# McNumJS: A JavaScript Library for Numerical Computations

*Sujay Kathrotia*

School of Computer Science
McGill University
Montréal, Canada

April 2015

---

A thesis submitted to the Faculty of Graduate Studies in partial fulfillment of the requirements for the degree of Master of Science.

# Dedication

*Dedicated to my supportive parents and kindhearted sisters, for their unconditional and unceasing love, belief and support. Also, to the entire research community, who with their hard work, make this world a better place!*

# Abstract

There has been a huge development in the web community recently, with an increasing focus on the performance of JavaScript. The development of state-of-the-art JavaScript engines and JavaScript technologies has improved the performance of JavaScript considerably and made it competitive with other dynamic languages. The major advantage of JavaScript applications is that they can run on any device that supports web browsers and distribution of these applications is very easy. This thesis reports on McNumJS, an easy-to-use and high-performance JavaScript library for numerical computations. This library is helpful to JavaScript developers for developing numerical applications and compiler writers who want to compile scientific languages like MATLAB or R to JavaScript.

There has been a surge of technologies like typed arrays, web workers and asm.js, developed to improve the performance of JavaScript. We analyze these technologies and report their suitability for numerical applications. We have also compiled a detailed study on asm.js and performed different experiments to find the parts of asm.js that we can use in regular development of JavaScript applications.

There are two main design goals behind the development of McNumJS: i) making it easy-to-use, and ii) provide high-performance. We achieved the easy-to-use goal by making an API similar to the NumPy, a popular python library for scientific computing. To make McNumJS high-performance, we used JavaScript typed arrays and type coercing rules defined by asm.js. We report the speedups we get by using McNumJS compared to other JavaScript libraries and JavaScript with regular arrays. We report the performance difference between McNumJS and native C. These experiments show that the performance of McNumJS library is competitive with native C and outperforms other JavaScript libraries for numerical computations.

# Résumé

Il y a eu de fortes avancées dans la communauté du web récemment, avec un focus particulier sur la performance de JavaScript. Le développement des machines virtuelles de dernière génération et des technologies associées ont considérablement amélioré la performance de JavaScript et l'ont rendu compétitif avec les autres langages dynamiques en vogue. Le principal avantage des applications web est qu'elles peuvent être exécutées sur n'importe quel appareil avec un navigateur web. De plus, leur déploiement est particulière aisé. Ce mémoire présente McNumJS, une librairie de calcul haute-performance et simple d'utilisation pour JavaScript. Cette librairie est utile à la fois pour les développeurs d'applications JavaScript et les développeurs de compilateurs pour des langages scientifiques, tel MATLAB et R, qui ciblent JavaScript.

De nombreuses technologies pour JavaScript sont apparues récemment, comme les tableaux typés (Typed Arrays), les sous-processus de calcul (Web Workers), et la définition d'un sous-ensemble du langage facilement optimisable (asm.js). Ces technologies sont étudiées et une analyse de leur applicabilité aux applications numériques est présentée. Une étude détaillée sur le sous-ensemble asm.js est présentée ainsi qu'une étude empirique de performance qui a permis de déterminer les parties de ce sous-ensemble qui peuvent efficacement être utilisée dans le développement d'applications JavaScript conventionnelles.

Deux critères principaux ont motivé le développement de la librairie McNumJS: i) la facilité d'utilisation et ii) la performance à l'exécution. La facilité d'utilisation provient d'une interface similaire à NumPy, une librairie de calcul scientifique populaire et éprouvée pour le language Python. L'objectif de performance fut atteint en utilisant une combinaison de tableaux typés ainsi que les annotations de typage définies par le sous-ensemble asm.js. La performance obtenue par l'utilisation de McNumJS est comparée à

celle des autres librairies de calcul numérique pour JavaScript. Elle est également comparée à la performance en utilisant directement JavaScript avec des tableaux réguliers. Finalement, la performance de la librairie utilisée dans un navigateur web est comparée à une version C des mêmes programmes de test. Ces expériences montrent que la librairie McNumJS est compétitive en performance avec des technologies natives et offre une performance supérieure aux autres librairies de calcul numérique pour JavaScript.

# Acknowledgements

Foremost, I would like to express my heartfelt gratitude to my adviser, Prof. Laurie Hendren for her patience, motivation, enthusiasm, and profound knowledge. Her continuous guidance and feedback helped me to not only improve the quality of my research and but also my writing and presentation skills.

I would like to thank MCJS team for putting up with me, for developing Ostrich benchmark suite and writing paper on JavaScript performance. It provided me the foundation for my performance analysis. I am thankful to Rahul Garg for his valuable inputs and suggestions. I would also like to thank Ismail Bidawi for searching names with me for my library. Many thanks to Erick Lavoie and Vincent Foley for translating my thesis's abstract to French. Thanks to Sridipta Misra for proof-reading my thesis.

At Last, I am thankful to the Sable Lab for all the facilities and resources that they provided to make my learning and research effective and interesting. Many thanks to CS Helpdesk for providing tech support whenever I needed.

# Contents

**Abstract** ii

**Résumé** iii

**Acknowledgement** v

**Chapter 1: Introduction** 1

1.1 Motivation . . . . . 1

1.2 Thesis Contributions . . . . . 2

1.3 Thesis Organization . . . . . 3

**Chapter 2: Background and Related Technologies** 4

2.1 JavaScript Features and Technologies . . . . . 4

2.1.1 Regular arrays . . . . . 4

2.1.2 Typed arrays . . . . . 6

2.1.3 Asm.js . . . . . 8

2.1.4 Web Workers . . . . . 11

2.1.5 WebCL . . . . . 14

2.2 Related Technologies . . . . . 16

2.2.1 NumericJS . . . . . 17

2.2.2 Google Closure Library . . . . . 17

2.2.3 Sylvester . . . . . 18

2.2.4 NumPy . . . . . 18

**Chapter 3: Performance Analysis of asm.js** 19

3.1 Overview of asm.js Specifications . . . . . 19

3.2 Asm.js Experiments . . . 20
3.2.1 Normal JavaScript . . . 20
3.2.2 Function with Type Coercion . . . 22
3.2.3 Function with Type Coercion in Strict Mode . . . 22
3.2.4 Calling asm.js Function from Normal JavaScript . . . 23
3.2.5 Calls between Different asm.js Modules . . . 23
3.2.6 Complete asm.js Module . . . 25
3.3 Results . . . 25

**Chapter 4: McNumJS - A JavaScript Library for Numerical Computations 29**
4.1 JavaScript Features and Technology Selection . . . 29
4.2 Architecture . . . 31
4.2.1 Core Module . . . 32
4.2.2 Generation Module . . . 40
4.2.3 Unary Operations Module . . . 43
4.2.4 Binary Operations Module . . . 45
4.3 Development Process . . . 45

**Chapter 5: Performance Results 47**
5.1 Methodology . . . 47
5.1.1 Measurement objectives . . . 47
5.1.2 Experimental setup . . . 48
5.1.3 Measurements . . . 49
5.2 Ostrich Results . . . 51
5.2.1 McNumJS vs C . . . 51
5.2.2 McNumJS vs JavaScript Typed Arrays . . . 53
5.2.3 McNumJS vs Asm.js . . . 55
5.2.4 McNumJS vs One dimensional Regular Arrays . . . 55
5.3 Performance compared to other libraries . . . 58

**Chapter 6: Conclusions and Future Work 62**
6.1 Conclusion . . . 62
6.2 Future Work . . . 63

**Appendix A: Ostrich Benchmark Suite Results** **66**
A.0.1 Execution times . . . . . 66

**Appendix B: Micro-benchmarks Results** **68**

**References** **71**

# List of Figures

2.1 Architecture of JavaScript Typed Arrays . . . . . 7

4.1 Architecture of JavaScript Typed Arrays . . . . . 41
4.2 Development Process of McNumJS . . . . . 46

5.1 Slowdown of McNumJS compared to C . . . . . 52
5.2 Slowdown of McNumJS compared to JavaScript Typed Arrays . . . . . 54
5.3 Slowdown of McNumJS compared to asm.js . . . . . 56
5.4 Speedup of McNumJS compared to One dimensional Regular Arrays . . 57
5.5 Slowdown of JavaScript libraries compared to McNumJS in Chrome (V = Vector, M = Matrix, S = Scalar) . . . . . 60
5.6 Slowdown of JavaScript libraries compared to McNumJS in Firefox . . . 61

# List of Tables

3.1 Machine specifications . . . . . 27
3.2 Asm.js results . . . . . 28

4.1 Typed Array alias in McNumJS . . . . . 33
4.2 TypedArray constructor parameters in McNumJS . . . . . 34
4.3 TypedArray constructor parameters in McNumJS . . . . . 40

5.1 Machine specifications . . . . . 48

5.2 Ostrich Benchmarks . . . . . 50
5.3 Micro-Benchmarks . . . . . 51

A.1 Ostrich benchmark suite results on C and Firefox . . . . . 66
A.2 Ostrich benchmark suite results on Chrome . . . . . 67

B.1 Micro-benchmark results on Firefox . . . . . 69
B.2 Micro-benchmark results on Chrome . . . . . 70

# List of listings


1 Sepia filter using regular arrays . . . . . 5
2 Sepia filter using typed arrays . . . . . 7
3 Sepia filter using asm.js module . . . . . 9
4 Sepia filter using Web Workers . . . . . 12
5 Sepia filter using Nokia WebCL on Mozilla Firefox . . . . . 15
6 Kernal for Sepia filter using Nokia WebCL . . . . . 16
7 Normal JavaScript . . . . . 21
8 Normal JavaScript with type coercion . . . . . 21
9 Type coercion in strict mode . . . . . 22
10 Add method in asm.js calling from Normal JavaScript . . . . . 23
11 Add method in asm.js calling from different asm.js module . . . . . 24
12 Add method in asm.js . . . . . 26
13 Changing Uint16Array Constructor . . . . . 32
14 Uint16Array Constructor in McNumJS . . . . . 33
15 Example of strides . . . . . 36
16 Example of get method for 2-dimensional array . . . . . 37
17 Example of methods added to typed array class . . . . . 39
18 Generation module functions syntax and examples . . . . . 42
19 Unary operations module functions syntax . . . . . 43
20 Example of transpose . . . . . 44
21 Binary operations module functions syntax . . . . . 45

# Chapter 1

# Introduction

## 1.1 Motivation

JavaScript was developed and released by NetScape Communications in 1995 as a non-professional scripting language to support small client-side computations like validating forms and changing HTML data [1]. JavaScript has evolved a lot since then. The language has become so widespread these days that it has powered server-side as well as desktop applications [2, 3]. Due to increased standardization and support [4], it has become possible to run same code on a wide variety of devices. This is advantageous to both developers and users as developers don't need write separate programs for different configurations and users can run these programs on different devices.

With the availability of sophisticated virtual machines and JIT compilers for JavaScript [5, 6], the application of JavaScript has evolved from simpler to more complex applications like 3D games, image editing, signal processing, data visualization etc., which were previously reserved for classical programing languages [7]. These applications are highly compute-intensive. So features like typed arrays, web workers, and technologies like asm.js [8], WebGL [9] and WebCL [10] have been developed to improve the performance of JavaScript. These technologies have been discussed in details in Chapter 2.

However, for developers and scientists, it is non-trivial to work with these technologies. Some developers stick to the classical programming languages as they provide significant performance and easy-to-use APIs while others use JavaScript without these technologies and hence receive with slower performance. We aim to solve this prob-

lem by creating a library which uses these technologies and exposes familiar numerical API like NumPy [11] to the developers. Developers can use this API 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 creating a library for numerical computation in JavaScript, other contributions of our work are: (1) analysis of JavaScript technologies; (2) performance evaluation of asm.js; (3) guide to coerce type information like asm.js; and (4) cost assessment of different numerical operations in JavaScript. We elaborate on these contributions below.

## 1.2 Thesis Contributions

The main contribution of our work is:

**McNumJS Library:** We have developed a JavaScript library McNumJS, which provides familiar API functions like NumPy to the developers for numerical computation. The library uses recent JavaScript features like typed arrays to improve the performance. McNumJS also uses coercing type information by accessing typed arrays or providing type annotations to the variables like asm.js so that JavaScript engines can better optimize the code. By developing this library, we aim to ease development of numerical programs.

Apart from the major contribution of developing library, our work also contributes in the following ways:

**Analysis of JavaScript technologies:** Our study provides detailed information on what JavaScript technologies are available which can provide speedup for numerical computations. We list these technologies along with their advantages and disadvantages. This study can equip developers to select optimal technologies for the development of their application in JavaScript.

**Performance evaluation of asm.js:** We study the performance of asm.js, a low-level and efficient subset of JavaScript with different configurations like asm.js code with and without "use asm" directive, asm.js code communicating with other subset of JavaScript and asm.js code within other subset of JavaScript code. We evaluate the performance results of these configuration.

**Guide to coerce type information like asm.js:** Asm.js is defined by a static type system. Compiler writers can easily use this system to produce target JavaScript code but it is not trivial to write asm.js modules in normal JavaScript code. However, it is still possible to coerce type information in most of the code with ease and allow JavaScript engines to better optimize the code. Our work will provide direction on how to coerce type information in the regular development of JavaScript applications.

**Cost assessment of matrix operations:** We analyzed time complexity of some common matrix operations in terms of big O notation. This study will set a benchmark for numerical application writers and allow them to minimize time complexity of their applications.

## 1.3 Thesis Organization

This thesis comprises seven chapters. Chapter 2 gives brief introduction about different JavaScript features, technologies and libraries available for numerical computation. Chapter 3 provides detailed performance analysis of asm.js with different configurations. Chapter 4 describes the development and architecture of the McNumJS library and provides details about API functions. Chapter 4 also provides information about time complexity of some matrix operations. Chapter 5 discusses the performance results of the JavaScript library with respect to other popular libraries. Chapter 6 concludes the thesis and discusses possible future work.

# Chapter 2

# Background and Related Technologies

In this chapter, we discuss different JavaScript features and technologies which we can use for numerical computation. Analysing these technologies is important as our goal is to create fast and easy-to-use JavaScript library for numerical computations. This chapter also touches on related work of JavaScript libraries for numerical computations. We also discuss the popular numerical library NumPy for Python.

## 2.1 JavaScript Features and Technologies

The following subsections lists JavaScript features and technologies. Each subsection starts with a brief introduction of the feature or technology followed by an example. We will briefly discuss about the example and then provide advantages and disadvantages of the feature or the technology.

### 2.1.1 Regular arrays

JavaScript arrays are high-level, list-like objects to store multiple values. JavaScript arrays have neither fixed length nor fixed type of elements. Thus, JavaScript arrays can grow or shrink dynamically and have any value. If we set a value at the index, which is outside the current bounds of the array, JavaScript engines update the array's length automatically. Similarly, if we dynamically decrease the length value of an array, it will delete the remaining elements from the array. JavaScript arrays are zero indexed.

The Array prototype object provides methods to traverse, filter, sort, map, and reduce operations on array elements.

```
function sepia(image, width, height) {
    for(var i=0; i < height; i++) {
        for(var j=0; j < width; j++) {
            var r = image[i][j][0];
            var g = image[i][j][1];
            var b = image[i][j][2];

            var sr = r * 0.393 + g * 0.769 + b * 0.189;
            var sg = r * 0.349 + g * 0.686 + b * 0.168;
            var sb = r * 0.272 + g * 0.534 + b * 0.131;

            image[i][j][0] = sr > 255 ? 255 : sr;
            image[i][j][1] = sg > 255 ? 255 : sg;
            image[i][j][2] = sb > 255 ? 255 : sb;
        }
    }
}
```

**Listing 1** Sepia filter using regular arrays

Listing 1 is the function which applies a sepia filter to an image represented by a 3-dimensional JavaScript array. You can notice that we can easily access the elements of multi-dimensional arrays by just providing indexing of each dimension in square braces. JavaScript stores this as array of arrays. You can also notice that as regular arrays can contain any data type, we have to manually make sure that the Red, Green and Blue values do not exceed 255.

**Advantages of Regular arrays are:**

- Regular arrays can grow or shrink dynamically.
- We can add any type of data in arrays which makes multi-dimension array indexing easy.

- Creating strided arrays are easy by exploiting above two advantages. Strided arrays are arrays in which only some of the elements are present. This can save memory by not storing the empty elements in between.

**Disadvantages of Regular arrays are:**

- Getting and setting value at given index of an array requires extra steps of checking bound conditions as well as maintaining the length property. This makes regular arrays slower to access.
- Operations like slicing of an array or transposing matrix in-place is not possible without copying or manipulating array elements.

### 2.1.2 Typed arrays

JavaScript typed arrays are array-like objects and provide a mechanism for accessing raw binary data. Unlike regular JavaScript arrays, typed arrays cannot grow dynamically.

The architecture of typed arrays comprises two main parts: array buffers and views. Figure 2.1 represents the architecture of the JavaScript typed arrays. ArrayBuffer object represents the chunk of data. ArrayBuffer object does not have any method to access or manipulate data. In order to use data contained in buffer, we need to use views (either TypedArrayView or DataView). Views provide context like data type, starting offset, number of elements, etc. to the array buffers.

Listing 2 is the function to apply sepia filter to an image, which is stored using Uint8ClampedArray. Typed arrays are only one dimensional arrays. Thus, if we want to use multi-dimensional indexing, we need to manually map it with one dimensional index (refer line 4 of Listing 2). As Uint8ClampedArray guarantees that the data will not exceed one byte, we won't have to check for the pixel color values exceeding 255.

**Advantages of Typed arrays are:**

- Typed arrays provide mechanism to work with binary data which makes easier to manipulate audio, video and images.

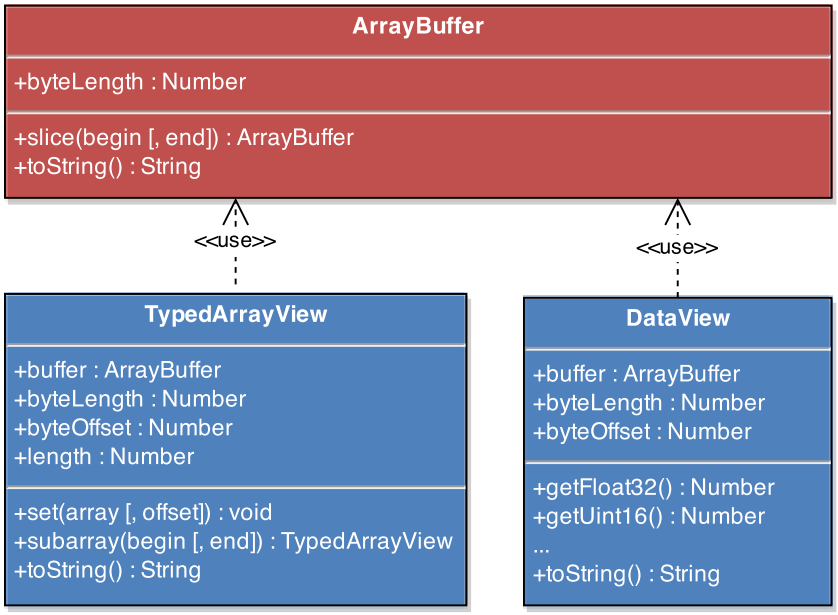


**Fig. 2.1** Architecture of JavaScript Typed Arrays

```
function sepia(image, width, height) {
    for(var i=0; i < height; i++) {
        for(var j=0; j < width; j++) {
            var offset = (i*width+j)*4;
            var r = image[offset];
            var g = image[offset+1];
            var b = image[offset+2];

            image[offset] = r * 0.393 + g * 0.769 + b * 0.189;
            image[offset+1] = r * 0.349 + g * 0.686 + b * 0.168;
            image[offset+2] = r * 0.272 + g * 0.534 + b * 0.131;
        }
    }
}
```

**Listing 2** Sepia filter using typed arrays

- Typed arrays provide element types for data contained in them. Without typed arrays, there is just one type of double-like representation for numbers.
- Array buffers are contiguous chunk of memory. This provides faster access and manipulation of the data than regular arrays.
- Having a separate view for the buffer enables slicing of the array without actually copying or manipulating entire array.

**Disadvantages of Typed arrays are:**

- Initialization of typed array takes more time than regular arrays as initialization allocates memory block as well as fill that memory block with zeros.
- Typed arrays cannot grow dynamically. If we need to change the size of the typed arrays, we have to create new array buffer with the new size and copy the data.
- Array buffers are just one contiguous chunk of memory. So typed arrays do not provide any way to use multi-dimensional indexing. However, we can provide this facility by adding properties like shape and stride to the typed arrays.

### 2.1.3 Asm.js

Asm.js is a strict subset of JavaScript, that can be used as a low-level, efficient target language for compilers. The subset is defined by a static type system. The asm.js programming model is built around integer and floating-point arithmetic and a virtual heap represented as a typed array. Although, JavaScript does not have a construct for integers, they can be emulated by either integer loads and stores using typed arrays, and integer coercions performed by the JavaScript bitwise operators. Thus, with the combination of static and dynamic validation, asm.js modules are amenable to ahead-of-time optimizing compilation. *Mozilla Firefox* employs an ahead-of-time compiler for valid asm.js code [12], while *Google Chrome* and *Opera* heavily optimizes asm.js style code during JIT compilation [13].

Listing 3 defines an asm.js module which returns a sepia function. The asm.js module is a special function which can have three optional arguments (Line 1). The *stdlib* object provides access to a limited subset of the JavaScript standard libraries (e.g. Math

```
function ImageProcessing(stdlib, foreign, buffer) {
    "use asm";  /* asm.js module */
    /* Globals */
    var image = new stdlib.Uint8Array(buffer);
    var min = stdlib.Math.min;
    /* Module body */
    function sepia(width, height) {
        width = +width;                     /* Double */
        height = +height;                   /* Double */
        var i=0.0,j=0.0,r=0.0,g=0.0,b=0.0,  /* Doubles */
            offset=0;                       /* Integer */

        for(i=0.0; i < +height; i=+(i+1.0)) {
            for(j=0.0; j < +width; j=+(j+1.0)) {
                /* Coercing double result to integer */
                offset = ~~(((i*width)+j)*4.0);

                /* Asm.js uses byte addressing, hence bit-wise shift
                 * operator. Second shift operator for reading
                 * uint value and the result is coerced to double.  */
                r = +(image[offset>>>0]>>>0);
                g = +(image[(offset+1)>>>0]>>>0);
                b = +(image[(offset+2)>>>0]>>>0);

                image[offset>>0] = min(
                        ~~(r * 0.393 + g * 0.769 + b * 0.189), 255)|0;
                image[(offset+1)>>0] = min(
                        ~~(r * 0.349 + g * 0.686 + b * 0.168), 255)|0;
                image[(offset+2)>>0] = min(
                        ~~(r * 0.272 + g * 0.534 + b * 0.131), 255)|0;
            }
        }
    }
    /* Export */
    return {
        sepia: sepia
    };
}
```

**Listing 3** Sepia filter using asm.js module

object). The *foreign* object provides access to custom external JavaScript functions. The *buffer* object provides single ArrayBuffer to act as the asm.js heap. These objects allow asm.js to communicate with external JavaScript. The asm.js module requests validation by using "use asm" prologue directive (Line 2).

The asm.js module is divided into three parts: global declarations (Lines 3-5), module body (Lines 6-33) and single return statement (Lines 34-37). The return statement returns object containing all the functions and properties we want to expose to the external JavaScript. The global declaration part usually creates variables from standard and foreign library that we want to use in the module (Line 5). This part also creates typed array from array buffer (Line 4). Although, JavaScript supports Uint8ClampedArray, asm.js doesn't support it. So we need to use Uint8Array and ensure that the value doesn't go over 255 (Lines 25-30).

Each function in the asm.js module is also divided into three parts. The first part implicitly declares type information about arguments (Lines 8-9). Type information is declared by using unary operators or bit-wise operators. For example, the unary plus operator declares double type variable, bit-wise OR operation with 0 declares an argument as integer type and bit-wise right shift operator with 0 declares an argument as unsigned integer.

The second part declares all the local variables used in the function (Lines 10-11). We cannot declare local variables anywhere else in the function. All the variables must be initialized with the type. We can use literals to initialize variables. A literal containing dot sign makes the variable double (e.g. i,j,r,g,b) and without it makes the variable integer (e.g. offset).

The third part of the function in asm.js module is the function body. All the operations including function calls requires type coercion to double, integer or unsigned integer. Binary operators like plus, minus, multiply, divide are operated only on same type of variables. Hence, addition operation on variable i require 1.0 instead of just 1 (Line 13). Also, multiplication of two integer variables is not allowed. One operator must be a literal to avoid overflow in integers. Thus, we need to declare i and width variables as doubles so that we can multiply them (Line 16). Conversion from integer to double is performed using the unary + operator (Lines 21-23) and conversion from double to integer is performed using special ~~operator (Lines 16, 25-30). Asm.js forces

byte addressing of the heap by requiring shifting operation. We are using Uint8Array (byteLength = 1), so we need to shift all the indexing with 0 (Lines 21-23). Similarly, if we want to use Float32Array (byteLength = 4), then we need to shift indexing with 2.

**Advantages of asm.js are:**

- As asm.js is a strict subset of JavaScript, with the use of static and dynamic optimization, asm.js code can directly be compiled to assembly which can provide great speedups.
- Absence of index bounds checks and garbage collection, and unboxed representations of integers and floating-point numbers make AOT compilation very efficient and the resulting code is much faster than normal JavaScript code.
- The asm.js subset is very well defined, which makes it a good compilation target.

**Disadvantages of asm.js are:**

- The programming model of asm.js is not conventional. It makes manual program writing very hard.
- Manual heap management makes implementation of algorithms very hard.
- As asm.js uses typed arrays, all the disadvantages of typed array also applies to asm.js.

### 2.1.4 Web Workers

Web workers [14] provide a mechanism for web pages to run scripts in background threads. Workers can communicate with the spawning task by posting and receiving messages. Each worker works on a separate context which is different from the current window (window object). The Web Workers API does not provide access to non-thread safe objects like DOM, and each data communication is done through serialized objects to ensure thread safety. Web workers are useful for heavy executions which makes web page unresponsive to user inputs. We can transfer these computations to the new worker thread to avoid the issue [15].

```
function ImageProcessing() {
    var worker, callback;

    function apply_sepia(image, width, height) {
        worker = new Worker("sepia-worker.js");
        worker.postMessage([image, width, height], [image.buffer]);
        worker.onmessage = function(e) {
            this.terminate();
            if(callback) callback.call();
        };
        return ret;
    };

    function done(e) {
        callback = e;
    };

    var ret = {
        sepia: apply_sepia,
        done: done
    };
    return ret;
}
```

```
importScripts('201-sepia-typed-arrays.js');

onmessage = function workerOnMessage(e) {
    sepia(e.data[0], e.data[1], e.data[2]);
    self.postMessage(e.data[0], [e.data[0].buffer]);
};
```

**Listing 4** Sepia filter using Web Workers

Listing 4 contains code for two JavaScript files. The first JavaScript will be included in the web page which will create a worker with the second JavaScript file (Line 5). The page script is creating functions in an ImageProcessing module which exposes sepia and done functions. The *sepia* function takes image and applies sepia filter in a new worker thread and the *done* function takes a callback which is called when worker thread completes its work (Line 9). The worker thread imports another JavaScript (Line 1 in second file) file which contains sepia function for Uint8ClampedArray typed array (Section 2.1.2). Data communication between the worker thread and the spawning thread is done through posting a message (Line 6) and an onmessage event handler (Line 7).

To maintain thread safety, the global context of each worker is different and data communications are done only through serialized objects. So there is no data sharing between worker threads, and during each communication all the passing objects get serialized and de-serialized. Since there is a copy of data, this operation takes more time as the data gets bigger. However, the web worker API provides access to transferable objects. Transferable objects can transfer ownership of the object without copying data. In the page script, the *postMessage* method takes two arguments: a data object and an array containing objects to transfer the ownership. In this example, reference of the image buffer is transfered to the worker thread so there is no actual copying of the image data. However, we are passing typed array view of image, width and height as data so there is a copy of these objects passing to the worker thread. The page script cannot access image buffer as long as the worker thread has its ownership. Thus, we need to pass back the ownership of the buffer to the page script as soon as the processing in worker thread is finished (Line 5 in second file).

**Advantages of web workers are:**

- Web workers makes it possible to work with threading in JavaScript, without worrying about low-level thread safety maintenance.
- Computation heavy operations or I/O operations can be transfered to the workers without affecting the current page.

**Disadvantages of web workers are:**

- Web workers are very resource-intensive. We cannot create a lot of web workers in a single machine. For example, when working with an image, we cannot create workers for each pixel. We should only work on the whole image in a single web worker.

- For regular arrays or non-transferable objects, data communication cost is very high. So if the task is not big enough, it is probably unwise to create worker threads for such objects.

### 2.1.5 WebCL

WebCL is a JavaScript binding to the OpenCL standard for parallel computing. WebCL enables applications to make use of GPU and multi-core CPU parallel processing from web browser which provides significant speedup in the numerical applications. Although WebCL has not been officially integrated with any of the major browsers, there have been some plug-ins developed which provide access to the WebCL API [16, 17].

Listing 5 contains JavaScript code and Listing 6 contains OpenCL kernel for Sepia filter. The program uses Nokia's implementation of WebCL for Mozilla Firefox. The WebCL specification provides API functions to query available GPUs or multi-core CPUs and platforms, and we can select specific device and platform to execute OpenCL kernel (Lines 3-4). Unlike web workers, we need to manually get the OpenCL kernel code by using AJAX query, and compile it before we can use it (Lines 6,7,10). Before we execute the kernel, we need to setup the device memory and copy the ArrayBuffers (Lines 17-18). Since JavaScript and OpenCL both have different data types, the WebCL implementation boxes and unboxes the data communication between JavaScript and OpenCL kernel. When the execution of the kernel gets finished, we can read back the device memory to ArrayBuffers to get the result (Line 30).

**Advantages of WebCL are:**

- WebCL is the first standard that allows for heterogeneous parallel computing in a browser exposing CPUs and GPUs.

- Provides good speed up for highly parallel algorithms.

```
function sepia(image, W, H) {
    //============ Setup WebCL Program ================
    var platform = webcl.getPlatforms()[0],
        device = platform.getDevices(WebCL.DEVICE_TYPE_ALL)[0],
        ctx = webcl.createContext(device),
        src = document.getElementById("clSepia").text,
        program = ctx.createProgram(src),
        queue = ctx.createCommandQueue(device);

    program.build ([device], "");

    // ============== Initialize Kernels ================
    var sepiaKernel = program.createKernel("sepia");

    // ============== Setup Kernel Memory ================
    // memory has to be allocated in terms of bytes
    var image_d = ctx.createBuffer(WebCL.MEM_READ_WRITE, W*H*4);
    queue.enqueueWriteBuffer(image_d, true, 0, W*H*4, image);

    var localSize = [ 4 ];
    var globalSize = [ H*W ];

    // ============== Set Args and Run Kernels ================
    sepiaKernel.setArg(0, image_d);

    queue.enqueueNDRangeKernel(sepiaKernel, 1, null, globalSize, localSize);
    queue.finish();

    // ============== Pull Results ================
    queue.enqueueReadBuffer(image_d, false, 0, W*H*4, image);
    queue.finish();

    // ============== Free Memory ================
    image_d.release();
    program.release();
    sepiaKernel.release();
    queue.release();
    ctx.release();
}
```

**Listing 5** Sepia filter using Nokia WebCL on Mozilla Firefox

```
<!-- Include this kernal script in the html -->
<script id="clSepia" type="text/x-opencl">
    __kernel void sepia(__global char *image) {

        int i = get_global_id(0)*4;

        float r = image[i];
        float g = image[i+1];
        float b = image[i+2];

        barrier(CLK_LOCAL_MEM_FENCE);

        image[i] = (r * 0.393f) + (g * 0.769f) + (b * 0.189f);
        image[i+1] = (r * 0.349f) + (g * 0.686f) + (b * 0.168f);
        image[i+2] = (r * 0.272f) + (g * 0.534f) + (b * 0.131f);
    }
</script>
```

Listing 6 Kernal for Sepia filter using Nokia WebCL

**Disadvantages of WebCL are:**

- There are no major browser vendors, who officially supports WebCL at the moment. So the WebCL implementation is still very immature.
- There are some security checks required to make WebCL applications safe. These checks can slow the performance of the application.

## 2.2 Related Technologies

In this section we will briefly discuss the popular JavaScript libraries for numerical computations and also look at the NumPy, a popular python library for scientific computations. We examine the performance of McNumJS with these JavaScript libraries in Chapter 5.

### 2.2.1 NumericJS

The Numeric Javascript library allows you to perform sophisticated numerical computations in pure JavaScript in the browser and elsewhere [18]. The NumericJS library uses regular arrays to represent matrices and vectors, but internally it creates a special tensor object for computation. It also has the support for complex numbers which are also represented using tensor objects. The NumericJS library is carefully tuned with modern JavaScript engines to obtain good performance for a Javascript program.

While NumericJS uses tensor object based on JavaScript regular arrays, McNumJS uses modified typed array view object for internal computations and data representation.

### 2.2.2 Google Closure Library

The Google Closure Library provides user interface widgets, an event framework, a packaging system, tools for DOM manipulation, tools for creating animation effects, communication utilities, a unit testing framework, and a wide variety of other packages including *math* package for numerical computations [19].

The Google Closure Library has classes for representing coordinates, line, curves, matrices, vectors, etc. All of these classes internally use JavaScript regular arrays and provide numerous methods to work on the objects of these classes. Since JavaScript only has a generic double like object to represent a Number, we cannot create integers. However, different data types are necessary for numerical computations. So the Closure Library also provides an *Integer* and a *Long* class which represents 32-bit and 64-bit signed integers respectively. For operations like addition and multiplication, the library splits each number into 16-bit pieces, and then uses JavaScript's floating-point representation without worrying about overflow or change of sign [20].

McNumJS uses typed arrays which provides different numeric data types like Int32Array, Uint16Array, Float32Array, Float64Array, etc. So there is no need to manually handle overflows or sign. The Closure Library provides a lot of modules used for building complex web applications, while our focus in McNumJS is only numerical computations.

### 2.2.3 Sylvester

Sylvester is a vector, matrix and geometrical library for JavaScript, that runs in the browser and on the server side. It includes classes for modeling vectors and matrices in any number of dimensions, and for modeling infinite lines and planes in 3-dimensional space [21]. Sylvester also uses JavaScript regular arrays for creating objects of these classes. Sylvester provides methods to work with accuracy and to handle floating-point rounding errors for JavaScript *Number* object. This library does not provide any way to represent Unsigned Integers, Integers or Long.

Sylvester uses objects like the Closure Library to represent matrices and vectors. These objects are not carefully tuned to provide better performance. The basic purpose of the library is to provide functionality not the performance, while McNumJS provides good performance as well as makes it easy-to-use.

### 2.2.4 NumPy

NumPy is the fundamental package for scientific computation with Python [11]. It contains among other things:

- a powerful N-dimensional array object,
- sophisticated (broadcasting) functions,
- tools for integrating C/C++ and Fortran code, and
- useful linear algebra, Fourier transform, and random number capabilities.

NumPy's main object is the homogeneous multidimensional array. It is a table of elements (usually numbers), all of the same type, indexed by a tuple of positive integers. In Numpy, dimensions are called axes. The number of axes is called rank.

We can compare this object with JavaScript's typed array. However, JavaScript typed arrays do not support multi-dimension. We can provide multi-dimension in the typed arrays by adding shape and stride properties. Considering this similarity, we can create a JavaScript library using typed arrays which provides API functions similar to the NumPy. This will also fulfill one of our goals of creating the library: ease of use.

# Chapter 3

# Performance Analysis of asm.js

In this chapter, we are going to find more about asm.js and we will experiment with the different ways to use asm.js. We are studying asm.js as it allows ahead-of-time compilation and provides better speedup than normal JavaScript. We would like to find the parts of asm.js specifications that we can incorporate into regular JavaScript development and get better performance.

## 3.1 Overview of asm.js Specifications

In this section, we are going to look over the asm.js specifications briefly. Listing 3 shows the example of the asm.js code. We summarize the asm.js specifications as follows:

- The asm.js programming model is built around integer and floating-point arithmetic and a virtual heap represented as a typed array.
- The asm.js module is a function with three optional arguments. Two of these arguments allows communicating with the JavaScript standard library and foreign functions. The third argument represents the virtual heap buffer.
- The asm.js module validation is requested by "use asm" directive at the beginning of the module.
- The asm.js module has exactly three parts: global declaration, module body and a single return statement.

- Function arguments must be of primitive types: unsigned int, int or double. Types must be provided to function arguments by using bit-wise operators.
- Variables must be initialized to primitive types. They cannot be null or undefined. Integers are initialized by integer literals and doubles are initialized by literals with floating-point.
- Arrays (regular, typed or ArrayBuffers) cannot be created dynamically. We can only use the heap buffer to access arrays.
- The heap cannot be modified outside the functions.
- All the operations must be typed and if type conversion is required, it must be done as per the grammar.

Based on these specifications, we will conduct few experiments with the asm.js modules. These experiments would ignore some of the grammar rules to make it more like regular JavaScript. This way we can check if some or any specification of asm.js can be used for regular JavaScript development.

## 3.2 Asm.js Experiments

In this section, we define six different versions of the same micro-benchmark to check the performance of the implementations using some or all of the asm.js specifications. In the following section, we compare the performance of each of these versions.

### 3.2.1 Normal JavaScript

Listing 7 measures the performance of element-wise addition of two vectors. In the listing, *in1, in2* are the input vectors of type Float64Array and of size 8000 and *out* is the output vector of same type and size. In normal JavaScript, we iterate over the length of the vector and add the input vectors element-wise, and store the result into the output vector.

```
function dotAdd(a, b) {
    return a + b;
}

function add(in1, in2, out) {
    for(var l=out.length, i=0; i<l; ++i) {
        out[i] = dotAdd(in1[i], in2[i]);
    }
}

// We will benchmark this fn call
add(in1, in2, out);
```

**Listing 7** Normal JavaScript

```
function dotAdd(a, b) {
    a = +a; b = +b;
    return +(a + b);
}

function add(in1, in2, out) {
    var l = out.length|0, i=0;
    for(; i<l; i = (i+1)|0) {
        out[i] = +dotAdd(in1[i], in2[i]);
    }
}

// We will benchmark this fn call
add(in1, in2, out);
```

**Listing 8** Normal JavaScript with type coercion

### 3.2.2 Function with Type Coercion

Listing 8 is the same micro-benchmark but this version contains the addition function with type coercion as per the asm.js specifications. It is important to note that we coerced type information as much as possible. We cannot provide type to *in1, in2* or *out*. In this version, we are ignoring all the other asm.js specifications. As we provide type information to the variables, JavaScript engines can better optimize the code and provide better performance.

### 3.2.3 Function with Type Coercion in Strict Mode

Listing 9 is the same as the previous version except the functions defined are in strict mode. JavaScript strict mode enforces stricter parsing and error handling on JavaScript code at runtime [22]. It is important to see that how strict mode affects the type coercion in JavaScript.

```
function dotAdd(a, b) {
    "use strict";
    a = +a; b = +b;
    return +(a + b);
}

function add(in1, in2, out) {
    "use strict";
    var l = out.length|0, i=0;
    for(; i<l; i = (i+1)|0) {
        out[i] = +dotAdd(in1[i], in2[i]);
    }
}

// We will benchmark this fn call
add(in1, in2, out);
```

Listing 9 Type coercion in strict mode

### 3.2.4 Calling asm.js Function from Normal JavaScript

In Listing 10, we kept the array iteration intact, but the addition function was changed to the asm.js module. In this version, we convert scalar operations to asm.js module while keeping the arrays outside of the module. The module contains the addition function (Line 4) with all the asm.js specifications kept in consideration. In this version, we inspect the performance of asm.js when called from a normal JavaScript loop body, without creating a heap. The outcome of this version is of great significance as attaining a better performance with this version would obviate the requirement of creating a heap.

```
1  function AsmPointMath() {
2      "use asm";
3
4      function dotAdd(a, b) {
5          a = +a; b = +b;
6          return +(a + b);
7      }
8
9      return { dotAdd: dotAdd };
10 }
11
12 function add(in1, in2, out) {
13     var math = AsmPointMath();
14     for(var l=out.length, i=0; i<l; ++i) {
15         out[i] = math.dotAdd(in1[i], in2[i]);
16     }
17 }
18
19 // We will benchmark this fn call
20 add(in1, in2, out);
```

**Listing 10** Add method in asm.js calling from Normal JavaScript

### 3.2.5 Calls between Different asm.js Modules

In Listing 11, we replace the array iteration and addition functions of normal JavaScript

```
function AsmPointMath() {
    "use asm";

    function dotAdd(a, b) {
        a = +a; b = +b;
        return +(a + b);
    }

    return { dotAdd: dotAdd };
}

function AsmMath(stdlib, foreign, buffer) {
    "use asm";

    var l = 8000,
        in1 = new stdlib.Float64Array(buffer, 0<<3, 8000),
        in2 = new stdlib.Float64Array(buffer, 8000<<3, 8000),
        out = new stdlib.Float64Array(buffer, 16000<<3, 8000),
        dotAdd = foreign.dotAdd;

    function add() {
        var i=0<<3, j = l<<3;
        for(; (i|0)<(j|0); i = (i+8)|0) {
            out[i>>3] = +dotAdd(+in1[i>>3], +in2[i>>3]);
        }
    }

    return { add: add };
}

var heap = new Float64Array(8000*3);
heap.set(in1);
heap.set(in2, 8000);
heap.set(out, 16000);

// We will benchmark following code
var pointMath = AsmPointMath();
AsmMath(window, pointMath, heap.buffer).add();
```

**Listing 11** Add method in asm.js calling from different asm.js module

with the respective asm.js modules. The module with addition function is same as the previous version. We pass this module as a foreign library and the *window* object as a standard library to the *AsmMath* module. However, we cannot pass multiple typed arrays or cannot create arrays dynamically inside asm.js module. So we created a *heap* and added *in1, in2* and *out* to this heap (Lines 31-34). It is important to note that we are not benchmarking the heap buffer creation time or data copying time. As asm.js only uses typed arrays, the data generation time for different types of asm.js modules will be same. So, we are more interested in the computation time than the data generation time. We then pass the heap buffer to the AsmMath module. In AsmMath module, we create *in1, in2* and *out* typed array views from the heap buffer using standard library functions (Lines 16-18). It is important to note that we are using byte addressing of typed arrays, as specified by asm.js grammar, instead of providing indices. We call the *dotAdd* function from foreign library for addition.

Inter-modular function calls are quite common in any library. Thus, it is critical to study the performance of this version.

### 3.2.6 Complete asm.js Module

Instead of creating two different modules for array iteration and addition functions, we compose them inside same the module in Listing 12. This is a complete asm.js module. Similar modules can be expected to be generated upon compilation of applications in other languages to asm.js. Thus, it is important to check the performance of this version and compare it with the other versions to see how the compilation target fares.

## 3.3 Results

In this section, we are going to see the performance results of the aforementioned versions of the micro-benchmark. We test the performance of the different versions of the benchmark for two types of arrays: Int32Array and Float64Array. We use popular JavaScript performance playground jsPerf [23] to measure the performance. We used standard consumer laptop to test these versions. The configuration of the laptop is given in Table 3.1. jsPerf provides performance result in terms of operations per second, thus higher numbers are desired.

```
function AsmMath(stdlib, foreign, buffer) {
    "use asm";

    var l = 8000,
        in1 = new stdlib.Float64Array(buffer, 0<<3, 8000),
        in2 = new stdlib.Float64Array(buffer, 8000<<3, 8000),
        out = new stdlib.Float64Array(buffer, 16000<<3, 8000);

    function dotAdd(a, b) {
        a = +a; b = +b;
        return +(a + b);
    }

    function add() {
        var i=0<<3, j = l<<3;
        for(; (i|0)<(j|0); i = (i+8)|0) {
            out[i>>3] = +dotAdd(+in1[i>>3], +in2[i>>3]);
        }
    }

    return { add: add };
}

var heap = new Float64Array(8000*3);
heap.set(in1);
heap.set(in2, 8000);
heap.set(out, 16000);

// We will benchmark following code
AsmMath(window, {}, heap.buffer).add();
```

**Listing 12** Add method in asm.js

| | Desktop |
|---|---|
| CPU | Intel Core i5, 2.50GHz × 4 |
| Cache | 4 MiB |
| Memory | 6 GiB |
| OS | Ubuntu 12.04 LTS |
| Chrome | 33.0.1750 |
| Firefox | 30.0 |

**Table 3.1** Specifications of the machine used for asm.js experiments

Table 3.2a and Table 3.2b shows the result in operations per second for the different versions of the micro-benchmark for Chrome and Firefox respectively. We have defined shorter names for these micro-benchmarks which are: normal for normal JavaScript; normal-typed for normal JavaScript with asm.js type coercing; strict-typed for type coercing in strict mode; js-asm for calling asm.js function from normal JavaScript; asm-asm for calls between different asm.js modules and asmjs for complete asm.js module.

In Chrome, the slowest performer is asm-asm version of the benchmark. The normal and strict-typed also do not perform that well in comparison to the other versions. The version delivering the best performance is the asm.js version for Int32Array and the normal-typed version for Float64Array. However, there is not much difference between the performance of the asm.js and the normal-typed versions for both, Int32Array and Float64Array.

We can notice the substantial performance hit that the Firefox takes for js-asm and asm-asm versions. In these cases, the addition function is inside a separate asm.js module. So the module is called several times. The asm.js implementor has reported a bug and mentioned that calls to asm.js functions from a non-asm.js functions and vice versa are much slower than normal function calls due to general-purpose enter/exit routines [24]. It is important to note that the Mozilla Firefox employs an ahead-of-time validation and compiler for asm.js code.

In Firefox, asm.js performs the fastest among all the other versions for both: Int32Array and Float64Array. However, we can see that normal-typed and normal versions are also not far behind.

Based on these numbers, Chapter 4 discusses the design and technological choices made for developing the McNumJS.

| | Int32Array | Float64Array |
|---|---|---|
| normal | 13774 | 8298 |
| normal-typed | 47648 | 58001 |
| strict-typed | 13128 | 8055 |
| js-asm | 47385 | 58606 |
| asm-asm | 12234 | 6196 |
| asmjs | 57983 | 57835 |

(a) Chrome

| | Int32Array | Float64Array |
|---|---|---|
| normal | 87979 | 57861 |
| normal-typed | 88007 | 57875 |
| strict-typed | 59034 | 56870 |
| js-asm | 805 | 795 |
| asm-asm | 873 | 896 |
| asmjs | 88280 | 57982 |

(b) Firefox

**Table 3.2** Asm.js experiment results in operations per second (Higher numbers desirable) (normal = Normal JavaScript; normal-typed = Normal JavaScript with asm.js type coercing; strict-typed = Type coercing in Strict mode; js-asm = Calling asm.js function from Normal JavaScript; asm-asm = Calls between different asm.js modules; asmjs = Complete asm.js module)

# Chapter 4

# McNumJS - A JavaScript Library for Numerical Computations

So far, we studied the JavaScript features and technologies available for numerical computations. We also studied the asm.js specifications and analyzed the performance of asm.js. In this chapter, we discuss the technological choices we made to develop McNumJS. We talk about the McNumJS library and its modules and we also briefly describe its development process.

## 4.1 JavaScript Features and Technology Selection

In this section, we argue for the selection of JavaScript features and technologies to develop McNumJS.

**Regular Arrays:** JavaScript regular arrays can contain any data including numeric, string, array or object. Moreover, the length of regular arrays can be varied dynamically. Thus, there exists array bounds check, dynamic memory allocation, type checking, etc. JavaScript regular arrays are represented as hash-tables in the JavaScript engines. Due to these reasons, JavaScript regular arrays do not provide good performance. So even though JavaScript regular arrays can support numerical computations, the lack of performance makes the numerical computations slow. For this reason, we decided not to use the JavaScript regular arrays as the base matrix container.

**Typed Arrays:** JavaScript typed arrays are similar to C arrays except that the typed arrays are only one-dimensional. Typed arrays provide better performance as they represent one-dimensional memory buffers. To access the memory buffer, there exists typed array view which provides a context - that is, a data type, starting offset, and number of elements - that turns the data into an actual typed array. So JavaScript typed arrays provide better performance but being single-dimensional makes them hard to use. However, we can add shape information to the typed array views to make it multi-dimensional. This makes them a better performing choice for numerical computations as well as easy-to-use.

**Asm.js:** We can use asm.js in three different ways: (1) compile existing libraries to asm.js; (2) develop a library using asm.js; or (3) develop a library in normal JavaScript but with type coercion provided by asm.js specifications.

There are several libraries available in many different languages that provide great performance. We can compile these libraries to asm.js, to use them as numerical libraries in JavaScript. However, the problem with this approach is that the compiled code size will be huge and debugging of the code will be a non-trivial task. Moreover, we will still need to write a wrapper code for this compiled code to keep the dynamic behavior of JavaScript and provide high-level functions. This wrapper will also need to create a heap buffer and copy array data to this buffer to use inside asm.js module. Thus, it may not be beneficial to compile an existing library to asm.js.

As mentioned in Section 2.1.3, developing a library in asm.js is not trivial in itself. The programming model of asm.js is not easy to work with and it was mainly developed for compilation target. Also, as previously mentioned, we need to create a heap buffer and copy the data. So even though we find in Section 3.3 that asm.js performs fastest among other implementations, we cannot use it to develop the library.

In Section 3.3, we reported the result of different asm.js experiments and we saw that the fastest version is the complete asm.js module. But we also noted that the normal JavaScript with type coercion is also not far behind in terms of performance. Providing type information to function arguments and local variables

according to asm.js specification, is easy enough. We ignore the other grammar rules specified by asm.js. So we do not need to create the heap buffer and copy data. Hence, we will use type coercing in the development of the McNumJS.

**Web Workers:** Web workers allows us to create multi-threaded JavaScript applications. However, the web workers are not light weight. We cannot create threads for each array index. To ensure thread safety, deep object copying occurs to pass the data between threads. Web workers are meant to defer script execution in a new thread, leaving main thread free to maintain its responsiveness. If we use web workers to parallelize numeric applications, they will introduce slowdown because of data copying. Web workers might show potential if we were to implement lazy computation and group multiple operations. However, we are not implementing lazy computation in our library for now and thus, web workers are not of any use.

**WebCL:** WebCL allows heterogeneous parallel computing in a browser exposing CPUs, GPUs and DSPs. It is a web equivalent of OpenCL. However, WebCL is still in development and is not supported by any major browser vendors and requires plug-ins to use it. WebCL also is still immature and provides performance gain only for highly parallel benchmarks. The results in WebCL over JavaScript are not congruent with OpenCL over C [25]. Due to the lack of support in popular browsers and the limitations of current implementation, we are not going to use it to develop McNumJS.

In conclusion, we use type coercing rules defined by the asm.js specifications to improve JIT optimization. We also use typed arrays to get faster array access. We change typed arrays and add shape information to map multi-dimensional indices to a single-dimensional index. In the next section, we will elaborate this topic and discuss more about the architecture of the McNumJS.

## 4.2 Architecture

The McNumJS library is built using modular design. There are currently four different modules: (1) core module; (2) generation module; (3) unary operations module; and

(4) binary operations module. We elaborate on these modules in the following sections.

### 4.2.1 Core Module

The core module in McNumJS creates a global object *mn*, which contains all the McNumJS API functions. The main part of core module is to extend typed array views and provide multi-dimensional array support.

In JavaScript, there is no standard way to directly change the constructor of typed arrays. So we first create an alias of the typed array and then create a function variable of the typed array. Listing 13 shows the example of changing Uint16Array constructor. Table 4.1 lists typed arrays in JavaScript and its alias in McNumJS library. The new constructor now takes four more optional arguments than standard constructor: shape, stride, offset and order. McNumJS creates an array view object by calling the standard constructor from the new constructor, then calculates the value of the extra properties, and adds them to the object. Thus, if someone want to use standard typed array constructor arguments while using McNumJS, it would still work.

```
1 var UI16A = Uint16Array;
2
3 var Uint16Array = function Uint16Array_constructor
4                     (data, shape, stride, offset, length) {
5     // ...
6 }
```

Listing 13 Changing Uint16Array Constructor

Listing 14 shows the possible Uint16Array constructor syntax in McNumJS. McNumJS typed arrays supports the normal typed array constructors specified in *ECMAScript 6* (Lines 2-5). In addition, McNumJS supports some more syntax to support multi-dimension (Lines 8-12). Table 4.2 shows the typed array constructor parameters and provides a brief description about them.

Before moving further, let us discuss more about the shape, stride and order parameters. The shape parameter represents the shape of the array. As JavaScript typed

| Typed Array | Description | Alias in McNumJS |
|---|---|---|
| Int8Array | 8-bit twos complement signed integer | I8A |
| Uint8Array | 8-bit unsigned integer | UI8A |
| Uint8ClampedArray | 8-bit unsigned integer (clamped) | UI8CA |
| Int16Array | 16-bit twos complement signed integer | I16A |
| Uint16Array | 16-bit unsigned integer | UI16A |
| Int32Array | 32-bit twos complement signed integer | I32A |
| Uint32Array | 32-bit unsigned integer | UI32A |
| Float32Array | 32-bit IEEE floating point number | F32A |
| Float64Array | 64-bit IEEE floating point number | F64A |

**Table 4.1** Typed array views in JavaScript and its alias in McNumJS.

```
// Standard Typed Array Constructors:
new Uint16Array(length);
new Uint16Array(typedarray);
new Uint16Array(object);    // Single-dimensional
new Uint16Array(buffer [, byteOffset [, length]]);

// New in McNumJS:
new Uint16Array(length [, shape, [stride, [offset, [order]]]]);
new Uint16Array(typedarray [, shape, [stride, [offset, [order]]]]);
new Uint16Array(object);    // Multi-dimensional
new Uint16Array(object [, shape, [stride, [offset, [order]]]]);
new Uint16Array(buffer [, shape, [stride, [offset, [order]]]]);
```

**Listing 14** Uint16Array Constructor in McNumJS

| Parameter | Description |
|---|---|
| length | length of the typed array to create. |
| typedArray | copies the buffer of the *typedArray* and returns a new typed array. |
| object | A new typed array is created from an object. In normal JavaScript, the object has to be single-dimensional regular array. However, in McNumJS, the object can be regular array or multi-dimensional arrays-of-arrays. McNumJS automatically calculates shape and stride in this case. |
| buffer | a new typed array is created on the given ArrayBuffer. |
| byteOffset (optional) | number of bytes to offset in the buffer from the first index. Default: 0. |
| shape (optional) | an array representing the shape of the array. Default: array of single element containing the length of the array |
| stride (optional) | an array representing the stride of the array. Default: Calculates based on the shape and order of the array. |
| offset (optional) | number of elements to offset from the starting element. Default: 0. |
| order (optional) | represents the order of the elements arranged in the array. This parameter can be either "r" representing row-major order or "c" representing column-major order. Default: "r". |

**Table 4.2** TypedArray constructor parameters in McNumJS

arrays are single-dimensional arrays, shape provides it dimensions. For example, a 2x3 bi-dimensional matrix has the shape of [2, 3].

We calculate stride property in the constructor if it is not provided in the constructor arguments. The stride is a tuple of elements to step in each dimension when traversing an array. Consider Listing 15 for the examples. In the first example, we create 2x3 matrix. The order of the elements is row-major by default, meaning the elements are stored in row-by-row. Lines 4-5 show the 2-dimensional matrix and line 6 shows the one-dimensional representation of the same matrix in row-major order. To step in the first dimension, we just need to go to the next element in one-dimensional representation. For example, stepping from (0,1) to (0,2) would just require moving to the next element in one-dimensional array. Thus, the stride of first dimension is 1. Similarly, to step in second dimension, we need to go to the third element. For example, stepping from (0,1) to (1,1) would require moving from (1) to (4) in one-dimensional array. Thus, the stride of second dimension is 3.

The second example in Listing 15 creates 2x3 matrix of the order column-major, meaning the elements are stored in column-by-column. Lines 17-18 show the 2-dimensional matrix and line 19 shows the one-dimensional representation of the matrix in column-major order. To specify the column-major order, we pass the argument "c" in the constructor. Similar to the previous example, we calculate the stride of the matrix.

We calculate the stride to convert multi-dimensional indices to a single-dimensional index. As shown in lines 12 and 25, we can multiply stride elements with that of indices and add them to get the single-dimensional index. The calculation of stride can vary based on the order of the data. The stride of the first dimension of the row-major matrix is 1. The stride for the rest of the dimensions are multiplication of the size of the previous dimensions. In Listing 15, the stride for the first dimension is 1 and the second dimension is 1*3=3 (Line 10). Similarly, the stride of the last dimension of the column-major matrix is 1. The stride for the rest of the dimensions are multiplication of the size of the higher dimensions. In the second example of Listing 15, the stride for the last dimension is 1 and the first dimension is 1*2=2 (Line 23).

Based on the aforementioned constructor, we calculate and add following properties or methods to the typed array view object[1]:

[1]Figure 4.1 shows the full syntax of these properties and methods.

```
// constructor: Int32Array(object, shape)
var intView = new Int32Array([1,2,3,4,5,6], [2,3]);
// 2x3 Int32 Matrix default row-major
/**  1 2 3
  *  4 5 6
  *  => [1, 2, 3, 4, 5, 6]
  */
console.log(intView.size);      // 6
console.log(intView.shape);     // [2, 3]
console.log(intView.stride);    // [3, 1]
console.log(intView.get(0,1));  // 2
// index: 0*3+1*1 = 1

// constructor: Int32Array(object, shape, stride, offset, order)
var intView = new Int32Array([1,2,3,4,5,6], [2,3], null, 0, 'c');
// 2x3 Int32 column-major Matrix
/**  1 3 5
  *  2 4 6
  *  => [1, 2, 3, 4, 5, 6]
  */
console.log(intView.size);      // 6
console.log(intView.shape);     // [2, 3]
console.log(intView.stride);    // [1, 2]
console.log(intView.get(0,1));  // 3
// index: 0*1+1*2 = 2
```

**Listing 15** Example of strides

**shape:** an array representing the shape of the array,

**stride:** an array representing the stride of the array,

**offset:** an integer representing the offset, and

**size:** an integer representing the size of the array,

```
var intView = new Int32Array([1,2,3,4,5,6], [2,3]);
/* 2x3 Int32 Matrix: [ 1 2 3
                       4 5 6 ]
*/
console.log(intView.shape);     // [2, 3]
console.log(intView.stride);    // [3, 1]
console.log(intView.get);
/*
function (i0, i1) {
    return this[2*(i0|0) + 1*(i1|0)];
}
*/

console.log(intView.get(1,1));  // 5
```

**Listing 16** Example of get method for 2-dimensional array

We also dynamically add the following methods to get, set and map the multi-dimensional indices to a single-dimensional index.

**index:** maps multi-dimensional indices to a single-dimensional index,

**get:** access the element at the given multi-dimensional indices, Listing 16 shows a get method for a 2-dimensional array. The ordering of the array is row-major, and

**set:** set the value of the element at the given multi-dimensional indices.

We have also added static methods to each of the typed array classes (i.e. Float64Array, Int32Array, etc.) to allow changing shape and stride dynamically. However, we do not

change the shape and stride of the same typed array view but we create a new view with the same array buffer (data) and new shape and stride properties. We make all the properties and methods read-only to improve the JIT compilation. Our preliminary results shows that for 2-dimensional matrix of size 400x400, we get a speedup of 2x by making these properties read-only as JIT compiler can better optimize computations if data is immutable. We have added following static properties/methods to the typed arrays view class:

**clone:** Creates a array buffer, copies the data from the calling object and creates a new view object on the new array buffer.

**reshape:** Creates a new view object with the new shape and stride properties.

**slice:** Slices the array given the lower and upper bounds arrays. It creates a new view object on same array buffer and changes the shape, stride and offset properties.

**class:** Refers to the class of the typed array view.

**map:** Similar to JavaScript regular array map function, iterates through the array and calls the callback function. Unlike regular map function, the second argument of the callback function provides the multi-dimensional array indices instead of single index value.

Listing 17 shows an example of the above methods. We create an Int32Array of shape 3x3. The clone method will copy the ArrayBuffer and return a new Int32Array object. The reshape method returns a 3x2 array view on same ArrayBuffer with updated shape and stride properties. The slice method, given lower and upper bound of (0,1) and (2,3) returns a new 2x2 array view on same buffer but with updated shape, stride and offset properties. It is important to note that the reshape and slice methods do not copy the underlying data but it returns a new view object with the changed properties. We can still use the current view object but it would have the unchanged shape and stride properties. The map function iterates through all the elements and add a constant 10 to the each of the element.

Let us compare the time complexity of traversing an array in regular arrays and McNumJS typed arrays. Consider $d$ as the number of dimension of an array and $n$ as

```
var intView = new Int32Array(
        [1,2,3,4,5,6,7,8,9], [3,3]);

var copy = intView.clone();
// Copies array buffer and returns a new view object

var clipped = intView.reshape([3, 2]);
// Returns a new view with new shape on same ArrayBuffer
/* 1 2
 * 3 4
 * 5 6
 */
console.log(clipped.shape);     // [3, 2]
console.log(clipped.stride);    // [2, 1]
console.log(clipped.size);      // 6

var sliced = intView.slice([0, 1], [2, 3]);
// Returns a new view with sliced shape on same ArrayBuffer
/* 2 3
 * 5 6
 */
console.log(clipped.shape);     // [2, 2]
console.log(clipped.stride);    // [3, 1]
console.log(clipped.size);      // 4
console.log(clipped.offset);    // 1

sliced.map(function(v) {
    return v + 10;
});
/* 12 13
 * 15 16
 */
```

**Listing 17** Example of methods added to typed array class

the total number of elements. To access an index in JavaScript regular arrays, there is $\mathcal{O}(d)$ indirect memory reads or array accesses. So, the time complexity of regular arrays (arrays of arrays) for traversal including indirect memory reads is $\mathcal{O}(dn)$. Also, array traversal is oblivious to the memory caching. The time complexity of McNumJS typed arrays for accessing an index is also $\mathcal{O}(d)$ as there is no indirect memory reads but there is mapping of multi-dimensional indices to a single-dimensional index. However, the array traversal is iterating through the elements and does not require mapping of indices, which makes the time complexity $\mathcal{O}(n)$. Also, array traversal in McNumJS is most likely to get benefit from the memory caching, as the typed arrays are stored as a stretch of contiguous memory location.

In summary, the architecture of the McNumJS typed array is shown in the Figure 4.1. We can compare this architecture with the normal JavaScript typed array architecture shown in Figure 2.1.

### 4.2.2 Generation Module

The generation module contains API functions to generate matrices or vectors and fill the values. For example, *zeros* will create a matrix of specified shape and class and fill it with zeros. Similarly, there are API functions to create matrices of ones, linearly spaced numbers, random numbers, etc. Listing 18 shows the syntax and examples of generation module functions. Table 4.3 provides more description about the types and meaning of all the parameters of these functions.

| Parameter | Description |
|---|---|
| shape | an array representing the shape of the array. |
| className | Class of the typed array to generate. Default: Float64Array. |
| start, stop | End points for the linearly spaced points or range. |
| n | Number of points to generate for linearly spaced points. Default: 100. |
| step | step to generate next number in a range. Default: 1 or -1 based on start and stop values. |
| N | Generates identity matrix of shape NxN. |

**Table 4.3** TypedArray constructor parameters in McNumJS

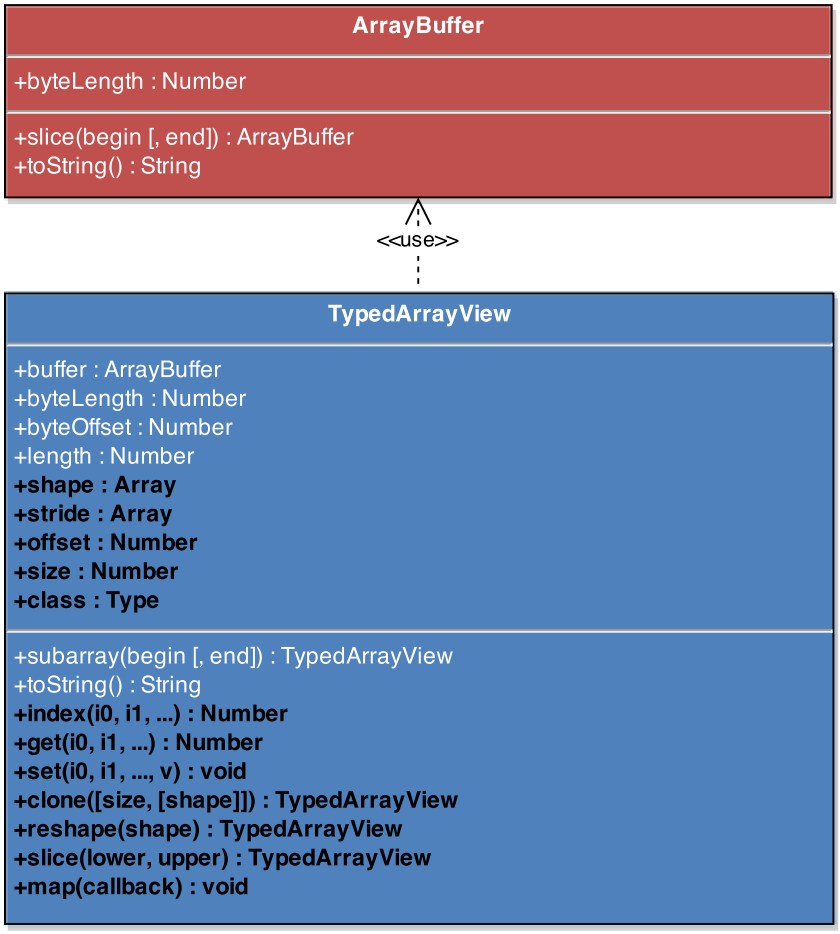


**Fig. 4.1** Architecture of JavaScript Typed Arrays (The properties and methods shown in black are added in the McNumJS)

```
// mn.zeros(shape [, className]);
mn.zeros([6, 6]);
// Float64Array of shape 6x6

// mn.ones(shape [, className]);
mn.ones([6, 6], Int32Array);
// Int32Array of shape 6x6 filled with 1

// mn.rand(shape [, className]);
mn.rand([6, 6], Int32Array);
// Int32Array of shape 6x6 filled with random numbers

// mn.linspace(start, stop [, n [, className]]);
mn.linspace(0, 99);
// Float64Array of size 100 filled 0 ... 99

// mn.range(start, stop [, step, [className]]);
mn.range(10, 1);
// Float64Array of size 10 filled 10 ... 1

// mn.identity(N [, className]);
mn.identity(5, Uint8Array);
// Identity matrix of type Uint8Array and shape 5x5
```

**Listing 18** Generation module functions syntax and examples

### 4.2.3 Unary Operations Module

The unary operations module provides functions to perform unary operations like negate matrix, sum of all the elements, get Sine of the matrix elements, etc. The unary operations also works on scalars. These API functions calls the functions specific to the type of the arguments. For example, the negate function checks if the argument is scalar or an array, and in case of scalars, it directly negates it and returns the result or calls the negation function for arrays which iterates through the array and negates the array elements. We also check if the array's size and length mismatches. If it is not, then we can just iterate through the entire array and do the operation to make the operation faster. Else, we iterate using strides and do the operation. Listing 19 shows the syntax of unary operation functions of McNumJS.

```
mn.fill(typedarray, value);
mn.negate(typedarray);

mn.sin(typedarray);
mn.cos(typedarray);
mn.abs(typedarray);
mn.sqrt(typedarray);
mn.exp(typedarray);
mn.ceil(typedarray);
mn.floor(typedarray);
mn.round(typedarray);

mn.sum(typedarray);
mn.average(typedarray);
mn.norm2(typedarray);

mn.transpose(typedarray [, axis1, axis2]);
```

**Listing 19** Unary operations module functions syntax

An important API function in the unary operations module is the transpose function. With regular arrays, it is necessary to create a new array with the transposed shape and copy the data according to the transpose of a matrix. In case of McNumJS typed arrays,

we can just change the order of accessing index to transpose a matrix. For example, an array ordered by row-major can be transposed by changing access order to column-major. Thus, we do not need to copy the data to get a transpose of a matrix.

In Listing 20, we create 2x3 Int32Array. The transpose function on this array changes shape of the array to [3, 2] and stride to [1, 3]. The new view is shown in the lines 15-17. Note that the ArrayBuffer is same and is not copied. Now we can see that the index (0, 1) maps to (3) in single-dimensional index instead of (1). Thus, regardless of the number of elements, we can create the transpose of the matrix fairly easily and efficiently by changing the shape and stride.

```
1  var intView = new Int32Array([1,2,3,4,5,6], [2,3]);
2  // 2x3 Int32 Matrix default row-major
3  /**  1 2 3
4    *  4 5 6
5    *  => [1, 2, 3, 4, 5, 6]
6    */
7  console.log(intView.size);      // 6
8  console.log(intView.shape);     // [2, 3]
9  console.log(intView.stride);    // [3, 1]
10 console.log(intView.get(0,1)); // 2
11 // index: 0*3+1*1 = 1
12
13 var T = mn.transpose(intView);
14 // 3x2 column-major Matrix
15 /**  1 4
16   *  2 5
17   *  3 6
18   *  => [1, 2, 3, 4, 5, 6]
19   */
20 console.log(T.size);      // 6
21 console.log(T.shape);     // [3, 2]
22 console.log(T.stride);    // [1, 3]
23 console.log(T.get(0,1)); // 4
24 // index: 0*1+1*3 = 3
```

**Listing 20** Example of transpose

### 4.2.4 Binary Operations Module

The binary operations module contains the binary operations like add, subtract, multiply, divide. The binary function can accept scalars and matrices. Based on the function arguments, these functions call the type specific function. Listing 21 shows the syntax of binary operation functions of McNumJS. In the syntax, *in1* and *in2* are the inputs which can be scalars, matrix. *out* is the optional argument for output which by default creates a new typed array based on the inputs.

```
mn.add(in1, in2 [, out]);
mn.addeq(in1, in2);

mn.subtract(in1, in2 [, out]);
mn.subtracteq(in1, in2);

mn.dot(in1, in2 [, out]);
mn.doteq(in1, in2);

mn.divide(in1, in2 [, out]);
mn.divideeq(in1, in2);
```

Listing 21 Binary operations module functions syntax

## 4.3 Development Process

In this section, we overview the development process of the McNumJS and the tools involved in the process.

In development of the library, there are several tasks we need to perform redundantly. For example, we override the constructor of the typed arrays to support multi-dimensional indices. There are currently nine typed arrays supported by the modern JavaScript engines. We need to create constructors for all of them. To automate these redundant tasks, we create macros and expand them using sweet.js. Sweet.js provides the hygienic macros from the languages like Scheme or Rust to JavaScript [26]. We

can expand the macros either dynamically or statically, but to make the McNumJS less dependent, we expand the macros statically.

Sweet.js generates the output code in three phases: (1) Parsing and expanding; (2) hygienic changes (like removing temporary variables, renaming variables, etc.); and (3) AST creation, parsing, validating AST and generate code using escodegen [27] (ECMAScript code generator). However, escodegen does not recognize the asm.js code. So it removes some parenthesis and operations while generating output code that are required to coerce type. For example, as shown in Listing 3, we use 0.0 to coerce double type to variables. This declaration is reduced to 0 instead making it of type integer. Thus, to develop McNumJS, we just use the first phase of the sweet.js to avoid removal of operations and parenthesis that are required.

We use grunt JavaScript task runner [28] to build the library. The grunt task runner works on node.js and automates the build process. Figure 4.2 shows the development process of the McNumJS. We first concatenate all the module files, including macro files and then pass it to the sweet.js to expand the macros. The output code is the distribution-ready McNumJS library. We have used unit-testing framework TAPE [29] to test the library. There are test files for each of the modules[2].

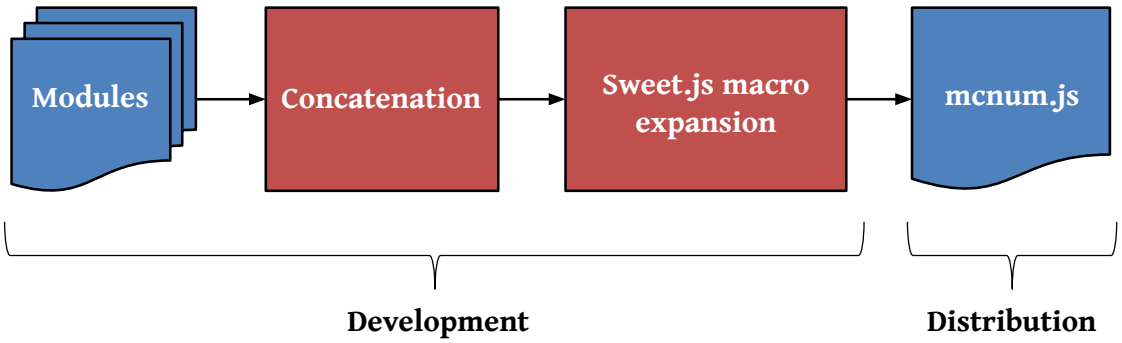


**Fig. 4.2** Development Process of McNumJS

[2]McNumJS library is available at http://www.sable.mcgill.ca/mclab/projects/mcnumjs/

# Chapter 5

# Performance Results

McNumJS dynamically adds properties and methods to typed arrays to support multi-dimensional arrays. We extend the constructor of typed arrays and calculate these properties dynamically inside this constructor. This additional calculation may affect the performance of typed arrays. Moreover, we use get and set methods to map multi-dimensional indices to a single dimensional index. Making calls to these functions for accessing index can also negatively affect the performance.

In this chapter, we want to study the loss of performance due to these extra calculations. We also want to study the performance of McNumJS compared to native C and other popular JavaScript libraries.

## 5.1 Methodology

### 5.1.1 Measurement objectives

One of the goal for McNumJS library is to improve performance for numerical applications. We study the performance of McNumJS by comparing it with that of native C. We also report the slowdowns we get by adding properties on typed arrays. In addition to using typed arrays, asm.js code enables ahead-of-time compilation or improves just-in-time compilation, thus providing best performance in JavaScript. So it is important to compare McNumJS with asm.js and analyze the performance. There are several popular JavaScript libraries for numerical computation like NumericJS, Sylvester and Google Closure. These libraries are based on regular JavaScript arrays.

The following research questions will help us to understand both the performance of McNumJS library and the slowdown introduced by the extra calculations done in typed array constructor:

**Q1:** Is the performance of McNumJS competitive with native C? (Section 5.2.1)

**Q2:** How much performance do we lose in McNumJS compared to typed arrays? (Section 5.2.2)

**Q3:** How does McNumJS fare compared to the asm.js? (Section 5.2.3)

**Q4:** Does McNumJS provide performance improvement compared to regular arrays? (Section 5.2.4)

**Q5:** How is the performance of McNumJS versus other JavaScript numerical libraries? (Section 5.3)

In the following sections, we describe the experimental setup to answer the research questions and provide details on measurement methods for minimizing standard deviation. We will also discuss briefly about the benchmarks and their different implementations used to perform these experiments.

### 5.1.2 Experimental setup

To perform the experiments, we have used a consumer-grade desktop environment. Detailed specifications of the desktop machine is listed in Table 5.1.

| | Desktop |
|---|---|
| CPU | Intel Core i7, 3.20GHz × 12 |
| Cache | 12 MiB |
| Memory | 16 GiB |
| OS | Ubuntu 12.04 LTS |
| GCC | 4.6.4 |
| Emscripten | 1.12.0 |
| Chrome | 40.0.2214 (Dev version) |
| Firefox | 36.0a1 (Nightly version) |

**Table 5.1** Specifications of the machine used for experiments.

### 5.1.3 Measurements

We can use the Ostrich benchmark suite [25] to answer the first four questions. Colella identified seven algorithmic patterns that occur in numerical computations which are referred to as Dwarfs [30]. The Ostrich benchmark suite [31] covers 12 dwarfs containing one benchmark for each dwarf. The suite comprise implementations in C, JSTA[1], OpenCL and WebCL. Table 5.2 briefly describes each benchmark from Ostrich benchmark suite and mentions which dwarf it belongs to.

We manually developed these benchmarks which make use of McNumJS library and compare them with the existing implementations of the benchmarks. To answer the first question, we used C implementation from the suite and compare its performance with that of McNumJS. We have used GCC compiler with highest optimization level to compile C code. We ran all the benchmarks 10 times and computed the arithmetic mean of the execution times. We calculate the execution time ratio against McNumJS and report the geometric mean of the ratios.

To find the cost of computing extra properties in McNumJS typed arrays, we used JSTA implementation from the benchmark suite and compare the performance with McNumJS benchmarks. The execution times are measured with the high-resolution timer *performance.now* in both the implementations.

To answer the third question, we need the asm.js version of benchmarks. We obtained asm.js code by using emscripten [32]. Emscripten is a compiler that translates LLVM [33] bytecode to asm.js. We can compile the C implementation to LLVM using the Clang compiler and then compile LLVM code to asm.js. Thus, we can compare compiled asm.js code with McNumJS benchmarks and answer the third question. To ensure correctness of the benchmarks, we are using O2 optimization level while using emscripten, instead of using highest level of optimizations. The highest level of optimization(O3) changed the results significantly in some of the benchmarks like back-propagation. The asm.js benchmarks uses *Date.now* while McNumJS uses high-resolution timer *performance.now* to measure the execution time.

We changed the JavaScript version of the benchmarks and obtain one dimensional regular array implementation of the benchmarks. We compare this implementation with McNumJS and answer our fourth question.

[1] JSTA = JavaScript with Typed Arrays

| Benchmark | Dwarf | Description |
|---|---|---|
| *back-prop* | Unstructured grid | a machine-learning algorithm that trains the weights of connecting nodes on a layered neural network |
| *bfs* | Graph traversal | a breadth-first search algorithm that assigns to each node of a randomly generated graph its distance from a source node |
| *crc* | Combinatorial logic | an error-detecting code which is designed to detect errors caused by network transmission or any other accidental error |
| *fft* | Spectral methods | the Fast Fourier Transform (FFT) function is applied to a random data set |
| *hmm* | Graphical models | a forward-backward algorithm to find the unknown parameters of a hidden Markov model |
| *lavamd* | N-body methods | an algorithm to calculate particle potential and relocation due to mutual forces between particles within a large 3D space |
| *lud* | Dense linear algebra | a LU decomposition is performed on a randomly-generated matrix |
| *nqueens* | Branch and bound | 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 | an algorithm to compute the optimal alignment of two protein sequences |
| *page-rank* | Map reduce | the algorithm famously used by Google Search to measure the popularity of a web site |
| *spmv* | Sparse linear algebra | an algorithm to multiply a randomly-generated sparse matrix with a randomly generated vector |
| *srad* | Structured grid | a diffusion method for ultrasonic and radar imaging applications based on partial differential equations |

**Table 5.2** Ostrich Benchmarks: dwarfs and description.

However, these benchmarks only contain primitive operations and thus, cannot use existing library functions provided by other JavaScript libraries for numerical computations. So to answer the fifth question, we need different set of benchmarks. We will use 6 micro-benchmarks based on linear algebra implemented in McNumJS, NumericJS, Closure library and Sylvester library. Table 5.3 briefly describes these six benchmarks. We will use same desktop machine to run these benchmarks.

| Benchmark | Description |
|---|---|
| *abs(V)* | Finding absolute value of vector elements |
| *I(M)* | Creating Identity matrix |
| *Transpose(M)* | Transpose a matrix |
| *Sum(M)* | Sum of all the elements of a matrix |
| $M .* S$ | Matrix-Scalar dot product |
| $M .+ M$ | Matrix-Matrix point-wise addition |

**Table 5.3** Micro-Benchmarks: Name and description.

## 5.2 Ostrich Results

In this section, we are going to compare the performance of McNumJS with that of other Ostrich benchmarks implementations to answer the first four questions.

### 5.2.1 McNumJS vs C

Programs written in C or other low-level languages that compile to efficient native code provides maximum performance out of the hardware. Thus it is very important to compare the performance of McNumJS with C code and answer Q1. We start by looking at the results in Figure 5.1 obtained on our desktop machine using two popular browsers for Linux, Google Chrome and Mozilla Firefox. We are reporting the slowdown of McNumJS benchmarks compared to C by dividing execution time of McNumJS by the execution time of C. We report the execution times of these benchmarks in Appendix A.

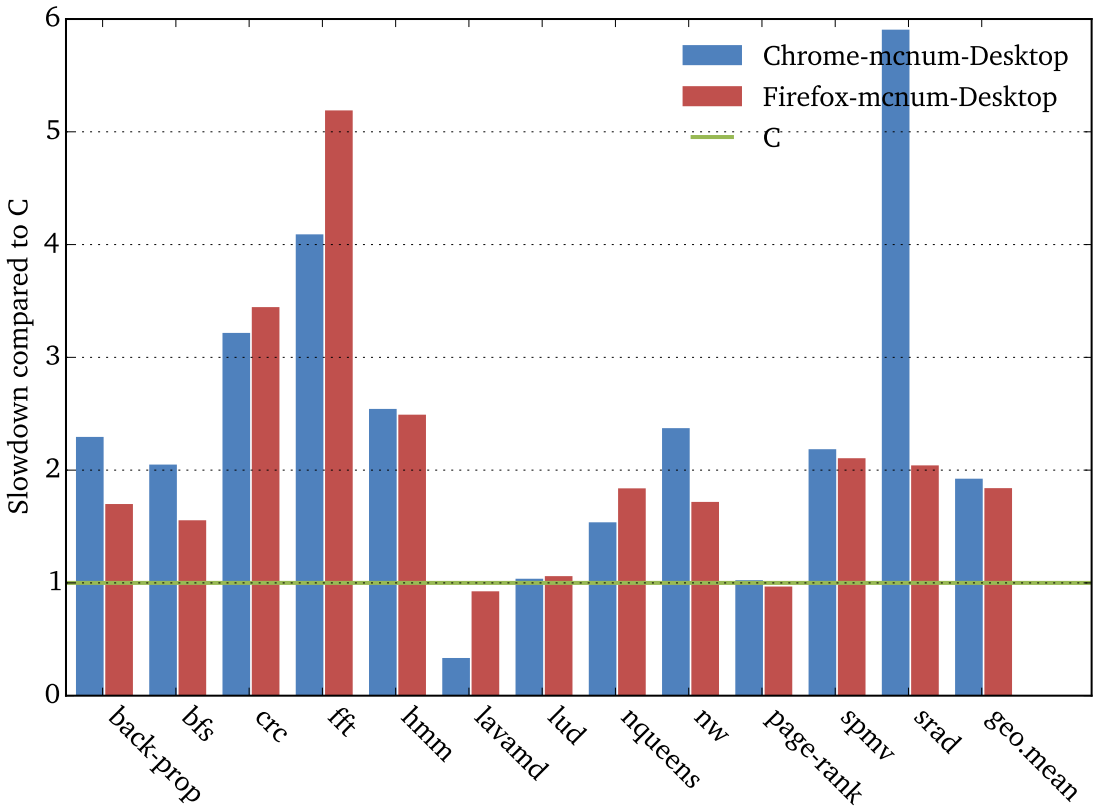


**Fig. 5.1** Slowdown of McNumJS compared to C

In Figure 5.1, we observe that on an average[2] McNumJS is just below 2 times (1.91 times for Chrome and 1.83 times for Firefox) slower than C.

The most interesting of all the benchmarks is *lavamd* which is approximately 3 times faster in McNumJS than C, in Chrome. The execution profile shows that the cause of speedup is a call to the exponential function in a deeply nested loop body. There is an article on Google Developers [34] explaining that a faster approximation of *exp* function was implemented in V8 which is faster than standard system libraries.

Another interesting benchmark is *srad*. This benchmark is 5.89 times slower in Chrome but only 2.03 times slower in Firefox. We are going to explain this behavior in the next section. The only remaining benchmark which is slower than 4x in McNumJS

[2]We use geometric means instead of normal mean. So in the rest of the report, average means geometric mean.

is *fft*. It is 4.08 and 5.18 times slower than C in Chrome and Firefox respectively. This is because multiple typed arrays are allocated inside a recursive function, and in JavaScript typed arrays are initialized to zero, which is more expensive than allocating arrays in C. Hence the severe penalty in the benchmark.

In conclusion, McNumJS performs well for numerical computations, resulting in slowdown within the factor of two in comparison to C.

### 5.2.2 McNumJS vs JavaScript Typed Arrays

McNumJS extends JavaScript typed arrays to provide multi-dimension support. So it is very interesting to see how much slowdown we get by performing the extra calculations of maintaining these properties and providing multi-dimension support to JavaScript typed arrays. Figure 5.2 shows the slowdown of McNumJS benchmarks compared to JSTA implementation by dividing execution time of McNumJS by the execution time of JSTA implementation.

In Figure 5.2, we can see that on an average, McNumJS is 1.12 and 1.002 times slower than JSTA on Chrome and Firefox respectively, making it nearly of similar performance.

The only sore thumb here is the *srad* benchmark which on chrome is 3.12 times slower than JSTA. We could not get the reason behind this weird behavior using Chrome profiler. So we tweaked the benchmark to make it run on D8 (Google Chrome's V8 JavaScript engine, built as a terminal application). We then used V8 profiling tools [34] to profile the benchmark to get the detailed execution profile and discovered the problem. The underlying data in the benchmark was created using Float64Array typed arrays and stored in row-major fashion. However, the access pattern in the rest of the benchmark was in column-major fashion. Thus, the profiling data suggests that the get and set function calls for accessing indices, make it hard for Chrome to optimize array bounds checks. This was simple in the JSTA implementation as there were no calls to function for accessing index, and the index was determined only once in the loop body. To double check the deduction, we incrementally changed the get and set function calls to manual index mapping. We observed that there is exponential speedup as we increase the manual index mapping, which suggests that there is a missed optimization opportunity for array bounds check in Chrome. This is the same reason, why we see big

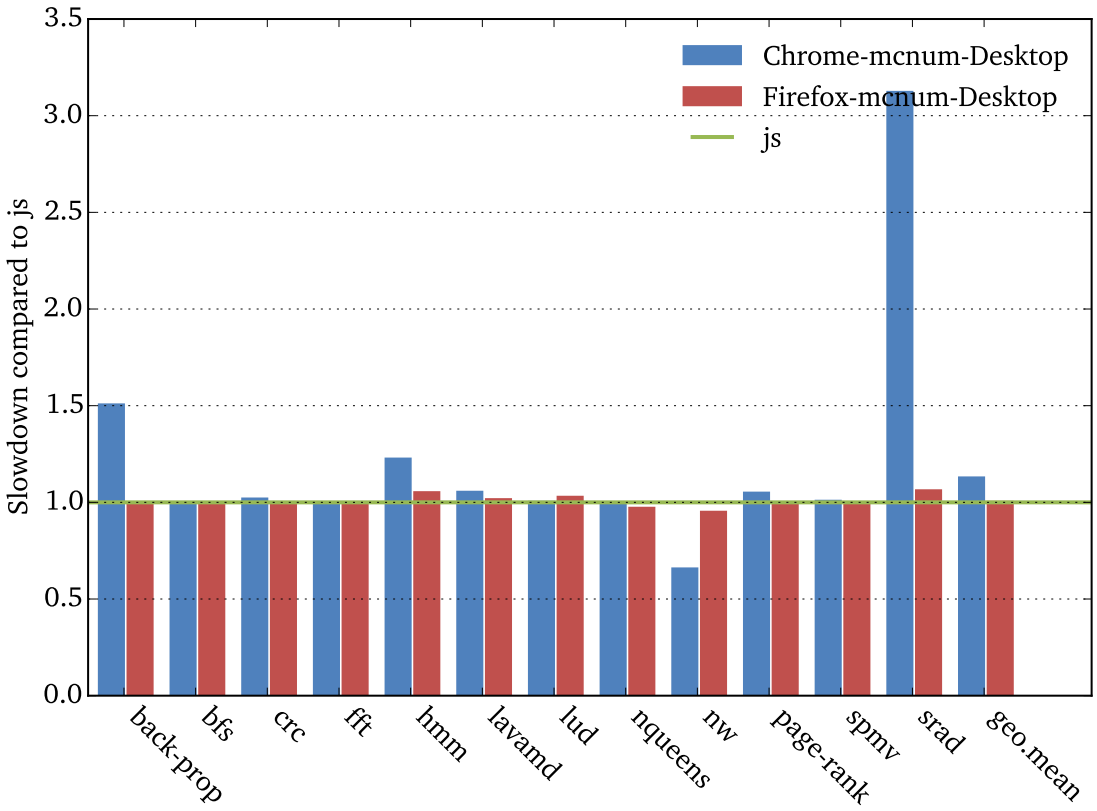


**Fig. 5.2** Slowdown of McNumJS compared to JavaScript Typed Arrays

difference between the Chrome and Firefox result for *srad* in Figure 5.1.

Another interesting benchmark in Figure 5.2 is *nw*. This benchmark is 1.519 and 1.051 times faster in McNumJS than JSTA. This benchmark contains a function for mapping 2-dimensional index to one-dimensional index. Therefore, both of the implementations (McNumJS and JSTA) are similar except the fact that McNumJS has get and set functions added to the typed array object prototype and the arguments of these functions are implicitly typed, while in JSTA the mapping function is separate from typed arrays and arguments to this function are not typed. Thus, McNumJS provides better performance than JSTA.

In Figure 5.2, there are many benchmarks that do not show any change. It is important to note that six of the benchmarks, namely *bfs, crc, fft, lavamd, nqueens* and *spmv*

contains just one-dimensional arrays. So there is no mapping of indices in JSTA and the cost of calculating extra properties in McNumJS is negligible in this case. As the figure shows, these benchmarks show little or no change.

In conclusion, we can say that in McNumJS, the cost of calculating extra properties to support multi-dimensional index and calls to get and set methods for accessing index is insignificant in comparison to the manual mapping in JSTA.

### 5.2.3 McNumJS vs Asm.js

Asm.js defines low-level, strict JavaScript subset, allowing ahead-of-time compilation in Firefox and better JIT optimizations in Chrome and other supported browsers. Therefore, asm.js mostly provides better performance than hand-written JavaScript. It is very interesting to see the performance of McNumJS in comparison to the compiler generated asm.js code. In this section, we will discuss the slowdown of McNumJS compared to asm.js code by dividing execution time of McNumJS by the execution time of asm.js.

In Figure 5.3, the average slowdown of McNumJS is 1.37 and 1.26 times that of asm.js code in Chrome and Firefox respectively. We can observe that only 3 benchmarks out of 24 have slowdowns of more than 2x and 17 out of 24 benchmarks have slowdowns under 1.5x.

Similar to the previous results, there is a huge difference between Chrome and Firefox performance of *srad* benchmark. The benchmark is 4.36 and 1.59 times slower than asm.js in Chrome and Firefox respectively. This was expected as the McNumJS benchmark runs slower in Chrome than Firefox.

It is noticeable that there is a huge speed up of 2.08 for *lavamd* benchmark in Firefox. The reason behind it is that the asm.js version of the benchmark performs badly in Firefox.

In conclusion, we can say that McNumJS performs well only by slowing down below the factor of 1.4 in comparison to the asm.js code.

### 5.2.4 McNumJS vs One dimensional Regular Arrays

JavaScript typed arrays are not popular in routine web application development. Scientists and engineers usually work with regular arrays as they are easy to work with.

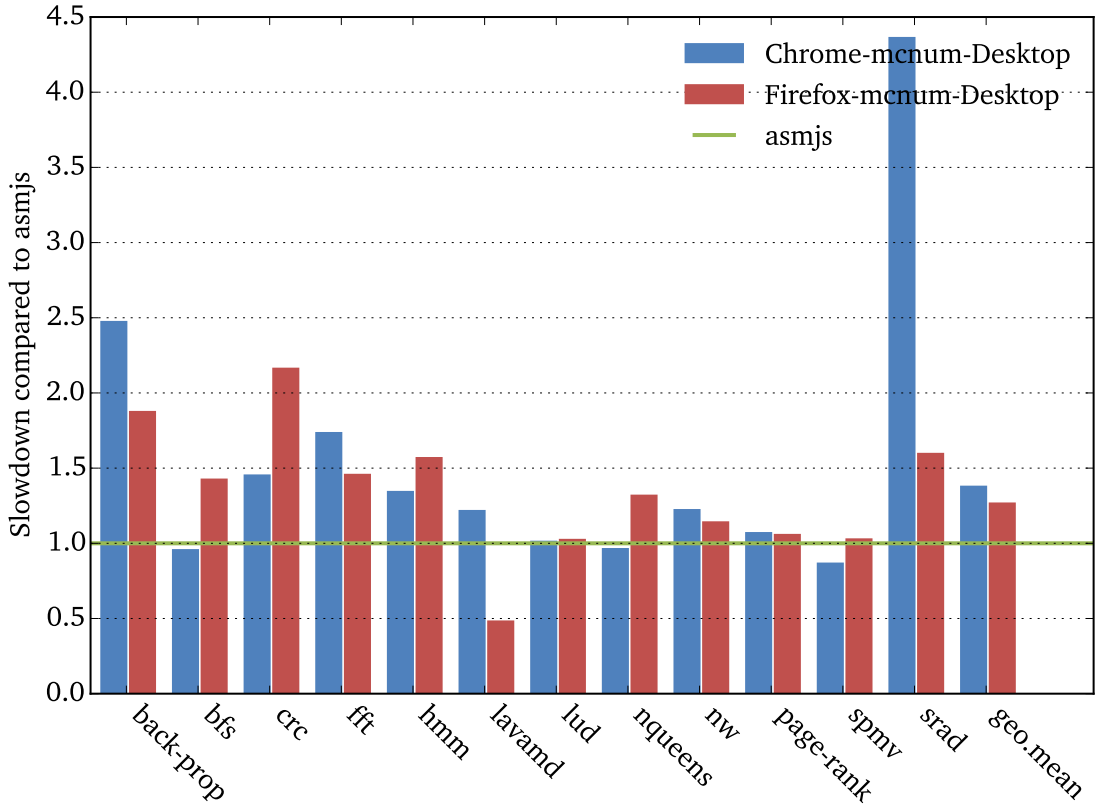


**Fig. 5.3** Slowdown of McNumJS compared to asm.js

However, we claim to make McNumJS to be of high-performance. So it is important to see how much speedup we get using McNumJS compared to using regular arrays.

Figure 5.4 shows the speedup graph of McNumJS compared to one-dimensional regular arrays. We obtained the regular arrays by changing typed array constructor to regular array constructor. We have kept some of the typed arrays to ensure correctness of the algorithm. For example, operations with unsigned integers would require additional conditions using regular array to ensure correctness. Thus, we have not changed these typed arrays to regular arrays. As the graph is a speedup graph, higher value is desirable. Also note that the graph is on log base of 4.

In Figure 5.4, average speedup we get by using McNumJS is 1.25 and 2.01 times that of the regular arrays in Chrome and Firefox respectively. This is a huge difference.

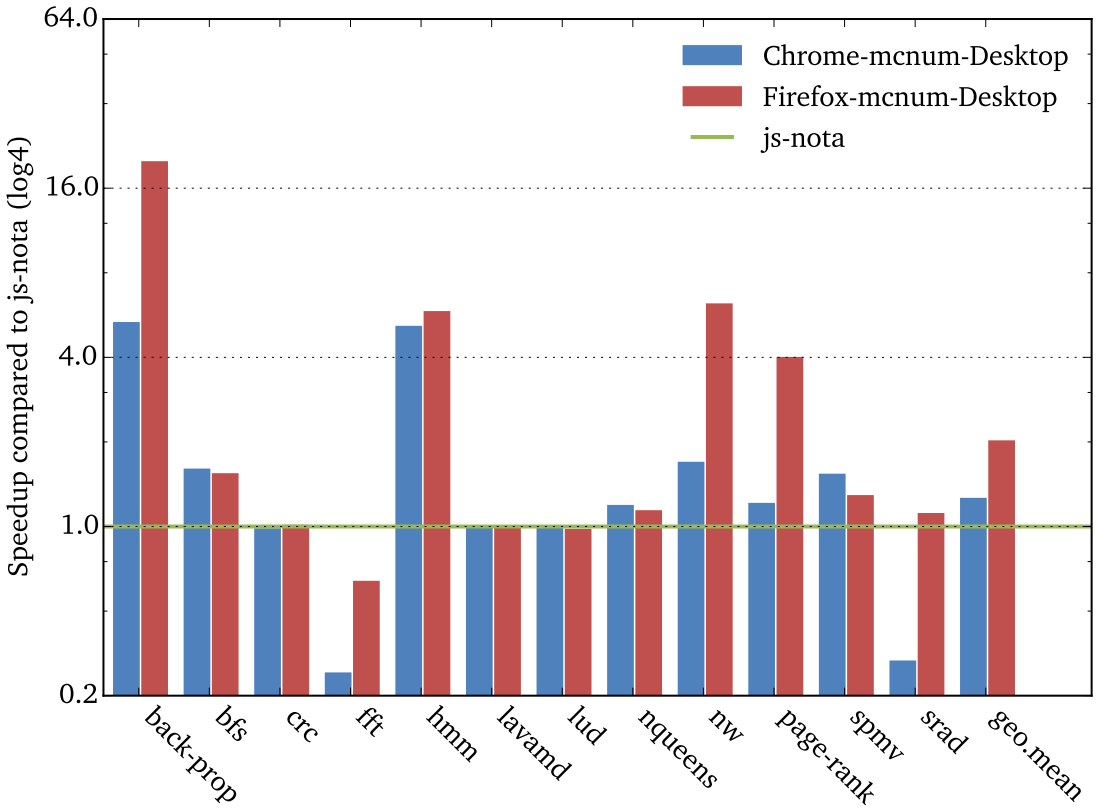


**Fig. 5.4** Speedup of McNumJS compared to One dimensional Regular Arrays

We will discuss the reason behind this gap later in this section.

As shown in the figure, we get major slowdown for *fft* benchmark. As discussed previously, *fft* creates multiple typed arrays in a recursive function, and because typed arrays takes more time to initialize than regular arrays, we get slowdown in McNumJS compared to regular arrays.

Another interesting benchmark in Figure 5.4 is *srad*, as we get speedup in Firefox and slowdown in Chrome. As mentioned before, there is a missed optimization opportunity in Chrome which makes the McNumJS implementation of the benchmark slower.

We can notice that for *back-prop, nw* and *page-rank*, there is a vast difference between the speedups of Chrome and Firefox. Firefox shows a huge speedup for McNumJS. The reason behind this is that Chrome optimizes the regular arrays better than Firefox.

Chrome internally converts regular arrays to arrays of doubles within hot loops, making it highly optimizable and faster to access. Also, Chrome has a missed optimization opportunity for some of these benchmarks.

As mentioned earlier, we converted JSTA version to regular arrays by changing the typed array constructor. Also, we have kept some of the typed arrays unchanged as they were required to ensure the correctness of the algorithm. If we use multi-dimensional regular arrays, some of the benchmarks run out of memory and some of them continue running more than three minutes. Thus, if scientists were to develop numeric application in JavaScript, they cannot use multi-dimensional regular arrays due to its limitations. In conclusion, McNumJS shows good speedup over one-dimensional regular arrays.

## 5.3 Performance compared to other libraries

In this section, we are going to use micro-benchmarks described in Table 5.3 to find how much performance we get using McNumJS compared to other popular JavaScript libraries for numerical computations as described in Chapter 2.

Micro-benchmarks have lower execution time compared to normal benchmarks. The execution time of these benchmarks are often misleading. We cannot account the times for context switches, garbage collections, library load, which occurs at any random time and take any nondeterministic amount of time. If we execute a micro-benchmark, and any one of these times add up, it will make an significant change in the execution time. Thus, to safely deduce the performance result for micro-benchamarks, we run these benchmarks repeatedly over a considerable period of time and calculate MEPS (Micro-benchmark Executions Per Second), the number of times the benchmark has run. Thus, we can safely ignore the other timings.

In this section, we calculate MEPS for all the micro-benchmarks and compare the performance of McNumJS with other libraries by dividing the MEPS of McNumJS to the MEPS of other libraries. Thus we show the slowdowns of the other libraries compared to McNumJS. We used four different input sizes to test the result. We ran these micro-benchmarks in Chrome and Firefox. We reported MEPS for all of these benchmarks in Appendix B.

Figure 5.5 and Figure 5.6 shows the slowdown of popular JavaScript libraries compared to McNumJS. So the higher numbers indicate better performance for McNumJS. The single line connecting four dots are slowdowns of a single benchmark, for different input sizes. The input size represents the number of elements in vector or row/column in matrix. We tested four different input sizes: 50, 200, 800 and 3200. We can see that for large inputs, McNumJS outperforms the other libraries. For smaller input size, McNumJS is just behind NumericJS for just two benchmarks. Google Closure Library and Sylvester library uses objects to represent the matrices or vectors. So we can expect that they do not perform as well as other libraries as objects are not very optimizable. Also, the primary focus of these two libraries is not good performance but to provide functionality.

Firefox shows a huge slowdown for NumericJS for large input size compared to McNumJS. As discussed in previous section, Firefox is not good with optimizing regular arrays but it is good with optimizing typed arrays. Thus we get huge speedup for McNumJS in Firefox.

On average, McNumJS is 1.76, 13.46 and 9.02 times faster than NumericJS, Closure library and Sylvester library in Chrome respectively. Similarly, on Firefox, McNumJS is 5.04, 12.34 and 8.35 times faster that NumericJS, Closure library and Sylvester library. We can see that as the input size increases, the speedup gets higher for McNumJS.

It is important to note that for transpose, as we do not copy the data and just create a new view object with updated shape and stride properties, transpose should be much faster. However, other libraries creates a new object when performing a transpose, it is only fair if we clone the object first and then perform transpose in McNumJS. Thus, in transpose benchmark, we also copy the data. We should get more speedup if we do not copy the data.

In conclusion, McNumJS outperforms the other libraries by at least a factor of 1.8.

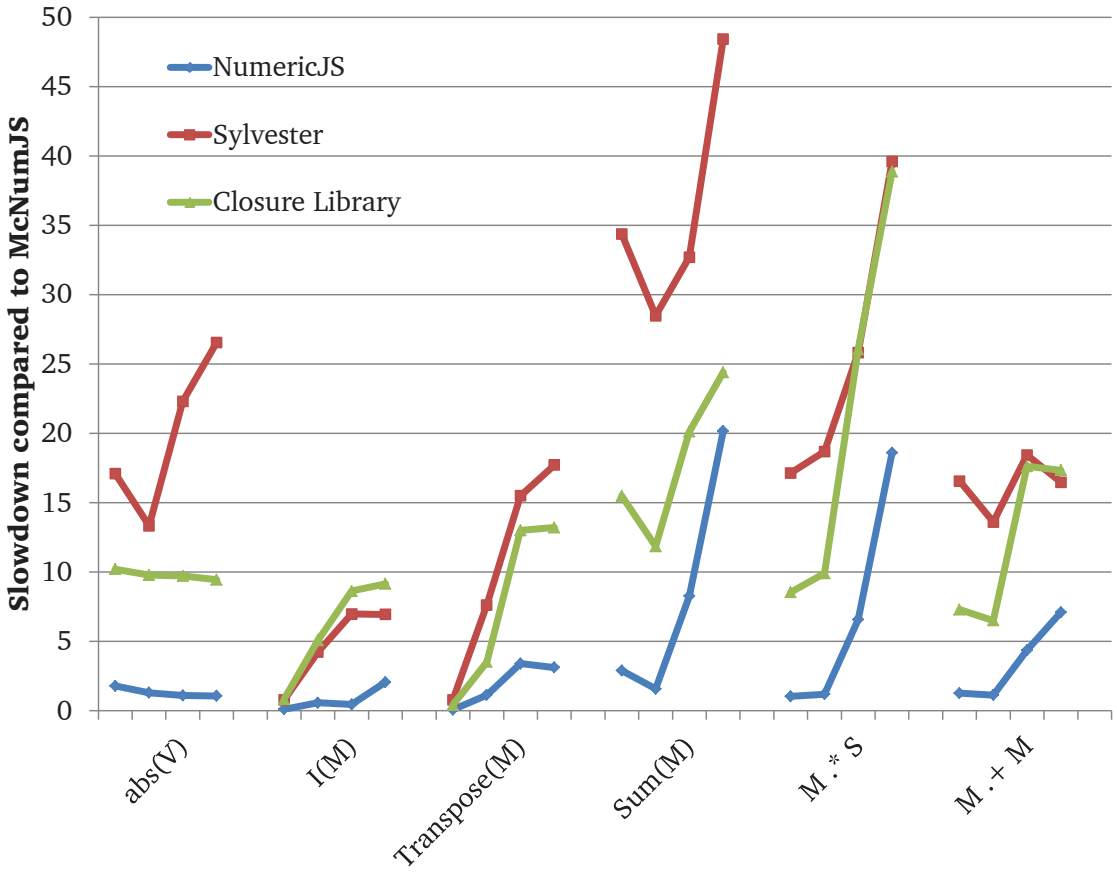


**Fig. 5.5** Slowdown of JavaScript libraries compared to McNumJS in Chrome (V = Vector, M = Matrix, S = Scalar)

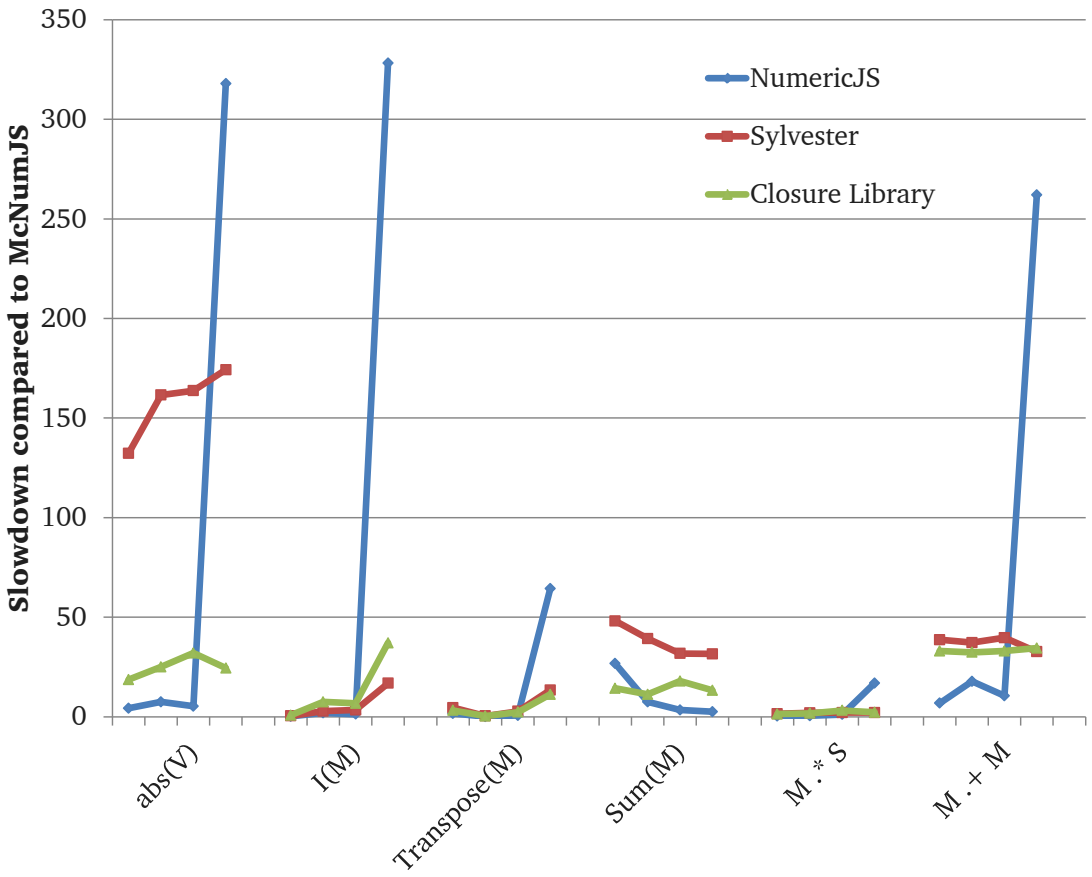


**Fig. 5.6** Slowdown of JavaScript libraries compared to McNumJS in Firefox

# Chapter 6

# Conclusions and Future Work

## 6.1 Conclusion

JavaScript allows applications to run on any device with web browsers and makes the distribution of the applications easier than ever before. With the increasing popularity of JavaScript, a lot of research has been done to improve the performance of JavaScript and minimize the quirks that the language has. However, the lack of high performance numeric library for JavaScript, makes it difficult to develop numeric applications that run with high temporal efficiency. With this thesis, we aim to solve the problem by providing a high-performance library for numerical computations.

To provide high-performance, we investigated existing and currently in development JavaScript technologies that can provide good performance and are suitable for numerical applications. In Chapter 2, we listed these technologies with their advantages and disadvantages, and provided examples by implementing sepia function in each of these technologies. Moreover, to make our library easy-to-use, we studied other popular numeric libraries and kept them as reference.

We then provided an in-depth analysis of asm.js with different JavaScript modes and asm.js calling patterns. Even though the main motivation behind asm.js was to make it compilation target, we discovered that the type coercion rules define by asm.js are easy to work with and provide performance improvements. We also explored the limitations of asm.js. For example, calling asm.js from normal JavaScript or another asm.js module changes the execution engine, which greatly reduces the performance.

McNumJS makes use of typed arrays to get faster array access. We extend the view object, and add shape and stride information to provide multi-dimensional array access. In Chapter 4, we provide cost analysis of maintaining these properties and benefits from them. McNumJS also uses asm.js style type coercing to get even better performance. In the case of transpose and slicing, McNumJS avoids copying data by just changing the shape and stride properties of the McNumJS typed arrays.

McNumJS is developed in four modules. The *core module* extends the typed arrays and provides multi-dimension array access. This module also provide methods like reshape, to change the shape of the array. This will create a new view object with new values of properties on the same data. The *generator module* provides functions to generate different matrices like zeros and ones. We can optionally specify type to create different type of arrays supported by JavaScript like Uint32Array or Float32Array. The *unary* and *binary operator modules* provide functions for unary and binary operations on matrices like add, subtract, sin and cos.

This thesis analyzes the performance of McNumJS by comparing it with C and other JavaScript libraries. We also check the slowdowns we get by maintaining extra properties to support multi-dimensional arrays. Our experiments show that McNumJS is slower than C within a factor of 2. The slowdowns we get are fairly negligible. However, in some cases, Chrome shows huge slowdowns for McNumJS. McNumJS provides good speedup in comparison to the regular arrays. We also showed that it performs very well in comparison to the other JavaScript libraries.

## 6.2 Future Work

Based on the deduced knowledge and results, we recognize some of the areas in which future research or development initiatives can be undertaken to make the API robust and more efficient. We categorize possible future works as follows:

**Adding more Library functions:** The current implementation of the library consists of generation of multi-dimensional matrices and basic arithmetic operations on them. However, for scientists and engineers to write numerical applications, these functions are not sufficient. Keeping this library as a base and providing more library functions should be fairly simple. Fourier transform and LU-Decomposition

are examples of some of the functions that are required to be implemented. Moreover, efficient implementation of complex numbers in JavaScript is also an interesting feature to develop.

**Efficient API implementation:** As more and more research is being done in the area of JavaScript performance, there are more chances of optimizing the library and providing better performance. Also, as we have seen in Chapter 5, there are some missed optimization opportunities for the Google Chrome. An efficient implementation is required to overcome such aberration in performance.

**Implementing lazy computation:** It is advantageous to combine non-conflicting operations and compute only when the result is required. There are many optimization opportunities if we combine the operations. However, the current implementation does not provide any function or way to combine operations or delay computation till the results are required. NumPy library uses lazy computation to optimize the computations. Thus, it is an interesting research topic to find a way to provide lazy computation in McNumJS.

**Making use of Parallelism:** There are several research projects going on to support parallelism in JavaScript. We also investigated the possibility of using Web Workers to provide parallelism but the Web Workers are not light-weight and they introduce slowdowns. We also studied the performance of WebCL and found that it provides good speedups only for highly parallel benchmarks. However, as the research progress, we can see the possibility of extending the library and making use of parallelism.

**Performance tuning for more JavaScript engines:** We have studied the performance of McNumJS in Google Chrome and Mozilla Firefox browsers. These browsers take around 60% of the browser usage market share [35]. However, Internet Explorer and Safari also have a fairly large user base. So it is important to tune the performance of the library for these browsers as well.

**Minimization (or Minification):** Minimization is the process of removing all unnecessary characters from source code without changing its functionality [36]. Minification is very important in JavaScript as it reduces the size of the code and the

reduced size takes less time to transfer over the Internet [37]. McNumJS uses type coercion rules and that requires some binary operations and some extra parentheses. Currently available JavaScript compressor tools like Google Closure tools [38] and UglifyJS [39] removes these extra parenthesis. Thus, we need to create a tool which recognizes asm.js code and do not remove parentheses.

# Appendix A

# Ostrich Benchmark Suite Results

In this appendix, we present the timing results from our experiments.

### A.0.1 Execution times

The tables below show the average times (in seconds) for 10 execution of each benchmark in each software configuration.[1]

| Benchmark | C | Fx-JS-noT | Fx-asmjs | Fx-JS-TA | Fx-mcnum |
|---|---|---|---|---|---|
| back-prop | 0.903 | 30.220 | 0.815 | 1.544 | 1.526 |
| bfs | 0.333 | 0.790 | 0.362 | 0.518 | 0.514 |
| crc | 0.643 | 2.224 | 1.023 | 2.213 | 2.210 |
| fft | 0.795 | 2.619 | 2.831 | 4.148 | 4.120 |
| hmm | 1.804 | 25.966 | 2.866 | 4.256 | 4.480 |
| lavamd | 2.435 | 2.229 | 4.648 | 2.195 | 2.231 |
| lud | 1.950 | 1.994 | 2.003 | 1.992 | 2.049 |
| nqueens | 2.947 | 6.113 | 4.097 | 5.547 | 5.392 |
| nw | 0.442 | 4.670 | 0.664 | 0.795 | 0.756 |
| page-rank | 3.373 | 12.881 | 3.064 | 3.255 | 3.235 |
| spmv | 0.679 | 1.827 | 1.393 | 1.442 | 1.424 |
| srad | 3.902 | 8.787 | 4.974 | 7.467 | 7.933 |

**Table A.1** Ostrich benchmark suite results on C and Firefox

[1] Cr = Chrome; Fx = Firefox; JS-TA = JavaScript with typed arrays; JS-noTA = JavaScript with regular arrays

| Benchmark | Cr-JS-noTA | Cr-asmjs | Cr-JS-TA | Cr-mcnum |
|---|---|---|---|---|
| back-prop | 10.935 | 0.835 | 1.370 | 2.064 |
| bfs | 1.082 | 0.712 | 0.679 | 0.679 |
| crc | 2.016 | 1.422 | 2.023 | 2.063 |
| fft | 0.974 | 1.873 | 3.250 | 3.246 |
| hmm | 23.484 | 3.408 | 3.728 | 4.572 |
| lavamd | 0.786 | 0.722 | 0.751 | 0.792 |
| lud | 1.986 | 1.983 | 1.998 | 2.004 |
| nqueens | 5.332 | 4.683 | 4.552 | 4.504 |
| nw | 1.763 | 0.857 | 1.589 | 1.046 |
| page-rank | 4.122 | 3.206 | 3.260 | 3.424 |
| spmv | 2.260 | 1.708 | 1.466 | 1.478 |
| srad | 7.612 | 5.277 | 7.367 | 23.008 |

**Table A.2** Ostrich benchmark suite results on Chrome

# Appendix B

# Micro-benchmarks Results

In this appendix, we present the MEPS results from our experiments with micro-benchmarks.

| benchmark | size | NumericJS | Google Closure | Sylvester | McNumJS |
|---|---|---|---|---|---|
| abs(V) | n=50 | 3071968.8 | 99064.51 | 702142.86 | 13107167 |
| | n=200 | 571511.63 | 26448.27 | 170638.89 | 4274065.2 |
| | n=800 | 211827.59 | 6821.42 | 34863.63 | 1117068.2 |
| | n=3200 | 920.00 | 1678.57 | 11968.75 | 292547.62 |
| I(M) | n=50 | 85305.55 | 33347.82 | 18707.31 | 14730.76 |
| | n=200 | 3745.09 | 2638.88 | 979.16 | 7346.15 |
| | n=800 | 407.40 | 151.51 | 78.12 | 523.80 |
| | n=3200 | 0.41 | 8.03 | 3.64 | 135.13 |
| Transpose(M) | n=200 | 2159.09 | 793.10 | 1146.34 | 3653.84 |
| | n=50 | 74902.43 | 15320.00 | 17431.81 | 6586.20 |
| | n=800 | 185.18 | 44.44 | 52.63 | 121.95 |
| | n=3200 | 0.31 | 1.50 | 1.79 | 20.20 |
| Sum(M) | n=50 | 19175.00 | 10638.88 | 35697.67 | 511958.33 |
| | n=200 | 3653.84 | 696.96 | 2435.89 | 27392.85 |
| | n=800 | 407.40 | 43.47 | 76.92 | 1382.35 |
| | n=3200 | 31.74 | 2.51 | 5.97 | 79.36 |
| M .* S | n=50 | 59038.46 | 10638.88 | 11606.06 | 16078.53 |
| | n=200 | 2768.11 | 696.96 | 793.10 | 1382.35 |
| | n=800 | 78.12 | 42.55 | 28.57 | 90.90 |
| | n=3200 | 0.31 | 2.53 | 2.46 | 5.42 |
| M .+ M | n=50 | 52931.03 | 9341.46 | 10942.85 | 361382.35 |
| | n=200 | 1270.27 | 605.26 | 696.96 | 22558.82 |
| | n=800 | 139.24 | 37.03 | 44.44 | 1468.75 |
| | n=3200 | 0.27 | 2.18 | 2.07 | 71.42 |

**Table B.1** Micro-benchmark results on Firefox

| Benchmarks | Size | Numeric | Google Closure | Sylvester | McNumJS |
|---|---|---|---|---|---|
| abs(V) | n=50 | 3780884.6 | 396354.84 | 664189.19 | 6779551.7 |
| | n=200 | 1404314.3 | 136511.11 | 186151.52 | 1820407.4 |
| | n=800 | 396354.84 | 19666.66 | 45147.06 | 438821.43 |
| | n=3200 | 102366.67 | 4130.43 | 11606.06 | 109678.57 |
| I(M) | n=50 | 198161.29 | 27392.85 | 25566.66 | 21305.55 |
| | n=200 | 13678.57 | 1880.00 | 1566.66 | 7958.33 |
| | n=800 | 920.00 | 58.82 | 47.61 | 410.71 |
| | n=3200 | 12.57 | 3.73 | 2.83 | 25.97 |
| Transpose(M) | n=50 | 136511.11 | 12766.66 | 24741.93 | 10078.94 |
| | n=200 | 4547.61 | 676.47 | 1468.75 | 5162.16 |
| | n=800 | 147.05 | 32.25 | 38.46 | 500.00 |
| | n=3200 | 9.56 | 1.68 | 2.25 | 29.85 |
| Sum(M) | n=50 | 73119.04 | 6161.29 | 13678.57 | 211827.59 |
| | n=200 | 7346.15 | 407.40 | 979.16 | 11606.06 |
| | n=800 | 86.95 | 21.97 | 35.71 | 718.75 |
| | n=3200 | 3.30 | 1.37 | 2.73 | 66.66 |
| M .* S | n=50 | 99064.51 | 5968.75 | 11968.75 | 102366.67 |
| | n=200 | 6161.29 | 392.85 | 741.93 | 7346.15 |
| | n=800 | 71.42 | 18.18 | 18.01 | 469.38 |
| | n=3200 | 2.75 | 1.29 | 1.31 | 51.28 |
| M .+ M | n=50 | 73119.04 | 5617.64 | 12766.66 | 93060.60 |
| | n=200 | 4547.61 | 379.31 | 793.10 | 5162.16 |
| | n=800 | 74.07 | 17.54 | 18.34 | 323.52 |
| | n=3200 | 2.81 | 1.21 | 1.15 | 20.00 |

**Table B.2** Micro-benchmark results on Chrome

# References

[1] C. Severance, "Javascript: Designing a Language in 10 Days," *Computer*, vol. 45, no. 2, pp. 7–8, 2012.

[2] "Develop High Performance Windows 8 Application with HTML5 and JavaScript." http://blogs.msdn.com/b/dorischen/archive/2013/04/26/develop-high-performance-windows-8-application-with-html5-and-javascript-best-practices-amp-tips.aspx, Apr. 2013.

[3] S. Tilkov and S. Vinoski, "Node.js: Using JavaScript to Build High-Performance Network Programs," *IEEE Internet Computing*, vol. 14, pp. 80–83, Nov. 2010.

[4] "Standard ECMA-262 ECMAScript® Language Specification Edition 5.1." http://www.ecma-international.org/publications/standards/Ecma-262.htm, June 2011.

[5] A. Gal, B. Eich, M. Shaver, D. Anderson, *et al.*, "Trace-based Just-in-time Type Specialization for Dynamic Languages,"

[6] L. Bak, "Google Chrome's Need for Speed." http://blog.chromium.org/2008/09/google-chromes-need-for-speed_02.html, Sept. 2008.

[7] "'Epic Citadel' Demo Shows the Power of the Web as a Platform for Gaming." https://blog.mozilla.org/futurereleases/2013/05/02/epic-citadel-demo-shows-the-power-of-the-web-as-a-platform-for-gaming/, May 2013.

[8] "Asm.js Specification." http://asmjs.org/spec/latest/.

[9] "WebGL Specification." https://www.khronos.org/registry/webgl/specs/1.0/, Mar. 2013.

[10] "WebCL Specification." http://www.khronos.org/registry/webcl/specs/latest/1.0/, May 2014.

[11] Numpy.org, http://www.numpy.org/, *NumPy*.

[12] L. Wagner, "Asm.js AOT compilation and startup performance." https://blog.mozilla.org/luke/2014/01/14/asm-js-aot-compilation-and-startup-performance/, Jan. 2014.

[13] "Chrome and Opera Optimize for Mozilla-Pioneered Asm.js." https://blog.mozilla.org/futurereleases/2013/11/26/chrome-and-opera-optimize-for-mozilla-pioneered-asm-js/, Nov. 2013.

[14] "Web Workers." http://www.w3.org/TR/workers/, May 2012.

[15] S. Okamoto and M. Kohana, "Load Distribution by Using Web Workers for a Real-time Web Application," in *Proceedings of the 12th International Conference on Information Integration and Web-based Applications; Services*, iiWAS '10, (New York, NY, USA), pp. 592–597, ACM, 2010.

[16] "Nokia WebCL for Firefox." http://www.khronos.org/registry/webcl/specs/latest/1.0/, May 2014.

[17] "WebCL Chromium." https://www.khronos.org/registry/webgl/specs/1.0/, Mar. 2013.

[18] S. Loisel, "Numeric Javascript." http://www.numericjs.com/.

[19] Google, "Closure Library API Documentation JavaScript." http://docs.closure-library.googlecode.com/git/index.html.

[20] "Integer - Closure Library." http://docs.closure-library.googlecode.com/git/class_goog_math_Integer.html.

[21] J. Coglan, "Sylvester." http://sylvester.jcoglan.com/.

[22] "Strict mode - JavaScript | MDN." https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode.

[23] M. Bynens, "jsPerf: JavaScript performance playground." http://jsperf.com/.

[24] "Asm.js in Firefox Nightly." http://blog.mozilla.org/luke/2013/03/21/asm-js-in-firefox-nightly/.

[25] F. Khan, V. Foley-Bourgon, S. Kathrotia, E. Lavoie, and L. Hendren, "Using JavaScript and WebCL for Numerical Computations: A Comparative Study of Native and Web Technologies," in *Proceedings of the 10th ACM Symposium on Dynamic Languages*, DLS '14, (New York, NY, USA), pp. 91–102, ACM, 2014.

[26] T. Disney, N. Faubion, D. Herman, and C. Flanagan, "Sweeten Your JavaScript: Hygienic Macros for ES5," in *Proceedings of the 10th ACM Symposium on Dynamic Languages*, DLS '14, (New York, NY, USA), pp. 35–44, ACM, 2014.

[27] Y. Suzuki, "estools / escodegen." https://github.com/estools/escodegen/.

[28] "Grunt: The JavaScript Task Runner." http://gruntjs.com/.

[29] "tap-producing test harness for node and browsers." https://github.com/substack/tape.

[30] 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, (New York, NY, USA), pp. 291–294, ACM, 2012.

[31] "Ostrich benchmark suite." https://github.com/Sable/Ostrich.

[32] 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, (New York, NY, USA), pp. 301–312, ACM, 2011.

[33] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, (Washington, DC, USA), pp. 75–, IEEE Computer Society, 2004.

[34] "Profile your web application with V8's internal profiler." https://developers.google.com/v8/profiler_example.

[35] "W3Counter: Global Web Stats - November 2014 Market Share." http://www.w3counter.com/globalstats.php.

[36] "Minification (programming)." http://en.wikipedia.org/wiki/Minification_(programming).

[37] D. Odell, " Boosting JavaScript Performance ," in *Pro JavaScript Development*, pp. 91–118, Apress, 2014.

[38] "Closure Tools - Google Developers." https://developers.google.com/closure/compiler/.

[39] "UglifyJS - JavaScript parser / mangler / compressor / beautifier library for NodeJS." https://github.com/mishoo/UglifyJS.