurrence of an external bottleneck. Removing exception throwing in the verification of the results and removing an extra level of indirection introduced by the use of nested arrays did not lead to noticeable improvements to the running time on Chrome. The simplicity of the remaining code and the fact that the Firefox JIT is producing efficient code suggests that there might be a performance optimization opportunity that is being missed by the Chrome JIT compiler. The results for the other benchmarks are similar to the ones obtained on the desktop machine. The mean slowdowns for JavaScript and asm.js are just above 2x, 2.25 and 2.01 respectively.

For Firefox in Figure 2(b), the ratios are fairly consistent with the desktop numbers. The JavaScript *fft* shows a smaller slowdown on the MacBook Air than on the desktop, but this is because the JavaScript version takes the same time on both machines and the C version on the desktop is faster than on the laptop. The mean slowdowns for JavaScript and asm.js are both below 2x, 1.81 and 1.23 respectively.

Finally, Safari in Figure 2(c) shows average slowdowns of 2.99 and 2.35 for JavaScript and asm.js respectively. Two benchmarks, *crc* and *fft*, have a severe penalty in JavaScript, 17x and 10x. This high overhead of *crc* suggest a performance optimization that is being missed by the Safari JIT compiler.

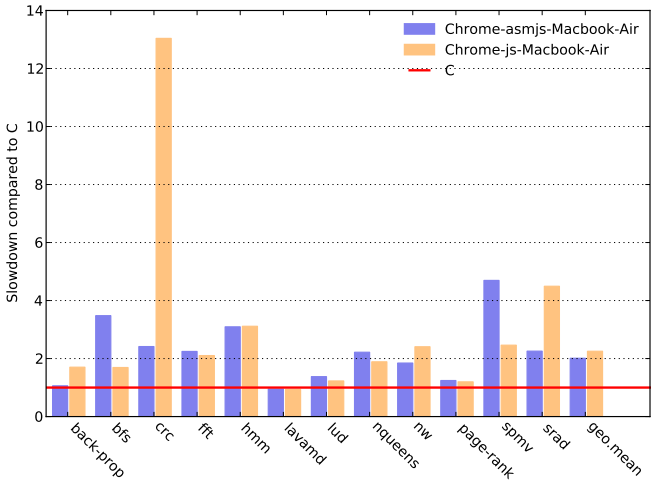
The results in this section show that, in a modern browser, the performance of JavaScript for most of the numerical benchmarks in our suite is competitive with native code.

5.2 Arrays vs Typed Arrays

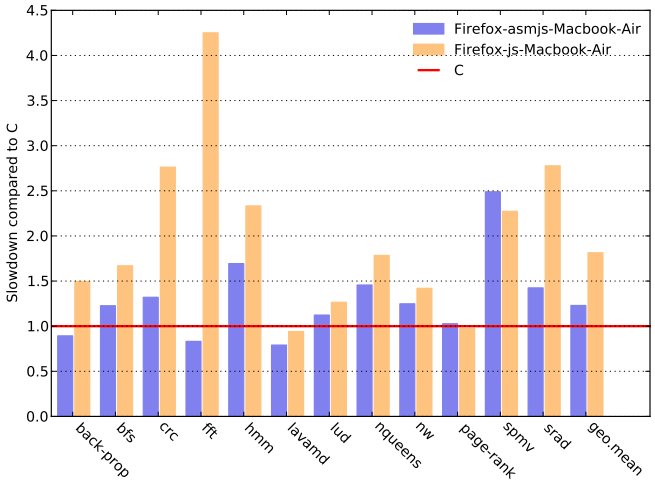
Our next experiment addresses *SQ2*. Recently, browsers have started supporting typed arrays which can contain only values of a given type, e.g. a Float32Array contains only single-precision floats. In contrast, JavaScript's regular arrays can contain values of any type, and the values need not be of the same type. In our results in the previous section we used typed arrays, and in this section we investigate if using these specialized arrays is an important factor for the good overall performance of our benchmarks.

For this experiment, we created JavaScript versions of the benchmarks where we replaced specialized arrays with regular arrays. [2] In Figure 3, we report the speedups that we observe by using typed arrays. There is a noticeable improvement for using typed arrays in most of the benchmarks, with a geometric mean speedup hovering around 2x for all browsers on both machines. In some cases, the speedup exceeds 4x and in *backprop* on Firefox on both machines, the speedup is nearly 16x!

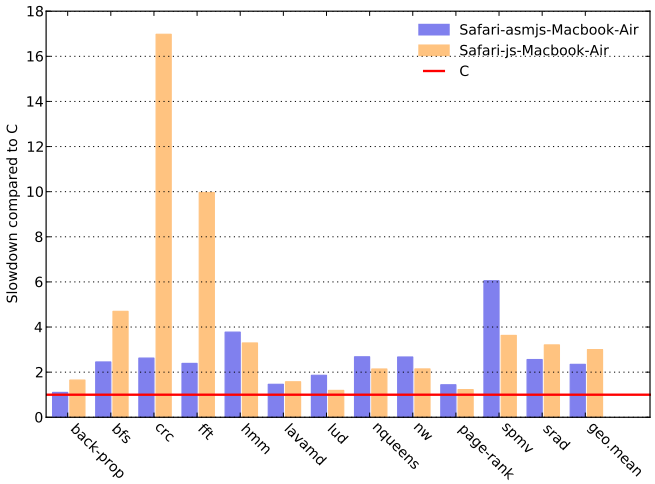
[2] Some typed arrays were kept if they were necessary to obtain correct results, e.g. when doing unsigned 32-bit integer manipulations.



(a) Chrome's JavaScript and asm.js ratios compared to C



(b) Firefox's JavaScript and asm.js ratios compared to C



(c) Safari's JavaScript and asm.js ratios compared to C

Figure 2: Slowdown of JavaScript and asm.js to C on MacBook Air (lower is better)

In one benchmark, *fft* in all browsers, the typed arrays version is slower. This is because multiple arrays are allocated inside a recursive function and for typed arrays, all the elements are set to zero, making their initialization slower than a regular array. Note that although the slowdown of Firefox is less severe than Chrome's, the execution time reveals that Chrome is faster than Firefox by a factor of 3.1.

We also notice that, but for a couple of cases (e.g. *nw* and *backprop*), the speedups obtained in Chrome and Firefox are similar. The results are consistent on the laptop both between Chrome and Firefox, and between Safari and the other two. One noticeable exception is the *fft* benchmark which is significantly slower when using typed arrays. We note that when using regular arrays on Safari, the input size of *back-prop* and *nw* had to be reduced to prevent swapping. The memory allocation pattern for Safari is non-linear and the amount of memory used by the program in these two cases would go from 1.7GB to 2.7GB for a slight change of input size, exhausting the amount of memory available in main memory on the laptop. We therefore ensured that all benchmarks ran using only main memory.

These results show that a good first step in improving the performance of a hand-written numerical JavaScript program is to replace the declarations of arrays that are manipulated in tight loops to typed arrays if possible. These results give a clear answer to *SQ2*: typed arrays improve the performance of JavaScript programs.

5.3 JavaScript vs asm.js

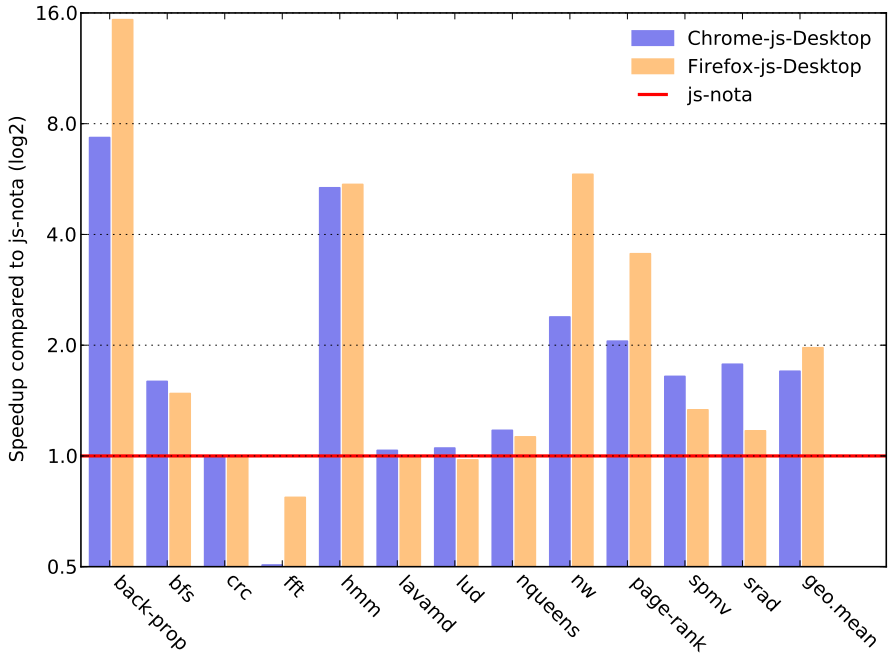
In this section, we will study whether the compiler-friendly asm.js can offer better performance than hand-written JavaScript using typed arrays.

On the desktop, the chart in Figure 4(a) shows that 11 out of 12 asm.js benchmarks in Firefox have an equal or better performance than regular JavaScript with typed arrays. In Chrome, 8 of the 12 asm.js benchmarks run equally fast or faster than the hand-written JavaScript version, and none of the slower benchmarks have a penalty as severe as *lavamd* in Firefox. The mean speedups are respectable, 1.15 for Chrome and 1.30 for Firefox.

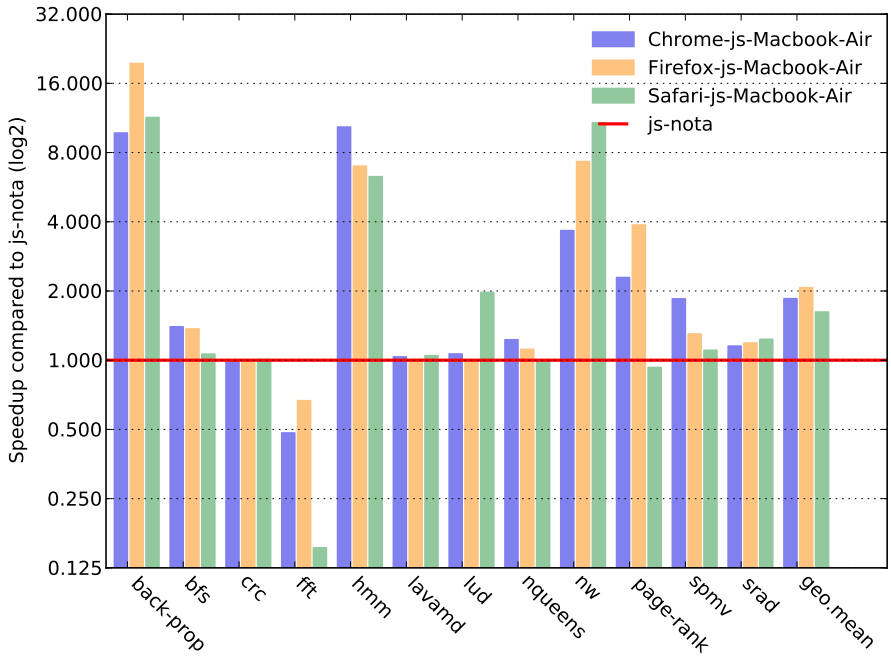
On the MacBook Air, in Figure 4(b), the range of the results is greater. The majority of the asm.js benchmarks in Firefox are still faster, but now the greatest improvement we have is 5x for *fft* and the 2x slowdown of *lavamd* has transformed into a marginal speedup. The *crc* and *fft* benchmarks in particular obtain an impressive speedup. The mean speedups for Chrome, Firefox and Safari are 1.12, 1.48, and 1.28 respectively.

Since asm.js also uses typed arrays, these results are a strong indication that the one of the most important factor in improving the performance of numerical JavaScript programs is to use typed arrays when appropriate. They also provide an answer to *SQ3*: asm.js offers performance improvements over regular JavaScript.

Let us note that although we compare their performance, JavaScript and asm.js address two different approaches to writing web applications. Asm.js is not a human-friendly format, and should be generated by a compiler; it is most useful when the developers of an application written in a language like C want to port it to the web without having to do a complete rewrite. JavaScript and typed arrays are going to be most interesting to developers who want to directly write their application for the web and want to interact with other JavaScript libraries and services.

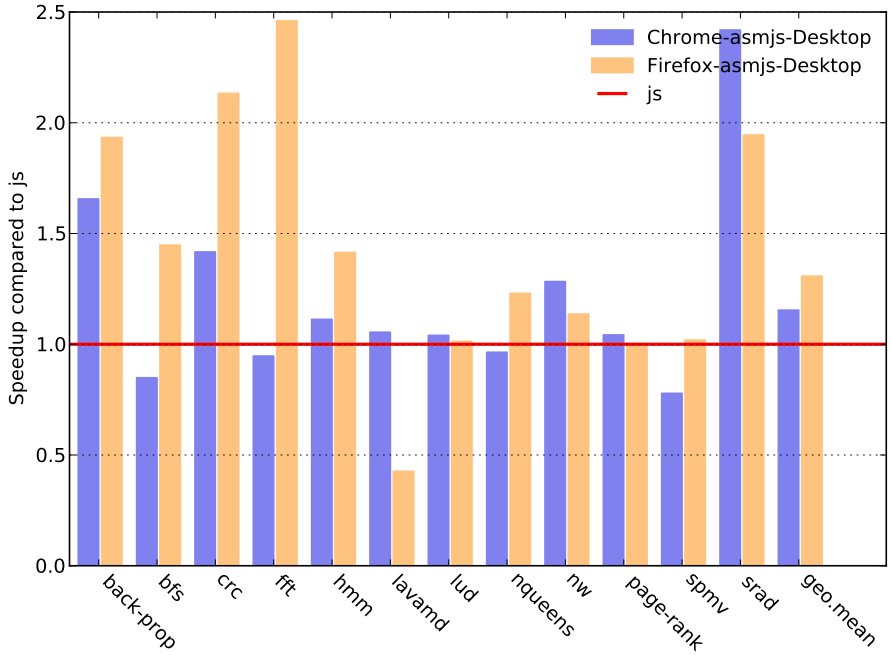


(a) Desktop

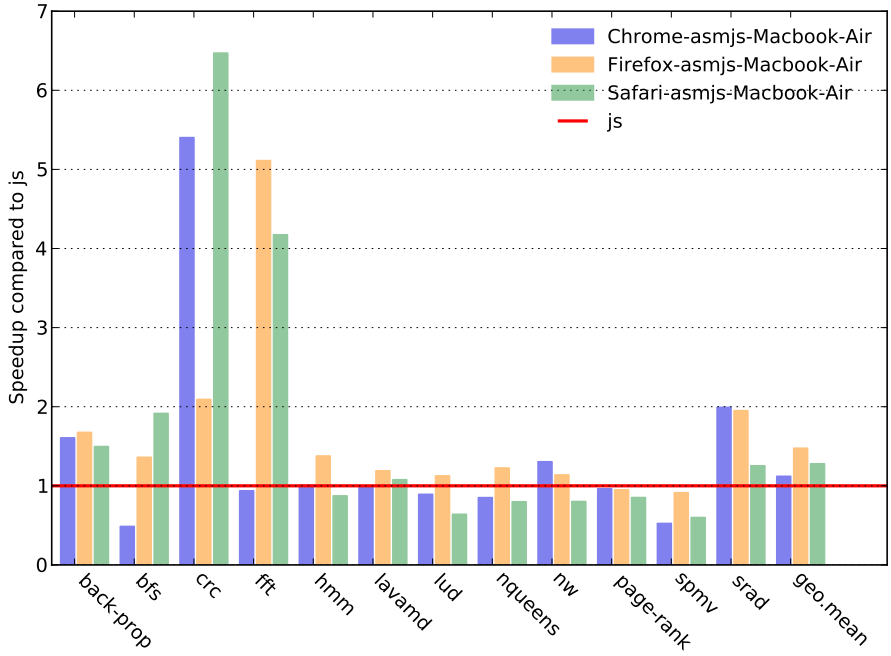


(b) MacBook Air

Figure 3: Speedup obtained by using typed arrays (js) compared to the same code without typed arrays (js-nota) (higher is better)



(a) Desktop



(b) MacBook Air

Figure 4: Speedup of asm.js against JavaScript using typed arrays (higher is better)

5.4 Other Observations

Although the results presented in the previous sections show that JavaScript's execution speed is competitive with C, there are other considerations that may need to be taken into account when using it to write numerical code.

The first point to be aware of is that not all instance sizes that can be handled by C can be handled by JavaScript. *Crc* and *bfs* are examples of benchmarks where we had to reduce the size of the input to accommodate JavaScript. In *bfs*, we initially had a graph of 4 million nodes, and it could be traversed by C, in both Chrome and Firefox with asm.js, and in Firefox with JavaScript and typed arrays. However, the execution never finished in Chrome, the process just stopped. We had to reduce the number of nodes to 3 million in order to obtain results in Chrome.

The choice of browser can also greatly affect how fast a program executes: the asm.js implementation of *lavamd* executes in 0.7 seconds in Chrome thanks to its fast exp() function, while it takes 4.4 seconds in Firefox, a slowdown of 6x.

Another point to consider when creating a numerical application for the web is how the browser reacts to long computations. In Chrome, with its one process per tab design[39], a long running computation in one tab does not affect the ability of the user to perform tasks in other tabs. In Firefox, a long running JavaScript computation will freeze the entire browser until it has finished.

5.5 Summary

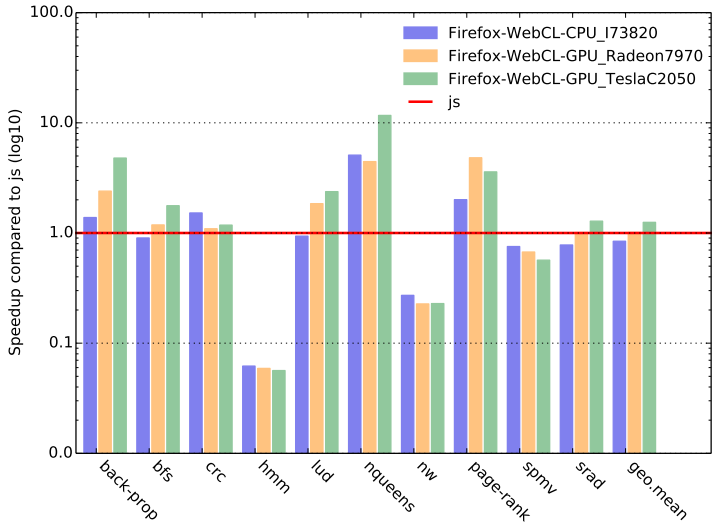
In this section, we have shown that the developer writing a numerical JavaScript application directly and the developer compiling his C application to asm.js will both obtain code that can be easily distributed to their users and is often reasonably competitive with the performance of native code (*SQ1*). We have shown that using typed arrays where applicable is an easy and effective way of improving the performance of a JavaScript application, especially if there are a lot of array accesses in the hot loops (*SQ2*). We have shown that asm.js delivers on its promise of being an efficient target for compilers; depending on the browser, it offers a speedup between 15% and 30% over hand-written JavaScript (*SQ3*).

6 Parallel Results

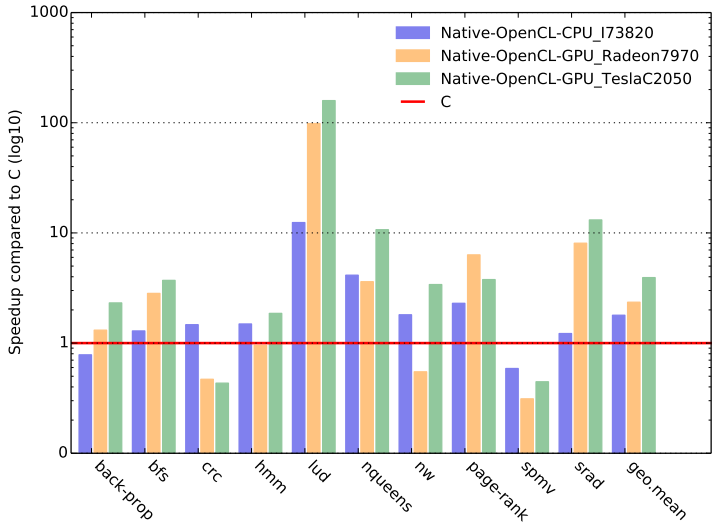
In this section we study the performance of benchmarks in WebCL. We compare how they perform against sequential JavaScript using typed arrays to answer *PQ1*. Then we answer *PQ2* by comparing the congruency of WebCL gains to OpenCL gains.

6.1 WebCL Parallel Performance Against Sequential JavaScript

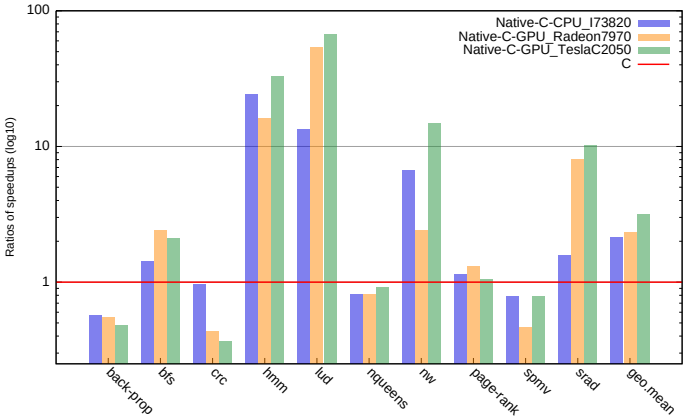
We start with answering *PQ1*. In Figure 5(a), we present the speedup of WebCL over JavaScript. The TeslaC2050 experiments were done on the Tiger machine and the i7-8820 along with the Radeon-7970 were done on the Lion machine mentioned in Section 4.2. The entire set of benchmarks shows means of 0.84, 1.01 and 1.25 for i7-3820, Radeon-7970 and Tesla-C2050, respectively. Only the Tesla platform seems to offer a performance gain on average. It seems that WebCL does not have an advantage but we note that not all algorithms benefit equally from parallel implementations.



(a) Speedup of WebCL over JavaScript (higher is better)



(b) Speedup of OpenCL over C (higher is better)



(c) WebCL-JS speedup over OpenCL-C speedup (higher is worse)

Figure 5: Parallel performance results for WebCL and OpenCL

With this in mind, we consider the effect of the slowest benchmarks and the performance gains over the most parallel benchmarks.

The two slowest benchmarks are *hmm* and *nw*. We explore the reasons for their slow performance in Section 6.2. *Hmm* is slower by a factor of 16.67 and *nw* is slower by a factor of 4.17 over all platforms. Removing these two benchmarks from our analysis yields mean speedups of 1.34, 1.74, 2.28 for i7-3820, Radeon-7970 and Tesla-C2050, respectively. With the majority of benchmarks offering speedups, there is an observable advantage in having a parallel platform such as WebCL.

The benchmarks that are most improved by using WebCL are *nqueens*, *page-rank* and *back-prop*. The *nqueens* kernel is highly optimized thus yields significant gains. *Page-rank* is a map-reduce problem that is embarrassingly parallel, therefore it performs better on a parallel platform. Similarly, *Back-prop* also falls on a parallel side as weights between the neural network layers can be calculated independently and is reflected in the speedup. The mean speedup of these three benchmarks are 2.42, 3.72 and 5.85 for i7-3820, Radeon-7970 and Tesla-C2050, respectively. It is clear that certain benchmarks are better suited for WebCL over others.

It is not surprising that there is a spectrum of performance gains from parallel code since some code is more parallel in nature than other. With respect to *PQ1* we see that there are significant gains that can be made with WebCL, albeit they are dependant on the application.

6.2 WebCL Specific Performance

If we observe Figure 5(b), we see the speedups offered by OpenCL over C. The average OpenCL speedups are 1.79, 1.99 and 3.93 for i7-3820, Radeon-7970 and Tesla-C2050, respectively. If we remove the two slowest benchmarks, *hmm* and *nw*, the speedups are 1.83, 2.56 and 4.40. The best performers are *back-prop*, *lud*, *nqueens*, and *page-rank*; the average speedup on i7-3820, Radeon-7970 and Tesla-C2050, for these four benchmarks, are 3.10, 7.36 and 11.03 respectively.

Figure 5(c) presents the ratio of the two speedups: the speedup of OpenCL over C against the speedup of WebCL over JavaScript. This offers us a perspective on how the gains realized in OpenCL carry over to WebCL. A higher bar means that OpenCL provided a greater performance improvement than WebCL. The WebCL gains over JavaScript were 2.50 times lower than the gains OpenCL realized over C. We find this to be interesting because although WebCL is a binding to OpenCL there seems to be excessive overhead somewhere in between.

Figure 5(c) shows that half of the benchmarks suffer significant execution penalties in WebCL. These benchmarks are *bfs*, *hmm*, *lud*, *nw* and *srad*. Compared to the OpenCL implementations, these benchmarks have mean slowdowns of 1.93, 23.42, 36.16, 6.20 and 5.04, respectively. To find the source of these slowdowns, we profiled the JavaScript code that wraps the kernels. Our investigation revealed that six functions were taking up most of the execution time in the code for the worst performing benchmarks.

Table IV shows the name and total run time percentage for the most time consuming functions on each of the five benchmarks[3]. The first function, setArg, points the kernel to the address in memory for a specific parameter to the kernel. The _validateKernel function is an internal function that

[3]The names in Table IV are truncated for readability. The setArg function is part of the WebCLKernel object. the device and queue getInfo functions are in the WebCLDevice and WebCLCommandQueue object prototyes, respectively. The read and write buffer functions are properly termed enqueueReadBuffer and enqueueWriteBuffer in the WebCLCommandQueue object prototype _validateKernel is also in the WebCLCommandQueue object prototype.

ensures that the kernel is a valid object, compiled and associated with the right context. The device and queue getInfo functions are information query functions for a specific device and the command queue working on a device. The buffer read and write functions are responsible for reading from and writing to device memory from the host.

Function	bfs	hmm	lud	nw	srad
setArg	5.6%	45.7%	34.2%	41.9%	15.0%
_validateKernel	16.7%	21.0%	25.6%	22.1%	41.1%
device.getInfo	-	-	5.4%	3.1%	6.5%
queue.getInfo	-	-	2.4%	2.8%	6.1%
readBuffer	11.2%	7.6%	-	-	12.5%
writeBuffer	33.3%	7.6%	-	-	-
Total	66.7%	82.0%	67.7%	69.9%	81.2%

Table IV: Percentage of WebCL execution spent on setup functions (results that are close to zero are marked as "-")

Host and device memory transfers are a known bottleneck in parallel applications, and we will examine them separately, and focus solely on the setArg, _validateKernel, and getInfo functions. The mean of the execution time percentage of these four functions is 56.39% when we include *bfs*, a significant outlier, and 68.26% without *bfs*. Examining the setArg and _validateKernel functions show that in addition to executing the OpenCL routines, they call slow query functions. These slow functions are mainly used for setting up the kernels and could be improved through caching or other optimizations.

Now we address the memory transfer operations, readBuffer and writeBuffer. Looking at Table IV we notice that *bfs*, *hmm* and *srad* are the only benchmarks that took significant time in transferring data. We profiled the equivalent functions in the OpenCL version of the three benchmarks and we calculated the ratio of the WebCL memory transfer time against the OpenCL memory transfer time. The time spent on transferring memory in *bfs* was close to zero. The *hmm* time for the readBuffer function was on average 67.8 times slower in WebCL than in OpenCL for all platforms, and the writeBuffer operation was 93.45 times slower. *Srad* only had slowdowns in the readBuffer operation and we saw that it was 12.41 times slower in WebCL. The WebCL functions are simple bindings to OpenCL therefore there are significant bottlenecks in the extension or the browser that are causing these slowdowns. We were not able to determine these values since the Firefox profiler does not allow for traversal into functions past the JavaScript API for an extension.

With regards to *PQ2* we state that currently the gains from WebCL are not congruent with the gains from OpenCL. The WebCL benchmarks showed an overall loss of 2.50. Half of the benchmarks had significant slowdowns on the WebCL side and most of this was due to setup code. This, however, may be addressable through optimizations or a direct implementation in the browser.

6.3 Other Performance Observations

Figure 5(c) shows that three benchmarks have better gains in WebCL than OpenCL: *back-prop*, *crc* and *spmv*. Looking back at Figure 5(a) and Figure 5(b) we see that *back-prop* actually executes faster in WebCL than in OpenCL.

Profiling reveals that the *back-prop* implementation slowdown occurs in the sequential portion of

the code. The original OpenCL implementation of the benchmark uses a two-dimensional representation of matrices, and has to convert it to a one-dimensional array before passing it to the kernel. In WebCL, we use a typed array, which are always one-dimensional, to represent the matrix, and the conversion was not necessary. The actual kernel execution time was similar.

In *crc* the WebCL execution time is faster than JavaScript, yet OpenCL is slower than C. This is a case where JavaScript is much slower than C, and using WebCL to compute the answer is faster. In such cases, it is beneficial to use WebCL even when it isn't beneficial to use OpenCL.

A similar event occurs in *spmv*. In Figure 5(a) and Figure 5(b), we see that *spmv* had worse performance for the parallel code for both WebCL and OpenCL than their sequential counterparts. Although Figure 5(c) seems to indicate that the WebCL gains are better, in reality WebCL is incurring less of a loss than OpenCL. This happens because the gap between JavaScript and C is wider than between WebCL and OpenCL.

6.4 WebCL Limits and Caveats

In this section, we report some observations that we obtained in our experiences with WebCL.

The WebCL specification does not allow for structs, but the same is not true for OpenCL. This imposes a limit on the ease of expression of complex data types in kernels for WebCL. Our *lavamd* benchmark makes liberal use of nested structs and would require a large number of arrays and careful indexing to convert to WebCL. This method would change the nature of the memory access patterns in our data structures and would skew our performance results.

As mentioned in the previous sections, a large portion of the slowdown occurs in the setup code in the WebCL extensions. The effect of this setup code diminishes for very large data sets. Massive data set sizes are a common occurrence in numerical computations. Although WebCL itself is capable of handling large data sets, the same is not true for the browser. The browser memory limitations forced us to reduce the sizes of inputs that we tested. As an example, even though OpenCL can handle 200 million non-zero elements on *spmv*, the browser often crashed on our machines with the same instance size for WebCL.

OpenCL allows for data to be copied to device memory with an offset. However, no offset can be specified for the host memory. In C, this is fixed with a simple increment of the pointer that is being copied. In JavaScript, a new sub-array view object on the typed array or a new typed array must be created in order to achieve the same result.

6.5 Summary

In this section, we have shown that WebCL can offer performance gains over JavaScript when the problem is a good parallel candidate (*PQ1*). We have also shown that currently, the performance gains of WebCL over JavaScript are 2.5x less than the gains of OpenCL over C (*PQ2*).

Overall, we find that WebCL does offer advantages as an extension to JavaScript. It shows that a parallel JavaScript framework can aid in numerical computations, particularly those that are amenable to parallelization. However, the inefficiencies of the current implementation do not make the gains over sequential JavaScript as important as OpenCL does with C.

7 Related Work

There has been a surge of interest towards numerical computing in the programming language community recently, with the coming of age of newer languages, such as Julia[19] and the study and improvement of older alternatives, such as R[37] and MATLAB[21]. The increasing and widely available parallelism in hardware, plus the establishment of the Web as a major development and deployment platform, means the ground is fertile for work that helps bring the available performance and convenience of the platform to users of numerical languages. Our work in this paper focused on the comparative study of native and web technologies performance for numerical computing using a benchmark suite inspired by the dwarf categories[16, 27]. In this section we situate our work by discussing contributions from the larger context of understanding and improving the performance characteristics of the Web platform for numerical computing.

A number of benchmark suites have been proposed by vendors over the years to measure the performance of browsers, such as Apple's JetStream[38] (previously SunSpider), Google's Octane[9] (previously V8 benchmarks), and Mozilla's Kraken[8] suites. These benchmark suites are all compute-intensive and their 2010 versions were shown not to be representative of real interactive web applications[41]. This has sparked initiatives to build infrastructure to automatically extract benchmarks from real web applications[42], which has been used to produce a suite of reference applications called JSBench[40]. To the best of our knowledge, however, there is no comprehensive web benchmark suite for the upcoming numerical applications typical of engineering and scientific activities, such as the Rodinia suite[22], which is tailored for multicore CPU and GPU computation. Our collection of benchmark implementations for both sequential (JavaScript) and parallel (WebCL) computation aim to answer that need by taking the Rodinia and OpenDwarfs suites and adding new web implementations.

The advent of multiple numerical computing libraries built entirely in JavaScript, such as *math.js*[25], *sylvester*[23], and *numeric.js*[34] shows the need for numerical computing abstractions in web applications. In addition, there have been numerous initiatives to leverage available parallelism in today's hardware, by extending JavaScript implementations without breaking the sequential semantics. Sluice provides JavaScript with a library that can dynamically recompile programs to target multicores for high performance stream processing[28]. ParaScript provides automatic runtime parallelization that is fully transparent to the programmer by optimistically identifying code regions for parallelization, generating on-the-fly parallel code, and speculatively executing it[35]. River Trail[31] extends the array datatype with functional operations that automatically leverage multicores and GPUs for execution. We believe our benchmarks could be adapted in a straightforward manner to compare the relative merits of these and similar approaches, and we hope that they will be useful in that context.

JavaScript is increasingly used as a compilation target and runtime system. Examples include Racket with the Whalesong compiler[44], Dart with dart2js[3], and SML with SML2JS[26]. Krahn and al. also introduced JavaScript-specific optimization techniques for dealing with modularization constructs with ContextJS [32]. When it is used as a target, compilers need to be aware of the underlying implementation to extract maximum performance as shown by Auler et al.[17]. In addition to performance-oriented contributions, the recent formalization of the semantics of the major parts of JavaScript standard[20] is paving the way for high-assurance translations of other languages to JavaScript with machine-checked implementation. One of the aims of our work is to shed light on the suitability of JavaScript and associated web technologies to serve as a competitive compilation target for numerical languages. Superconductor[36] has shown the suitability of WebCL, WebGL

and web workers for large scale data visualization. In complement, our work provides quantitative measurements for the relative merits of both sequential (JavaScript) and parallel (WebCL) computation in the context of numerical computation on the web.

Web browsers are complicated pieces of software and multiple factors make performance evaluation non-trivial. A study of the performance variations in the Mozilla Firefox web browser[33] has shown that external factors to the browser, in particular memory randomization done by the operating system, are the dominant factors that explain noise in performance measurements. Grosskurth and Godfrey[30] introduced a reference architecture for web browsers in 2005, which identify and explain the major sub-system in web browsers such as Firefox. Chrome, a more recent architecture, was presented and studied later in a paper by Reis and Gribble[39]. Our work has shown that the current performance of web technologies is competitive with native technologies on many benchmarks typical of numerical computation. Pinpointing the bottlenecks with regard to the exact services in web browsers will be addressed in future work.

8 Conclusions and future work

In this paper, we have presented our benchmark suite, Ostrich, which draws benchmarks from two reputable sources, Rodinia and OpenDwarfs, and adds JavaScript and WebCL implementations. Ostrich provides implementations for numerical algorithms in C, JavaScript, asm.js, OpenCL and WebCL for twelve of the thirteen dwarfs.

Using the benchmarks in this suite, we have shown that the performance of both hand-written JavaScript (using typed arrays) and asm.js is competitive (within a factor of 2) with C, in a large number of benchmarks (*SQ1*). We have shown that to achieve this performance, the use of typed arrays is important. Compared to regular JavaScript arrays, typed arrays provide nearly double the performance in all browsers (*SQ2*). We have also shown that the asm.js subset can give additional peformance improvements over just using typed arrays. Our experiments showed that the asm.js versions run 15% to 30% faster than hand-written JavaScript (using typed arrays), depending on the browser (*SQ3*). These results should help web developers of numerical applications and compiler writers for numerical languages that target JavaScript in choosing languages features to use and target.

We have also shown that there are significant gains from parallel code for a majority of benchmarks. The fastest three benchmarks were around 5.85 times faster than JavaScript on GPUs (*PQ1*). We found that the WebCL code speedup over JavaScript had significant losses when compared to the OpenCL speedup over C. We attribute this to technical bottlenecks in the current implementation. Therefore, we find that WebCL gains are incongruent with OpenCL gains but the issue is not inherent to the WebCL specification (*PQ2*). These results provide a glimpse of the upcoming performance of general-purpose parallel computing using WebCL, which is still at the prototype phase and currently only supported by the Firefox browser. They should help its current proponents to focus their efforts on minimizing the performance overhead of providing additional security guarantees compared to OpenCL. Our benchmark suite will help to evaluate the performance evolution of the technology.

In addition to extending the suite with more benchmarks, one interesting possibility of extension for our work is to port the benchmarks to other numerical languages, such as MATLAB and R, to evaluate the performance of the generated JavaScript code produced by JavaScript compilers for

these languages.

A Sequential time data

In this appendix, we present the timing results from our experiments. The tables below show the average times and standard deviation (in seconds) for 10 execution of each benchmark in each software configuration.[4]

A.1 Sequential results

Benchmark	Cr, asm.js	Cr, JS	Cr, JS-noTA	Fx, asm.js	Fx, JS	Fx, JS-noTA	C
back-prop	0.8200 ± 0.01	1.3575 ± 0.01	9.9587 ± 0.15	0.7988 ± 0.01	1.5443 ± 0.01	23.6966 ± 0.23	0.8953 ± 0.01
bfs	0.7903 ± 0.03	0.6703 ± 0.01	1.0689 ± 0.01	0.3751 ± 0.01	0.5426 ± 0.00	0.8020 ± 0.02	0.3249 ± 0.01
crc	1.4220 ± 0.01	2.0132 ± 0.02	2.0152 ± 0.01	1.0275 ± 0.01	2.1897 ± 0.02	2.1833 ± 0.03	0.6274 ± 0.01
fft	1.7744 ± 0.03	1.6794 ± 0.01	0.8487 ± 0.04	2.7374 ± 0.02	6.7326 ± 0.07	5.1950 ± 0.05	0.7801 ± 0.01
hmm	3.3623 ± 0.04	3.7307 ± 0.35	19.9860 ± 0.75	2.9866 ± 0.03	4.2239 ± 0.04	23.1092 ± 0.20	1.7943 ± 0.03
lavamd	0.6938 ± 0.01	0.7315 ± 0.01	0.7575 ± 0.03	4.5569 ± 0.20	1.9439 ± 0.02	1.9406 ± 0.02	2.4171 ± 0.03
lud	1.9848 ± 0.04	2.0641 ± 0.04	2.1688 ± 0.03	1.9646 ± 0.02	1.9899 ± 0.03	1.9412 ± 0.02	1.9319 ± 0.03
nqueens	4.6285 ± 0.19	4.4565 ± 0.06	5.2315 ± 0.06	4.3920 ± 0.14	5.4017 ± 0.06	6.0873 ± 0.06	2.9202 ± 0.02
nw	0.8442 ± 0.00	1.0823 ± 0.01	2.5831 ± 0.02	0.6893 ± 0.01	0.7828 ± 0.01	4.5624 ± 0.30	0.4509 ± 0.00
page-rank	3.1273 ± 0.03	3.2587 ± 0.04	6.6846 ± 0.03	3.1920 ± 0.03	3.2033 ± 0.05	11.3570 ± 0.16	3.3659 ± 0.03
spmv	1.8655 ± 0.06	1.4522 ± 0.02	2.3894 ± 0.02	1.3953 ± 0.03	1.4212 ± 0.02	1.8970 ± 0.02	0.6666 ± 0.02

Table V: Average times in seconds on Desktop.

Benchmark	Cr, asm.js	Cr, JS	Cr, JS-noTA	Fx, asm.js	Fx, JS	Fx, JS-noTA	C
back-prop	1.3973 ± 0.02	2.1145 ± 0.11	1.1245 ± 0.01	1.8846 ± 0.02	1.2450 ± 0.03	1.3846 ± 0.01	2.0876 ± 0.03
bfs	1.5619 ± 0.01	0.8852 ± 0.06	0.5991 ± 0.03	0.8304 ± 0.05	0.5052 ± 0.03	1.2359 ± 0.03	2.3826 ± 0.06
crc	2.5125 ± 0.01	13.5556 ± 0.22	1.3787 ± 0.01	2.8748 ± 0.01	1.0461 ± 0.02	2.7447 ± 0.06	17.6367 ± 0.09
fft	3.5470 ± 0.03	3.3230 ± 0.02	1.3286 ± 0.01	6.7281 ± 0.10	1.5803 ± 0.03	3.7691 ± 0.01	15.7503 ± 0.11
hmm	6.7838 ± 0.03	6.8985 ± 0.03	4.6138 ± 0.14	5.1601 ± 0.02	2.2067 ± 0.01	8.3414 ± 0.01	7.2849 ± 0.02
lavamd	1.4546 ± 0.01	1.4294 ± 0.01	1.2078 ± 0.00	1.4339 ± 0.01	1.5130 ± 0.01	2.1969 ± 0.01	2.5016 ± 0.01
lud	3.8714 ± 0.09	3.5079 ± 0.04	3.5932 ± 0.08	3.1437 ± 0.14	2.7698 ± 0.07	5.5382 ± 0.07	3.3465 ± 0.08
nqueens	8.6880 ± 0.05	7.4113 ± 0.03	5.6962 ± 0.12	7.0104 ± 0.09	3.9163 ± 0.01	10.5524 ± 0.02	8.3635 ± 0.04
nw	1.3191 ± 0.01	1.7191 ± 0.01	0.8454 ± 0.03	1.0114 ± 0.03	0.6958 ± 0.01	1.9286 ± 0.03	1.4817 ± 0.04
page-rank	4.8429 ± 0.01	4.6529 ± 0.01	3.9266 ± 0.11	3.8034 ± 0.01	3.8657 ± 0.14	5.6164 ± 0.01	4.7816 ± 0.02
spmv	3.6317 ± 0.03	1.9281 ± 0.06	1.9482 ± 0.01	1.7706 ± 0.01	0.7782 ± 0.01	4.6554 ± 0.01	2.8205 ± 0.01
srad	9.9713 ± 0.41	19.8897 ± 0.11	6.3674 ± 0.02	12.2547 ± 0.03	4.3593 ± 0.04	11.1628 ± 0.01	14.0219 ± 0.05

Benchmark	Safari, asm.js	Safari, JS	Safari, JS-noTA
back-prop	1.2450 ± 0.03	1.3846 ± 0.01	2.0876 ± 0.03
bfs	0.5052 ± 0.03	1.2359 ± 0.03	2.3826 ± 0.06
crc	1.0461 ± 0.02	2.7447 ± 0.06	17.6367 ± 0.09
fft	1.5803 ± 0.03	3.7691 ± 0.01	15.7503 ± 0.11
hmm	2.2067 ± 0.01	8.3414 ± 0.01	7.2849 ± 0.02
lavamd	1.5130 ± 0.01	2.1969 ± 0.01	2.5016 ± 0.01
lud	2.7698 ± 0.07	5.5382 ± 0.07	3.3465 ± 0.08
nqueens	3.9163 ± 0.01	10.5524 ± 0.02	8.3635 ± 0.04
nw	0.6958 ± 0.01	1.9286 ± 0.03	1.4817 ± 0.04
page-rank	3.8657 ± 0.14	5.6164 ± 0.01	4.7816 ± 0.02
spmv	0.7782 ± 0.01	4.6554 ± 0.01	2.8205 ± 0.01
srad	4.3593 ± 0.04	11.1628 ± 0.01	14.0219 ± 0.05

Table VI: Average times in seconds on MacBook Air

[4]Cr = Chrome; Fx = Firefox; JS = JavaScript with typed arrays; JS-noTA = JavaScript without typed arrays

A.2 Parallel results

Benchmark	Firefox, WebCL	Firefox, JS	C	OpenCL
back-prop	0.6291 ± 0.00	1.5101 ± 0.01	0.8960 ± 0.00	0.6836 ± 0.00
bfs	0.4780 ± 0.00	0.5660 ± 0.00	0.3366 ± 0.00	0.1191 ± 0.01
crc	2.0794 ± 0.01	2.2725 ± 0.01	0.6488 ± 0.00	1.3817 ± 0.03
hmm	73.1601 ± 1.11	4.3188 ± 0.02	1.8377 ± 0.01	1.9245 ± 0.03
lud	1.1124 ± 0.02	2.0545 ± 0.02	1.9959 ± 0.01	0.0203 ± 0.01
nqueens	1.2740 ± 0.01	5.6656 ± 0.25	3.0197 ± 0.01	0.8374 ± 0.01
nw	3.5725 ± 0.02	0.8114 ± 0.02	0.4689 ± 0.00	0.8548 ± 0.01
page-rank	0.6734 ± 0.01	3.2443 ± 0.02	3.4556 ± 0.00	0.5468 ± 0.02
spmv	2.1439 ± 0.01	1.4410 ± 0.04	0.6965 ± 0.02	2.2333 ± 0.01
srad	9.2527 ± 0.06	9.3437 ± 0.50	3.9523 ± 0.01	0.4900 ± 0.02

Table VII: Average times in seconds on AMD Thaiti

Benchmark	Firefox, WebCL	Firefox, JS	C	OpenCL
back-prop	0.3778 ± 0.00	1.8092 ± 0.01	1.2399 ± 0.00	0.5355 ± 0.01
bfs	0.3891 ± 0.02	0.6880 ± 0.00	0.4517 ± 0.00	0.1218 ± 0.00
crc	2.8803 ± 0.17	3.3968 ± 0.19	0.7003 ± 0.00	1.6178 ± 0.01
hmm	94.8387 ± 0.64	5.3518 ± 0.01	2.4236 ± 0.00	1.3022 ± 0.01
lud	1.1191 ± 0.01	2.6567 ± 0.09	2.5930 ± 0.01	0.0163 ± 0.00
nqueens	0.7489 ± 0.01	8.7431 ± 0.68	4.7604 ± 0.02	0.4453 ± 0.00
nw	4.1874 ± 0.18	0.9583 ± 0.05	0.5634 ± 0.00	0.1658 ± 0.02
page-rank	1.1375 ± 0.01	4.0802 ± 0.01	3.7118 ± 0.00	0.9841 ± 0.02
spmv	3.3228 ± 0.04	1.8842 ± 0.04	1.1153 ± 0.01	2.5034 ± 0.01
srad	10.3852 ± 0.07	13.3040 ± 0.59	4.6794 ± 0.01	0.3567 ± 0.01

Table VIII: Average times in seconds on Nvidia Tesla

Benchmark	Firefox, WebCL	Firefox, JS	C	OpenCL
back-prop	1.0896 ± 0.01	1.5059 ± 0.00	0.8960 ± 0.00	1.1451 ± 0.01
bfs	0.6295 ± 0.01	0.5675 ± 0.00	0.3362 ± 0.00	0.2608 ± 0.00
crc	1.4927 ± 0.01	2.2696 ± 0.00	0.6485 ± 0.00	0.4415 ± 0.03
hmm	69.6483 ± 0.45	4.3153 ± 0.02	1.8389 ± 0.02	1.2330 ± 0.02
lud	2.2132 ± 0.02	2.0722 ± 0.06	1.9963 ± 0.01	0.1607 ± 0.00
nqueens	1.0982 ± 0.02	5.5920 ± 0.07	3.0190 ± 0.01	0.7303 ± 0.00
nw	3.0066 ± 0.02	0.8169 ± 0.02	0.4681 ± 0.00	0.2591 ± 0.00
page-rank	1.6185 ± 0.02	3.2474 ± 0.01	3.4555 ± 0.00	1.5042 ± 0.02
spmv	1.9155 ± 0.02	1.4439 ± 0.06	0.6632 ± 0.00	1.1275 ± 0.02
srad	12.2432 ± 0.09	9.5318 ± 0.03	3.9490 ± 0.01	3.2337 ± 0.05

Table IX: Average times in seconds on Intel i7

References

[1] asm.js. URL `http://asmjs.org/spec/latest/`.

[2] Cascading style sheets (CSS) snapshot 2010. URL `http://www.w3.org/TR/css-2010/`.

[3] dart2js. URL `https://www.dartlang.org/tools/dart2js`.

[4] Emscripten. URL `http://emscripten.org`.

[5] Google web toolkit. URL `http://www.gwtproject.org/`.

[6] A vocabulary and associated APIs for HTML and XHTML. URL `http://www.w3.org/TR/html5/`.

[7] Standard ECMA-262 ECMAScript language specification edition 5.1 (june 2011). URL http://www.ecma-international.org/publications/standards/Ecma-262.htm.

[8] Kraken Benchmark Suite. URL http://krakenbenchmark.mozilla.org/.

[9] Octane Benchmark Suite. URL https://developers.google.com/octane/.

[10] Opal: Ruby to JavaScript compiler. URL http://opalrb.org/.

[11] OpenCL. URL https://www.khronos.org/opencl/.

[12] Pyjs. URL http://pyjs.org/.

[13] Profile your web application with v8's internal profiler. URL https://developers.google.com/v8/profiler_example.

[14] WebGL specifications. URL http://www.khronos.org/registry/webgl/specs/latest/.

[15] WebCL specification. May 2014. URL http://www.khronos.org/registry/webcl/specs/latest/1.0/.

[16] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and Others. The landscape of parallel computing research: A view from Berkeley. Technical report, Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[17] R. Auler, E. Borin, P. de Halleux, Moskal, and N. Tillmann. Addressing JavaScript JIT engines performance quirks: A crowdsourced adaptive compiler. URL http://research.microsoft.com/en-us/um/people/moskal/pdf/cc2014.pdf.

[18] L. Bak. Google chrome's need for speed. URL http://blog.chromium.org/2008/09/google-chromes-need-for-speed_02.html.

[19] J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman. Julia: A Fast Dynamic Language for Technical Computing, Sept. 2012. URL http://arxiv.org/abs/1209.5145.

[20] M. Bodin, A. Charguéraud, D. Filaretti, P. Gardner, S. Maffeis, D. Naudziuniene, A. Schmitt, and G. Smith. A Trusted Mechanised JavaSript Specification. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 87–100, New York, NY, USA, 2014. ACM. doi: 10.1145/2535838.2535876. URL http://dx.doi.org/10.1145/2535838.2535876.

[21] A. Casey, J. Li, J. Doherty, M. C. Boisvert, T. Aslam, A. Dubrau, N. Lameed, A. Aslam, R. Garg, S. Radpour, O. S. Belanger, L. Hendren, and C. Verbrugge. McLab: An Extensible Compiler Toolkit for MATLAB and Related Languages. In *Proceedings of the Third C* Conference on Computer Science and Software Engineering*, C3S2E '10, pages 114–117, New York, NY, USA, 2010. ACM. doi: 10.1145/1822327.1822343. URL http://dx.doi.org/10.1145/1822327.1822343.

[22] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54. IEEE, Oct. 2009. doi: 10.1109/iiswc.2009.5306797. URL http://dx.doi.org/10.1109/iiswc.2009.5306797.

[23] J. Coglan. Sylvester: Vector and Matrix math for JavaScript.

[24] P. Colella. Defining software requirements for scientific computing, 2004.

[25] J. de Jong. math.js. URL http://mathjs.org/.

[26] M. Elsman. SMLtoJs: Hosting a standard ML compiler in a web browser. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Programming Language and Systems Technologies for Internet Clients*, PLASTIC '11, pages 39–48, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-1171-7. doi: 10.1145/2093328.2093336. URL http://dx.doi.org/10.1145/2093328.2093336.

[27] W. C. Feng, H. Lin, T. Scogland, and J. Zhang. OpenCL and the 13 Dwarfs: A Work in Progress. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, ICPE '12, pages 291–294, New York, NY, USA, 2012. ACM. doi: 10.1145/2188286.2188341. URL http://dx.doi.org/10.1145/2188286.2188341.

[28] J. Fifield and D. Grunwald. A Methodology for Fine-Grained Parallelism in JavaScript Applications. In S. Rajopadhye and M. Mills Strout, editors, *Languages and Compilers for Parallel Computing*, volume 7146 of *Lecture Notes in Computer Science*, pages 16–30. Springer Berlin Heidelberg, 2013. doi: 10.1007/978-3-642-36036-7_2. URL http://dx.doi.org/10.1007/978-3-642-36036-7_2.

[29] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based just-in-time type specialization for dynamic languages. *SIGPLAN Not.*, 44(6):465–478, June 2009. ISSN 0362-1340. doi: 10.1145/1542476.1542528. URL http://dx.doi.org/10.1145/1542476.1542528.

[30] A. Grosskurth and M. W. Godfrey. A Reference Architecture for Web Browsers. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, ICSM '05, pages 661–664, Washington, DC, USA, 2005. IEEE Computer Society. doi: 10.1109/icsm.2005.13. URL http://dx.doi.org/10.1109/icsm.2005.13.

[31] S. Herhut, R. L. Hudson, T. Shpeisman, and J. Sreeram. River Trail: A Path to Parallelism in JavaScript. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 729–744, New York, NY, USA, 2013. ACM. doi: 10.1145/2509136.2509516. URL http://dx.doi.org/10.1145/2509136.2509516.

[32] R. Krahn, J. Lincke, and R. Hirschfeld. Efficient Layer Activation in Context JS. In *Creating, Connecting and Collaborating through Computing (C5), 2012 10th International Conference on*, pages 76–83. IEEE, Jan. 2012. doi: 10.1109/c5.2012.20. URL http://dx.doi.org/10.1109/c5.2012.20.

[33] J. Larres, A. Potanin, and Y. Hirose. A Study of Performance Variations in the Mozilla Firefox Web Browser. In *Proceedings of the Thirty-Sixth Australasian Computer Science Conference - Volume 135*, ACSC '13, pages 3–12, Darlinghurst, Australia, Australia, 2013. Australian Computer Society, Inc. URL http://portal.acm.org/citation.cfm?id=2525401.2525402.

[34] S. Loisel. Numeric Javascript. URL http://numericjs.com/.

[35] M. Mehrara, P.-C. Hsu, M. Samadi, and S. Mahlke. Dynamic parallelization of JavaScript applications using an ultra-lightweight speculation mechanism. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 87–98. IEEE, Feb. 2011. ISBN 978-1-4244-9432-3. doi: 10.1109/hpca.2011.5749719. URL http://dx.doi.org/10.1109/hpca.2011.5749719.

[36] L. A. Meyerovich, M. E. Torok, E. Atkinson, and R. Bodık. Superconductor: A Language for Big Data Visualization, 2013.

[37] F. Morandat, B. Hill, L. Osvald, and J. Vitek. Evaluating the Design of the R Language: Objects and Functions for Data Analysis. In *Proceedings of the 26th European Conference on Object-Oriented Programming*, ECOOP'12, pages 104–131, Berlin, Heidelberg, 2012. Springer-Verlag. doi: 10.1007/978-3-642-31057-7_6. URL http://dx.doi.org/10.1007/978-3-642-31057-7_6.

[38] F. Pizlo. JetStream Benchmark Suite. URL https://www.webkit.org/blog/3418/introducing-the-jetstream-benchmark-suite/.

[39] C. Reis and S. D. Gribble. Isolating Web Programs in Modern Browser Architectures. In *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09, pages 219–232, New York, NY, USA, 2009. ACM. doi: 10.1145/1519065.1519090. URL http://dx.doi.org/10.1145/1519065.1519090.

[40] G. Richards. JSBench Benchmark Suite. URL http://jsbench.cs.purdue.edu/.

[41] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An Analysis of the Dynamic Behavior of JavaScript Programs. *SIGPLAN Not.*, 45(6):1–12, June 2010. doi: 10.1145/1809028.1806598. URL http://dx.doi.org/10.1145/1809028.1806598.

[42] G. Richards, A. Gal, B. Eich, and J. Vitek. Automated Construction of JavaScript Benchmarks. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 677–694, New York, NY, USA, 2011. ACM. doi: 10.1145/2048066.2048119. URL http://dx.doi.org/10.1145/2048066.2048119.

[43] C. Severance. JavaScript: Designing a language in 10 days. *Computer*, 45(2):7–8, Feb. 2012. ISSN 0018-9162. doi: 10.1109/mc.2012.57. URL http://dx.doi.org/10.1109/mc.2012.57.

[44] D. Yoo and S. Krishnamurthi. Whalesong: Running Racket in the Browser. *SIGPLAN Not.*, 49(2):97–108, Oct. 2013. doi: 10.1145/2578856.2508172. URL http://dx.doi.org/10.1145/2578856.2508172.

[45] A. Zakai. Emscripten: An LLVM-to-JavaScript compiler. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, SPLASH '11, pages 301–312, New York, NY, USA, 2011. ACM. doi: 10.1145/2048147.2048224. URL http://doi.acm.org/10.1145/2048147.2048224.