McGill University
School of Computer Science
Sable Research Group

Sparse matrices on the web -- Characterizing the performance and optimal format selection of sparse matrix-vector multiplication in JavaScript

Sable Technical Report No. McLAB-2018-04

Prabhjot Sandhu, David Herrera and Laurie Hendren

March 20, 2018

Contents

List of Figures

List of Tables

Abstract

JavaScript is the most widely used language for web programming, and now increasingly becoming popular for high performance computing, data-intensive applications, and deep learning. Sparse matrix-vector multiplication (SpMV) is an important kernel that is considered critical for the performance of those applications. In SpMV, the optimal selection of storage format is one of the key aspects of developing effective applications. This paper describes the distinctive nature of the performance and choice of optimal sparse matrix storage format for sequential SpMV in JavaScript as compared to native languages like C. Based on exhaustive experiments with 2000 real-life sparse matrices, we explored three main research questions. First, we examined the difference in performance between native C and JavaScript for the two major browsers, Firefox and Chrome. We observed that the best performing browser demonstrated a slowdown of only 1.2x to 3.9x, depending on the choice of sparse storage format. Second, we explored the performance of single-precision versus double-precision SpMV. In contrast to C, in JavaScript, we found that double-precision is more efficient than single-precision. Finally, we examined the choice of optimal storage format. To do this in a rigorous manner we introduced the notion of *x%-affinity* which allows us to identify those formats that are at least $x\%$ better than all other formats. Somewhat surprisingly, the best format choices are very different for C as compared to JavaScript, and even quite different between the two browsers.

Sparse matrices on the web -- Characterizing the performance and optimal format selection of sparse matrix-vector multiplication in JavaScript

Prabhjot Sandhu, David Herrera and Laurie Hendren

March 20, 2018

1 Introduction

The rapid proliferation of web-enabled devices, and the recent advances in JavaScript engines in web browsers provides new opportunities for sophisticated and compute-intensive applications [21, 12, 13, 19, 7, 15]. Among those compute-intensive applications, sparse matrices arise frequently, and the operations used for their manipulations are considered important for the performance of those applications. Sparse matrix-vector multiplication (SpMV) is one such critical operation used in many iterative methods for solving linear systems and partial differential equations. It plays an essential role in the performance of those applications as it dominates the overall application runtime through its recurring nature. In its simplest form, SpMV computes $y = Ax$, where matrix A is sparse and vector x is dense. For its fundamental significance, SpMV has become a good candidate for optimization to improve the overall performance of the applications.

In different domains including structural mechanics, fluid dynamics, social network analysis, data mining and deep learning, the matrices are large and sparse with different sparsity characteristics. So, the choice of sparse storage format can make a significant difference for SpMV performance. One single format is not appropriate for all the given input matrices. One way to optimize SpMV is to store the input matrix in the optimal format based on its structure, and then use a specialized SpMV kernel implementation for that format, which reduces the complexity of both space and computation. A number of storage formats have been proposed [9, 14, 17, 23], but the well-known four basic formats that have been extensively used are: COO, CSR, DIA and ELL, from which others formats can be derived.

In this paper we study three main research questions which examine web-based SpMV performance and the choice of best storage format for JavaScript. We aim to provide results which will help guide both the current practices for web-based SpMV, as well as showing opportunities for future improvements. The results are based on a set of rigorous experiments making use of over almost 2000 real-life sparse matrices [6], four different matrix formats (COO, CSR, DIA and ELL), and two major web browsers (Firefox and Chrome).

Our first research question, **RQ1**, examines the relative performance of SpMV in C and JavaScript. The main question is to determine if SpMV on the web is reasonably performant or not. How much slower is SpMV in JavaScript when compared to highly-optimized sequential C code? In performing these experiments we systematically chose the best format for each input matrix, and we examined both single- and double-precision versions of SpMV. The results from **RQ1** showed that JavaScript is reasonably performant, and led to two further important research questions.

Our second research question, **RQ2**, examines the performance differences and format choices for C and JavaScript between single- and double-precision. These results clearly demonstrate that the conventional

wisdom that single precision is more efficient than double precision does not always hold in the JavaScript context. Furthermore, the choice of language (C or JavaScript) and the choice of precision (single or double) both contribute to determining the best choice of format.

Our final research question, **RQ3**, examines the choice of best format in more detail. There are two important aspects of this question. The first is to examine the affinity of the input matrices for a specific format. To do this we introduce a new notion of *x%-affinity*. We say that an input matrix `A` has an *x%-affinity* for storage format `F`, if the performance for `F` is at least $x\%$ better than all other formats (and the performance difference is greater than the measurement error). For example, if input array `A` in format `CSR`, is more than 10% faster than input `A` in all other formats, and 10% is more than the measurement error, then we say that `A` has a *10%-affinity* for CSR. Given this definition we examine the 10%-, 25%- and 50%- affinities for both C and JavaScript, and examine the resulting trends. How is the choice of format impacted by the performance improvement expected? How does this differ for C and JavaScript? The second aspect of the question is to demonstrate the similarities and differences in affinities between C and JavaScript. When is it reasonable to use the same format for both C and JavaScript, and when should a different format be chosen for JavaScript?

The rest of the paper is organized as follows. In Section 2 we provide more detailed background about both sparse matrix storage formats and modern JavaScript engines ,and Section 3 details our experimental design. The three research questions are addressed in Section 4. Finally, Section 5 discusses related work and Section 6 discusses conclusions and future work.

2 Background

In this section we provide background on the four sparse matrix formats that we study in this paper, and we give some historical context and background about JavaScript.

2.1 Sparse Matrix Formats

In this paper we have chosen to study four key sparse matrix representations as shown in Figure 1 with a simple example discussed below. Our reference C implementation for computing SpMV on each format is given in Section 3.4.

Coordinate Format (COO): This is the simplest storage format that consists of three arrays : `row`, `column` and `val` to store the row indices, column indices and values of the non-zero entries respectively. We also assume here that the entries are sorted by row index.

Compressed Sparse Row Format (CSR): This is the most popular and widely used format. It is the compressed version of COO format, where the `row` array is compressed to include only one entry per row. Each row entry points to the location of the first element of that row.

Diagonal Format (DIA): This format only stores the diagonals that include non-zero elements. It consists of two arrays : `data` and `offset` to store the non-zero values and offset of each diagonal from the main diagonal respectively.

ELLPACK Format (ELL): This format consists of two arrays: `data` and `indices` to store the fixed number of non-zeros per row in a 2-dimensional array and corresponding column index for each non-zero entry respectively. The rows with fewer non-zero entries are padded with zero values.

	$A = \begin{bmatrix} 1 & 0 & 6 & 0 \\ 0 & 2 & 0 & 7 \\ 0 & 0 & 3 & 0 \\ 5 & 0 & 0 & 4 \end{bmatrix}$
COO	$row = \begin{bmatrix} 0 & 0 & 1 & 1 & 2 & 3 & 3 \end{bmatrix}$ $col = \begin{bmatrix} 0 & 2 & 1 & 3 & 2 & 0 & 3 \end{bmatrix}$ $val = \begin{bmatrix} 1 & 6 & 2 & 7 & 3 & 5 & 4 \end{bmatrix}$
CSR	$row\_ptr = \begin{bmatrix} 0 & 2 & 4 & 5 & 7 \end{bmatrix}$ $col = \begin{bmatrix} 0 & 2 & 1 & 3 & 2 & 0 & 3 \end{bmatrix}$ $val = \begin{bmatrix} 1 & 6 & 2 & 7 & 3 & 5 & 4 \end{bmatrix}$
DIA	$data = \begin{bmatrix} * & 1 & 6 \\ * & 2 & 7 \\ * & 3 & * \\ 5 & 4 & * \end{bmatrix}$ $offset = \begin{bmatrix} -3 & 0 & 2 \end{bmatrix}$
ELL	$data = \begin{bmatrix} 1 & 6 \\ 2 & 7 \\ 3 & * \\ 5 & 4 \end{bmatrix}$ $indices = \begin{bmatrix} 0 & 2 \\ 1 & 3 \\ 2 & * \\ 0 & 3 \end{bmatrix}$

Figure 1: A Simple Example for Sparse Matrix Formats

2.2 JavaScript

The web started as a simple network to exchange static documents. By historical accident, JavaScript resulted in the only natively supported language. Initially, JavaScript was targeted to be a companion scripting language meant for non-professional programmers [25]. The language was made to be simple, with features such as a single supported type for numbers, in the shape of double-precision floating points, a simple concurrency model with only one main thread, and a dynamically-typed nature. In the beginning, JavaScript was mostly used to do simple tasks such as scrolling.

In 2004, the AJAX technology came along introducing dynamic content to the web and thus placing JavaScript in the spotlight, increasing the demand for performance in the web. In 2008, seeking this performance, the major browser vendors introduced their JavaScript engines and sophisticated JIT compilers [8], the compilers would bring many optimizations and evolve into complex applications, each with their own

optimization strategies and criteria. Since then the web has evolved to become a ubiquitous platform for sharing information and the demand for performant applications on the web has increased. As such the engines have continued to improve their performance coming to within 1.5 to 2 factors from native C code [13].

In 2017, both Mozilla and Google introduced significant changes to their browsers. First, similar to Chrome, Firefox was changed to start a separated process for each browser tab. Moreover, Mozilla introduced their brand new Firefox Quantum Browser, which increased the performance of its engine significantly [18, 27]. Chrome has recently revamped their compiler architecture to make it easier to scale, and to add new optimizations. In this paper we use the most recent versions of both Chrome and Firefox, so that the JavaScript versions of SpMV can benefit from the most recent execution engines and optimizations.

3 Experimental Design

In this section we outline our experimental setup, including details about our target languages and architecture, the input matrices we used, our reference implementations in C and JavaScript, important aspects of how the matrices were input, and our data collection strategy.

3.1 Target Languages and Runtime

We conducted our experiments on Intel Core i7-3930K with 12 3.20GHz cores, 12MB last-level cache and 16GB memory, running Ubuntu Linux 16.04.2. We have compiled our C implementations with gcc version 7.2.0 at optimization level -O3. For JavaScript, we used the latest browsers – Chrome 63 (Official build 63.0.3239.84 with V8 JavaScript engine 6.3.292.46) and Firefox Quantum (version 57.0.4).

3.2 Input Matrices

We used 1,981 real-life square sparse matrices from The SuiteSparse Matrix Collection (formerly the University of Florida Sparse Matrix Collection) which served as the set of sparse matrix benchmarks for our experiments [6].[1] This collection provides matrices in three external storage formats : MATLAB, Rutherford Boeing and Matrix Market. We chose Matrix Market format as an input to our programs, and used [5] library for Matrix Market I/O.

There are some important details when using the input data set. First, for some of the input files, there are more row, col, val entries than the number of non-zeros in the matrices. In this case the explicit zero values from the input file must be filtered out before storing the array in a specific sparse format. Second, for some symmetric matrices, one entry for (i,j) or (j,i) is included in the file, and the other entry is implicit. This must also be dealt with when reading in the inputs. Finally, some input files have just row and col values for the non-zero entries, and do not contain any input for values, so we initialized the values to 1 for those kind of inputs.

We have implemented SpMV for both float and double, but the `val` [2], of every matrix is read as a double and later cast to float. This is because if we read it as a float, some values which underflow will appear to be 0, and we need to distinguish these values from explicitly stored zeros.

[1] This is the complete set of square matrices, with a few exceptions where a matrix would not fit into available browser memory, and thus was excluded from our study.

[2] The array containing the actual non-zero values.

3.3 Measurement Setup

We ran SpMV for each matrix and each format 100 times, measured the time for each execution, and then used the mean to calculate the final GFLOPS, which is the ratio of number of floating-point operations to execution time. In JavaScript, we removed the effect of JIT compile time by warming up the computation with 10 runs, and then taking the measurement for the next 100 runs.

For C, we used the `clock()` method from `time.h`, while for JavaScript, we used the `performance.now()` method to measure the execution time.

3.4 Reference C Implementation

We developed reference set of sequential C implementations of SpMV, one for each of the four formats that we studied. Our implementations[3] follow closely to conventional implementations of SpMV that target cache-based superscalar uniprocessor machines. We have focused on uniprocessor sequential SpMV to have a fair comparison with JavaScript which is single threaded. Figure 2 shows our SpMV reference implementations for all four formats. We have used a macro called `MYTYPE` which is assigned to either float or double to compile the implementation for single- or double-precision respectively.

SpMV COO: Figure 2a shows our implementation for SpMV COO which is as simple as this format representation. Since the `row`, `col` and `val` arrays are accessed sequentially, the access is predictable which provides the benefit of spatial locality. The `x` array is used repeatedly, but may not provide useful temporal locality because of the irregular access due to indirect addressing.

SpMV CSR: Our implementation for CSR, shown in Figure 2c, has two loops, where outer loop iterates through the rows, and the inner loop iterates through all the columns in a particular row. Due to the fact that CSR is inherently sorted by row, the values of `y` can be stored in a register after being fetched once. Also, similar to COO, the predictable access to `row_ptr`, `col` and `val` arrays presents the advantage of spatial locality. It is important to note that if there are only a few elements in each row, then the overhead of inner loop increases, and it can hurt the performance. Also, if the number of non-zeros are less than the dimension of the sparse matrix, then CSR will have more iterations for outer loop as compared to SpMV COO, where the number of iterations will be equal to the number of non-zeros. The access pattern for the `x` array remains the same as SpMV COO.

SpMV DIA: Figure 2d has two loops, where the outer loop iterates through each diagonal, and the inner loop iterates through the elements of a specific diagonal. For every diagonal, the elements of arrays `x` and `y` are accessed contiguously, and may be used repeatedly, hence it is possible to harness the advantage of both spatial and temporal locality in this case. The accesses to the `data` and `offset` arrays are also predictable, and can benefit from spatial locality.

SpMV ELL: The implementation of ELL in Figure 2b utilizes the idea of storing the data and indices 2-D arrays in a column- major order. This format is preferred for those matrices where the number of non-zero elements in all the rows is almost equal. It is assumed here that the number of rows will be more than the number of non-zero elements per row. To reduce the loop overhead, the inner loop iterates over the number of rows in each column of the `data` array, and the outer loop iterates over the number of columns in the `data` array. In order to keep the benefit of spatial locality in this kind of setup, it

[3]Implemenation can be found in Github repository – link removed for double blind review

```
void spmv_coo(int *rowind
    , int *colind, MYTYPE *val, int
     nz, int N, MYTYPE *x, MYTYPE *y)
{ int i;
  for(i = 0; i < nz ; i++)
    y[rowind
        [i]] += val[i] * x[colind[i]];
}
```

(a) SpMV COO

```
void spmv_ell
    (int *indices, MYTYPE *data, int
     N, int nc, MYTYPE *x, MYTYPE *y)
{ int i, j;
  for(j = 0; j < nc ; j++){
    for(i = 0; i < N; i++)
      y[i] += data[j * N
          + i] * x[indices[j * N + i]];
  }
}
```

(b) SpMV ELL

```
void spmv_csr
    (int *row_ptr, int *colind, MYTYPE
     *val, int N, MYTYPE *x, MYTYPE *y)
{ int i, j;
  MYTYPE temp;
  for(i = 0; i < N ; i++){
    temp = y[i];
    for(j = row_ptr
        [i]; j < row_ptr[i+1]; j++)
      temp += val[j] * x[colind[j]];
    y[i] = temp;
  }
}
```

(c) SpMV CSR

```
void spmv_dia(int *
    offset, MYTYPE *data, int N, int nd
    , int stride, MYTYPE *x, MYTYPE *y)
{ int i, k, n, istart, iend, index;
  for(i = 0; i < nd; i++){
    k = offset[i];
    index = 0;
    istart = (0 <
       -k) ? index = N-stride, -k : 0;
    iend = (N-1 < N-1-k) ? N-1 : N-1-k;
    for(n = istart; n <= iend; n++)
      y[n] += (data[(size_t
          )i*stride+n-index] * x[n+k]);
  }
}
```

(d) SpMV DIA

Figure 2: Reference C Implementation for SpMV

makes perfect sense to store the arrays in that order. Due to indirect addressing, the access to the `x` array is not predictable like in cases of CSR and COO.

In order to demonstrate that our reference C implementation is reasonable, we compared it to two popular libraries, Intel MKL [30] and Python SciPy [11], both of which provide the SpMV routines for three of our formats : COO, CSR and DIA. Also, to check the correctness of the output of our implementations, we have computed the fletcher sum of the output `y` array to verify that we compute the same results as the existing libraries.

Intel MKL provides multithreading support, but we explicitly set the number of threads to one by calling mkl_set_num_threads(1) before utilizing their routines to execute SpMV. We use the same I/O library with Intel MKL as our implementation to read the input matrices from the files. The conversion routines that are used to convert the format from COO to CSR, and CSR to DIA are those provided by the Intel MKL library.

On the other hand, Python SciPy has its own routines to read the matrices from Matrix Market

format input files, and also for the conversion between the two storage formats. [4]. We have used the `time.clock()` routine in Python to measure the execution time.

Table 1 presents the comparison between our reference C implementation and both Intel MKL and SciPy. Each n column in this table indicates the size of the working set of matrices. Each *ratio* column in this table shows the relative performance of our reference C implementation and the other library. If the ratio is less than 1, then the other implementation is faster than our reference C implementation. If the ratio is greater than 1, then the other implementation is slower than our reference implementation.

The detailed computation we performed is as follows. Consider that we are computing the slowdown of a library L versus our implementation C for a specific format F. We first determined the set of input matrices that have *10%-affinity* for format F for both the L and C implementations. Let us call these matrices A_1 through A_m. We then compute, for each A_i, the slowdown of $L(A_i)$ versus $C(A_i)$, call this slowdown s_i. To summarize the slowdown over the set of input matrices, we then compute the geometric mean of all the s_i.

It is quite evident from the table that the performance of our implementation is close to both Intel MKL and Python SciPy, in most cases. In fact, for COO matrices, the performance number is almost the same among all the libraries for both single- and double-precision. The routines in Intel MKL are hand-optimized specifically for intel processors, which explains the better performance of Intel MKL in case of CSR and DIA formats. We also observe that there is a significant slowdown in case of DIA in Python SciPy as compared to our implementation.

		COO		**CSR**		**DIA**	
		n	**ratio**	**n**	**ratio**	**n**	**ratio**
MKL	single	97	1.04	221	0.76	103	0.97
	double	49	1.09	174	1.078	22	0.92
Scipy	single	122	0.95	399	1.03	32	2.28
	double	53	0.96	790	1.09	23	1.90

Table 1: Performance comparison of reference C implementation versus Intel MKL and Python SciPy

3.5 Reference JavaScript Implementation

Our JavaScript implementations algorithmically follow the C versions, but of course must follow the JavaScript-specific notions for arrays and types. In particular, as illustrated in Listing 1, we use JavaScript typed arrays: `Int32Array` for the auxiliary arrays, `Float32Array` for single-precision and `Float64Array` for double-precision. The use of typed arrays provides some optimization opportunities for the JavaScript engines. Another key point is that in JavaScript the numbers returned by an arithmetic operation are by default double-precision, so the single-precision versions of SpMV must cast those double-precision values back to single-precision. We found that the most efficient way of doing this was by the careful use of `Math.fround`, as illustrated in the loop body of `spmv_coo` in Listing 1.

[4]We should note that the Python routings for reading matrices do not eliminate extraneous zero entries from the input files, and so may store and compute extra values, as compared to the standard approach of filtering out extraneous input zero entries.

```
\\ efficient representation, using typed arrays
var coo_row = new Int32Array(nz)
var coo_col = new Int32Array(nz)
var coo_val = new Float32Array(nz)
var x = new Float32Array(cols)
var y = new Float32Array(rows);

\\ note the use of Math.fround in the loop body
function spmv_coo(coo_row, coo_col, coo_val, N, nz, x, y)
{
  for(var i = 0; i < nz; i++)
    y[coo_row[i]] += Math.fround(coo_val[i] * x[coo_col[i]]);
}
```

Listing 1: Single-precision SpMV COO implementation in JavaScript

4 Results and Analysis

SpMV is usually evaluated in the traditional context of native implementations (often in C or C++) on server machines. However, the point of our experiments is to evaluate SpMV in the context of web-based computation and JavaScript. We hope that these results will shed light onto best practices for SpMV using modern web technologies, which will help enable web-based scientific, big data and deep learning applications using SpMV. We also hope that our observations and our methodology will be useful to develop and evaluate future optimizations and web-based technologies.

Each execution environment has its own idiosyncrasies coming from the language, compiler and target architecture. Optimizations that benefit programs in one specific setup may not benefit another setup. This phenomenon similarly affects other aspects which make the study and comparison of performance across different environments interesting and ultimately useful for application developers. In this case, we study the performance of the SpMV operation for both the Firefox and Chrome environments. In this analysis, we have focused on the effect of floating point precision in SpMV, and the choice of optimal format in each environment. **RQ1** motivates the study by reporting a relative performance comparison of C and JavaScript using the native, Firefox and Chrome environments. We highlight some of the main features of these environments and introduce our next two research questions which explore those highlighted remarks in more detail. In **RQ2**, we explore the effect of a given floating point precision on the performance and the choice of optimal format within the same environment. Lastly, in **RQ3**, we study the difference between optimal format for the C native environment and the two web browsers.

4.1 RQ1: Performance Comparison between C and JavaScript

To begin, we compare the performance of the JavaScript environments versus the native C environment for both single- and double-precision.

The plots in Figure 3 present the slowdown in performance of JavaScript relative to C for both Firefox and Chrome, single- and double-precision. A performance higher than the C baseline means the JavaScript SpMV implementation for a given web environment is slower than the C implementation for the native environment by the factor shown on the y-axis.

To obtain Figure 3 plots, we first determine the set of matrices in C that fulfill the *10%-affinity* criteria.

Using these matrices, we make two comparisons between native C and the two JavaScript environments. i.e. *best-vs-best* and *best-vs-same*. The computation for *best-vs-best* is done as follows. Let the matrices in the set be numbered m_1,...,m_k. Let $best_c(m_i)$ be the performance of the best format for matrix m_i using the native C environment. Similarly, let $best_{firefox}(m_i)$, $best_{chrome}(m_i)$, represent the performance of the best performing format for matrix m_i in Firefox and Chrome respectively. For each of the m_i, we compute the ratios $F_i = \frac{best_c(m_i)}{best_{firefox}(m_i)}$ and $C_i = \frac{best_c(m_i)}{best_{chrome}(m_i)}$, representing the slowdowns for best formats of Firefox and Chrome relative to the best native C format of matrix m_i. To summarize the results, we compute the geometric mean of sets F_1,...,F_k and C_1,...,C_k, which represent the overall slowdown across all matrices for Firefox and Chrome relative to C, resulting in our *best-vs-best* comparison. The *best-vs-same* comparison follows a similar procedure, in this case, however, the sets F_i and C_i are computed by taking the performance of the best format in C, say COO for matrix m_i and using the same COO format performance for matrix m_i in Firefox and Chrome, thus the name *best-vs-same*. The intention of *best-vs-same* is to compare the performance of JavaScript and C if we were to assume that the optimal format in C for a given matrix is also the optimal format for that matrix in the JavaScript environments.

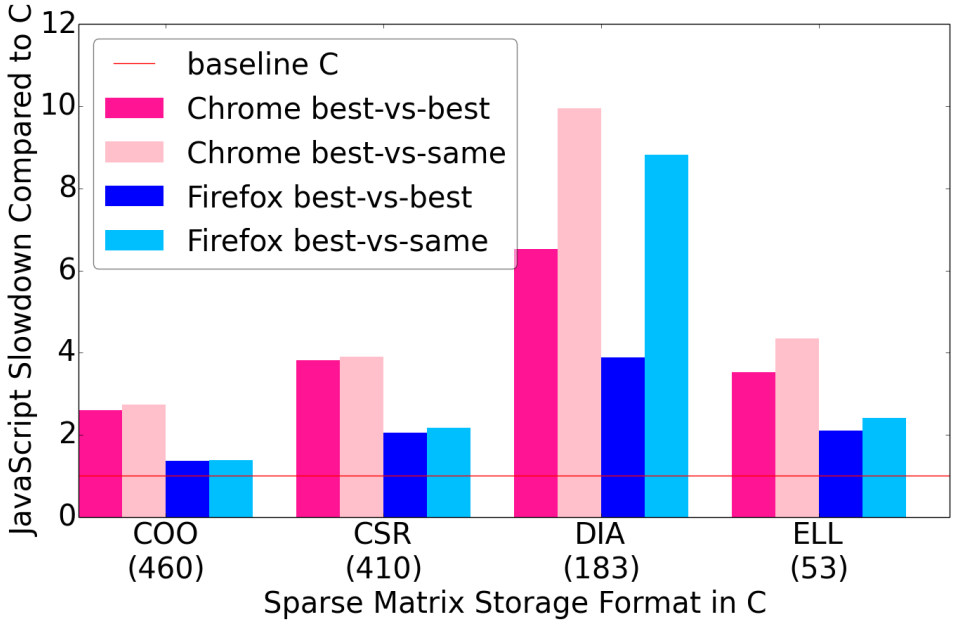
Figure 3a and Figure 3b present the comparison of the single-precision and double-precision implementations in Firefox and Chrome versus the C implementation using the native environment, here we have used our *best-vs-best* and *best-vs-same* comparisons. The x label contains the different formats. For each format F_i, the value for the bars is based only on the matrices that showed *10%-affinity* to that format in C. The numbers below the x-labels correspond to the number of matrices for each format F_i in C. Note that the numbers are different in Figure 3a and Figure 3b due to having different matrix sets for both single-precision and double-precision that met the *10%-affinity* criteria. Therefore comparison of performance between the different precisions should not be made here, we give a more rigorous comparison for RQ2 in Section 4.2.

From the plots we make the following observations. First, Firefox outperforms Chrome when comparing *best-vs-best* and *best-vs-same*, this difference is partly due to the fact that Firefox optimizes their `Math.fround` function to perform single-precision floating point arithmetic, which is less computationally expensive than the equivalent double-precision arithmetic [4, 3]. On the other hand, the Chrome V8 team seems to have left the optimizations of single-precision floating point arithmetic to WebAssembly [28, 2]. Second, the values for *best-vs-best* and *best-vs-same* differ when it comes to the DIA and ELL formats, this highlights and inspires the need to explore the difference between optimal formats in C and JavaScript, which we explore in **RQ3**. Third, the DIA format performs very poorly in JavaScript for both Firefox and Chrome, single- and double-precision. This is due to the optimization difference that benefits the DIA format in C which are missing in both the Firefox and Chrome engines. All these phenomenon will be studied in more depth when we explore the difference in formats and precisions within an environment in **RQ2**, and across environments in **RQ3**.

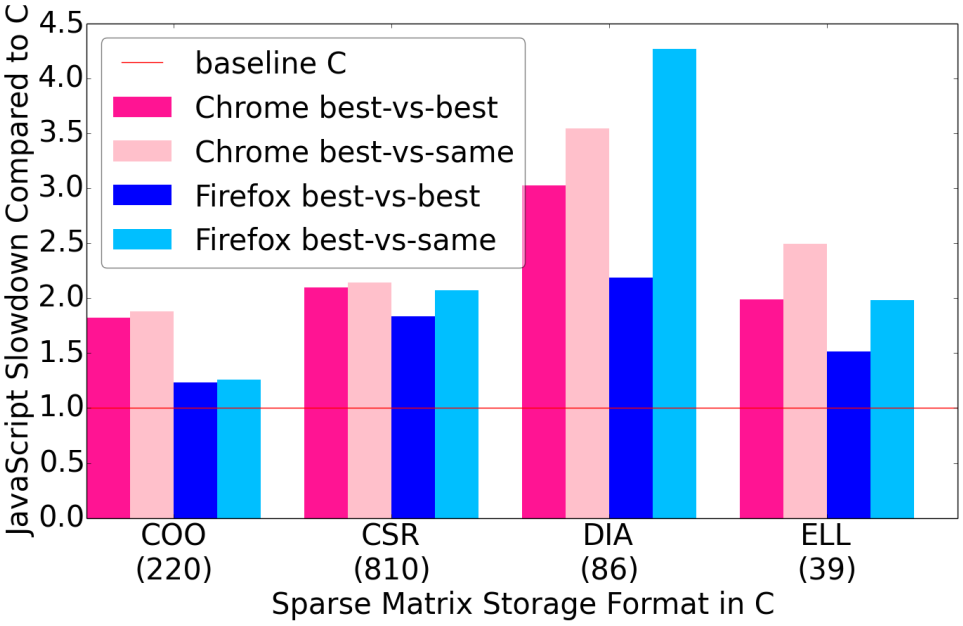
4.2 RQ2 : Performance Comparison and Format Difference between Single- and Double-precision for both C and JavaScript

In the previous section we observed that JavaScript is slower than C, but one intriguing question is the relative slowdowns for single- and double-precision. The usual expectation is that single-precision computations are more efficient than double-precision, but is this the case for SpMV in the C and JavaScript contexts? Secondly, we know that the efficiency of SpMV depends on a good format choice. So another interesting question is to examine if the format choice is the same for single- and double-precision?

Thus, in this section we analyze single- and double-precision SpMV within each environment looking at both performance and format distribution. The order of the section is as follows, first we discuss the



(a) Single-precision



(b) Double-precision

Figure 3: Slowdown of JavaScript relative to C using the 10%-affinity

difference in performance between double- and single-precision in each environment starting from the native C environment, second, we study the format distribution in C, and discuss the performance of each format, using our *x%-affinity* notion.

4.2.1 Performance of Single-precision versus Double-precision

		COO			CSR			DIA			ELL		
		n	**GFLOPS**	**ratio**	**n**	**GFLOPS**	**ratio**	**n**	**GFLOPS**	**ratio**	**n**	**GFLOPS**	**ratio**
C	single	184	1.045	1.094	366	1.88	1.08	81	3.67	2.0	14	1.46	1.24
	double		0.95			1.74			1.83			1.17	
Chrome	single	274	0.37	0.75	748	0.52	0.65	21	0.05	0.83	56	0.34	0.80
	double		0.49			0.80			0.06			0.42	
Firefox	single	932	0.77	0.96	217	0.90	0.92	-	-	-	1	0.73	0.97
	double		0.82			0.85			-			0.75	

Table 2: Performance comparison between single- and double-precision SpMV for both C and JavaScript using the 10%-affinity

Table 2 compares the different formats between single- and double-precision implementations. For this experiment, we selected the matrices for each format based on our *10%-affinity* criteria for both single- and double-precision, and then took the intersection of those sets of matrices to generate a working set of benchmarks. This ensures that we compare the performance for the same set of matrices in both the single- and double-precision contexts. We calculated the arithmetic mean of GFLOPS for the chosen matrices among the different formats for each precision. We also computed the ratio of single- versus double-precision which represents the performance difference between single- and double-precision for SpMV.

For the native C environment, we observe that single-precision is more efficient than double-precision SpMV for all formats. In single-precision, a 32-bit number takes half the space compared to a 64-bit number in double-precision, which decreases the total memory used by the application, and increases the speed of memory-bound operations like SpMV. In addition, doubling the memory requirement for each floating-point number increases the load on cache and memory bandwidth to fill and spill those cache lines. For the DIA format, in particular, single-precision is almost twice as fast as double-precision. We have identified one important source of this difference, the relative effectiveness of SIMD (Single Instruction, Multiple Data) optimizations. Consider Figure 2d, which displays the implementation of SpMV DIA. In the inner loop the elements of the arrays are accessed contiguously as `n` goes from `istart` to `iend`. This situation provides a clear opportunity for the compiler to perform the SIMD optimization. In GCC, the optimization flag `--ftree-loop-vectorize` included within -O3 optimization level, triggers the loop vectorization for *SpMV DIA* in C, which happens for both single- and double-precision. However, single-precision gains more from this optimization since a a register can pack double the number of single-precision floating-point numbers, thus halving the number of SIMD instructions required.

For the web environments, we conducted a similar evaluation between single- and double-precision in both Chrome and Firefox browsers. Contrary to the behavior in C, we observed that double-precision SpMV has better performance than single-precision SpMV in JavaScript for Chrome, and Firefox. In our single-precision implementation for JavaScript, we have used `Float32Array` typed array to allocate space for `x`, `y` and `val` arrays. JavaScript natively only supports double-precision which means every arithmetic operation on numbers is a double precision operation. Therefore, for the multiplication operation between `val` and `x`, we use float64 arithmetic, but the operands are float32. The float32 operands are first cast to float64, and then multiplied together using float64 arithmetic. To store back in the `Float32Array` array, we include a call to `Math.fround()` which returns the nearest 32-bit single-precision floating point representation. This results in overheads for both Firefox and Chrome. In case of Firefox, however, the casting penalty is lower since the `Math.fround()` function has been optimized by the Firefox engine to perform float32 arithmetic instead of casting the values back and forth between single and double [4].

4.2.2 Storage Formats

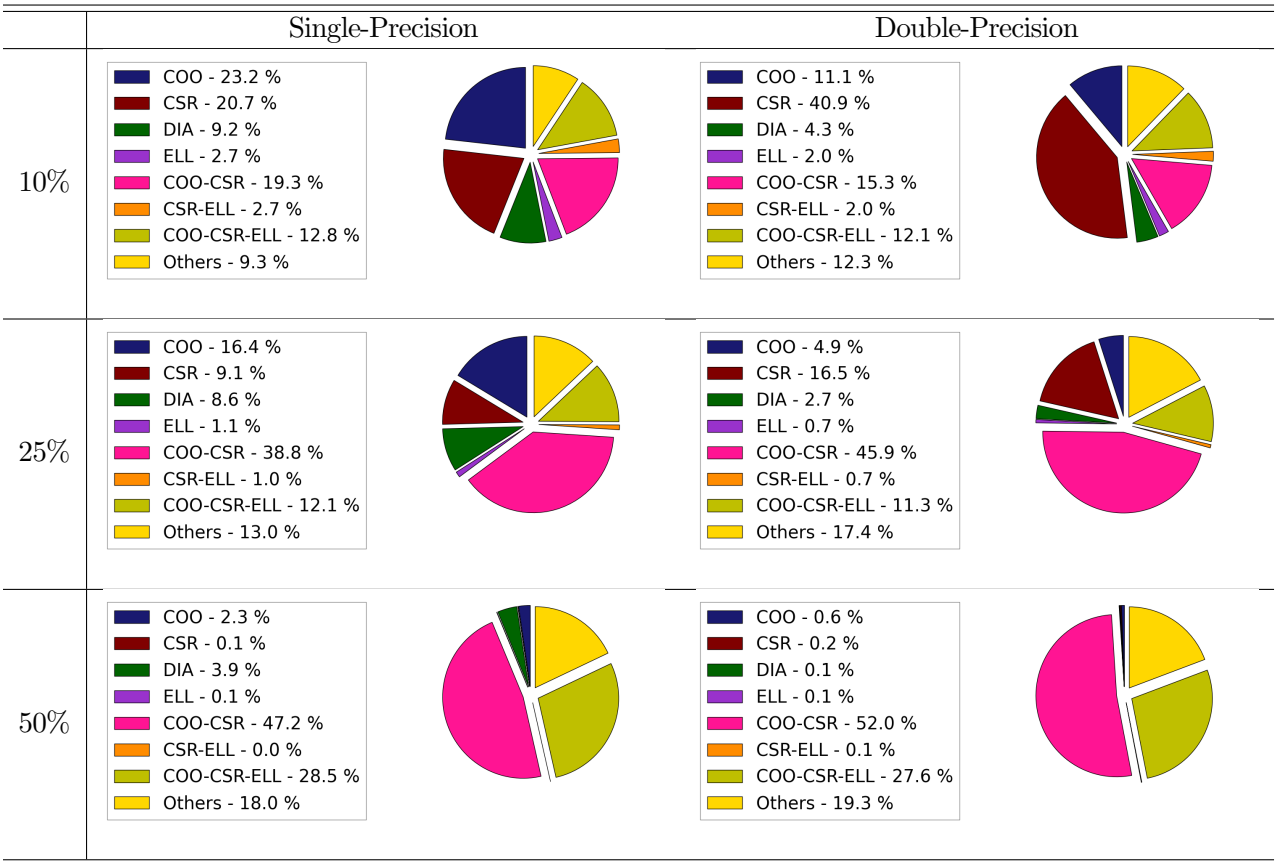


Figure 4: Format distribution for C using different x%-affinity for both single- and double-precision

Along with the performance difference, we also observed the difference in choice of storage format between single-and double-precision implementations. Figure 4 shows the format distribution in C for both single- and double-precision again based on 3 levels of affinity, at 10%, 25% and 50%.

First consider the upper left part of Figure 4, which shows the distribution for *10%-affinity*, single-precision. We see that there are four *single-format* categories (COO, CSR, DIA and ELL), with 23.2% of the matrices showing *10%-affinity* COO, 20.7% for CSR, 9.2% for DIA and 2.7% for ELL. However, often there is no clear 10% winner because several different formats may show significantly better performance than the others. In these cases we create *combination-formats*, such as COO-CSR, which accounts for 19.3% of the input matrices in this example. An input matrix is counted in the *10%-affinity* COO-CSR category if COO and CSR are the two best performing formats and both of them are at least 10% better than all the other formats. We identified three important *combination-formats*, COO-CSR, CSR-ELL, and COO-CSR-ELL. The "Others" category represents all other combinations.

Considering all thresholds in Figure 4, we can see some interesting trends. Firstly, as the threshold increases, the *single-format* slices shrink, meaning that there is less likely to be a clear winner when a larger performance improvement is expected. Indeed, for the 25% and 50% thresholds, the COO-CSR slice increases dramatically. In these cases if the input matrix is already in COO or CSR format, then the input format should probably just be used. In contrast, if an input matrix has high-affinity to a *single-format*, say CSR, and the input matrix is not already in CSR format, then it may be beneficial to convert to CSR.

Comparing the single-precision column to the double-precision column, we observe that COO is more prevalent for single-precision, and CSR is more prevalent for double-precision. We also note that DIA appears to be more important for single-precision than for double-precision, likely due to the greater benefit of SIMD optimizations for DIA single-precision.

A final interesting observation is that if one looks at the total of the slices for CSR plus all *combination-formats* containing CSR, we can see that CSR is the most useful format overall. Thus, if one wanted to provide just one format, then CSR would be a good format to choose. This corresponds to common practice.

4.3 RQ3 : Format Difference betweeen C and JavaScript

The fact that *best-vs-best* was better than *best-vs-same* as presented in **RQ1** provides us with a hint that the choice of best performing storage format for SpMV for a particular matrix is not always the same for C and JavaScript environments. To investigate this behavior we studied the similarities and differences in storage format between C and JavaScript for both browsers – Chrome and Firefox for both single- and double-precision using our *10%-affinity* criteria, as summarized in Figure 5. The whole stacked bar for an x-label format category represents the total number of matrices having *10%-affinity* for this format category in C, while each segment of this stacked bar represents the number of matrices having *10%-affinity* for a specific format category in JavaScript. For example, in Figure 5a, the first stacked bar shows that there are 460 matrices which have *10%-affinity* for COO format in C, and the three segments of this bar represent that out of these 460 matrices, 217, 75 and 162 matrices have *10%-affinity* for COO, CSR and COO-CSR format categories respectively in JavaScript.

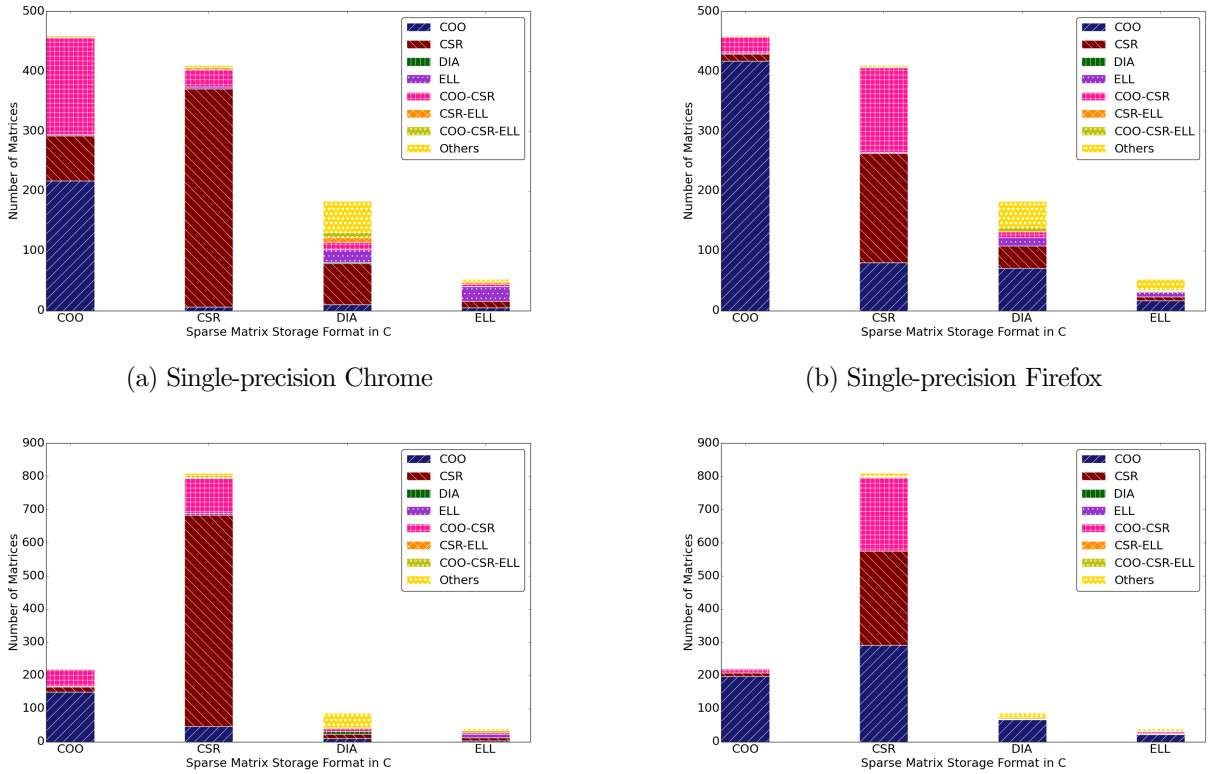
There are some similarities in affinity in C and in JavaScript, most notably CSR for Chrome (Figures 5a and 5c), and COO for Firefox (Figures 5b and 5d). However, probably the most interesting point is that overall there are a lot of differences in the affinities between C and JavaScript, and even where we see similarities, they are different for the two browsers.

One most striking example of differences is for DIA. For the matrices whose optimal format is DIA in C, it is clearly evident from the plot that the choice of format in JavaScript is radically different from C. As discussed in **RQ2**, SIMD optimization in C plays an important role in SpMV DIA performance, leading DIA to become the optimal format for these matrices. However, JavaScript currently doesn't have support for SIMD optimization. Jibaja et al. [10] proposed the design and implementation of SIMD language extensions and compiler support to add vector parallelism in JavaScript, but it is not being pursued by web browsers for implementation anymore, however, SIMD operations are under active development within WebAssembly [1].

5 Related Work

To the best of our knowledge, we are the first to examine SpMV in the context of JavaScript. In this section we examine previous work in the SpMV and JavaScript fields, and relate it to the work in this paper.

Several new SpMV storage formats have been proposed. For example, Eun-Jin Im et al. [9] proposed BCSR (Blocked CSR) format to take the advantage of performance of dense blocks in a sparse matrix. Kourtis et al. [14] developed CSX (Compressed Sparse eXtended) to compress metadata by exploiting dense structures like dense blocks, 1-D bars and dense diagonals. Liu et al. [17] proposed CSR5 format that has been claimed to be insensitive to the sparsity structure of the input matrix and needs some extra space to store two more groups of data than classic CSR. In this paper we have examined four



(a) Single-precision Chrome

(b) Single-precision Firefox

(c) Double-precision Chrome

(d) Double-precision Firefox

Figure 5: Affinity of matrices towards different format(s) for JavaScript relative to C using the 10%-affinity

common formats, and found that in JavaScript, the classic CSR and COO formats are the best overall, and that DIA performs very poorly in JavaScript.

Many SpMV optimizations have also been proposed, targeting either application- or architecture-specific domains. For example, Tang et al. [26] implemented and tuned SpMV for Intel Xeon Phi coprocessor and optimized its performance by employing 2D jagged partitioning, tiling and vectorized prefix sum computations. Vuduc et al. [29] improved BCSR to VBR (Variable Block Row) to take advantage of dense blocks with different sizes and built OSKI library to tune block size for a matrix in VBR format. Further, when the input matrix for SpMV operation is only available during the execution time, choosing the right format to store the matrix becomes a non-trivial task as it depends on a number of factors. In this paper we have shown that different factors are important for tuning SpMV for the web, and that both the choice of single-precision versus double-precision, and the optimal choice of format, is different in JavaScript than in C. Indeed, the performance even varies significantly between different web browsers.

Interesting work has also been proposed to predict the best sparse matrix storage format for SpMV computation [20, 16]. SMAT [16] focused on providing SpMV kernels based on a machine learning approach by automatically determining the best storage format and implementation on a given architecture for diverse matrices. For those matrices where more than one format achieve similar SpMV performance, it is hard to claim one of them as the best format. We think that our concept of *x%-affinity* to classify the matrices into *single-format* and *combination-format* categories could be useful for both evaluating new formats and classifying formats for machine-learning.

Research in analyzing and improving the behavior and performance of JavaScript programs has also been on the upswing for more than a decade now. Richards et al. [22] conducted an empirical study to understand the dynamic behavior of JavaScript programs. Our work focuses on understanding the behavior of a specific JavaScript program that performs computations on sparse matrices. Khan et al. [13] developed Ostrich, an extended benchmark suite to provide empirical data that evaluated the effectiveness of JavaScript for numerical computations including SpMV for CSR format. The results in our paper help confirm the overall performance improvements shown in Ostrich, and we provide a more detailed examination of different sparse formats. Selakovic et al. [24] presented an empirical study of fixed performance issues from JavaScript projects and identified their causes. In addition to general performance analysis, we found a specific performance issue with storing the sparse matrix in DIA format for SpMV computation, and identified SIMD as the potential reason.

There have been significant efforts to provide machine learning frameworks in JavaScript, in which SpMV forms an important core computation. For example, Meeds et al. [19] introduced a prototype machine learning framework called MLitB (Machine Learning in the Browser), written entirely in JavaScript which is capable of performing large-scale distributed computing. Thorat et al. [21] developed a JavaScript library called deeplearn.js at Google Brain that allows training of neural networks in a JavaScript environment. We hope that our work in analyzing the performance of SpMV can help to optimize these and similar JavaScript projects in machine learning domains.

6 Conclusion and Future Work

In this paper we have provided a detailed analysis of SpMV for web-based JavaScript, examining both the performance as compared to C, the relative performance of single- versus double-precision in both C and JavaScript, and a detailed examination of the best choice of sparse matrix format.

In doing this work we found that it was very important to identify which format is the best in a rigorous manner. To that end, we defined the notion of *x%-affinity*, to identify when a particular format is a winner by at least *x%*. We also found that it was useful to look at both: (1) *single-format* cases, where a matrix has affinity to one specific format, and (2) common *combination-format* cases, where a matrix has affinity to a group of formats, all of which perform well. We hope that these ideas will be useful for others in future studies of new and existing formats.

Our experiments, using almost 2000 real-life sparse matrices, showed some very interesting results. First, in terms of raw performance, the best performing browser showed a geometric mean slowdown of only 1.2 to 3.9 over C. This shows that SpMV on the web is becoming very practical, and thus it has become realistic to utilize web-connected devices for compute-intensive applications using SpMV.

Second, unlike in C, where single-precision SpMV is faster than double-precision, the opposite is true for web-based JavaScript. This was particularly the case for Chrome, where single-precision SpMV was only 0.65 (CSR) to 0.83 (DIA) times the speed of double-precision SpMV.

Finally, we saw several interesting behaviours regarding the best choice of format. We first examined the best choice of format for C, for both single- and double-precision, and for 10%-, 25%- and 50%-affinities. These results showed that there are some significant differences between the best format choice for single- and double-precision. Most notably, single-precision shows greater affinity for COO and and double-precision shows greater affinity for CSR. As we increased the affinity thresholds, we noted that several *combination-formats* become more signficant. In particular, COO-CSR and COO-CSR-ELL become the dominant categories. Indeed, the *50%-affinity* experiment shows that CSR covers the vast

majority of *single-format* and *combination-format* groups. This demonstrates that the conventional wisdom that CSR is a good overall choice for SpMV is very valid.

We also examined the differences in best choice of format for C versus JavaScript. These results show that the best format choice is very different in the two contexts, and even show different behaviours for the two browsers. The only case which showed similar affinities between C and JavaScript were for CSR for Chrome. All other cases showed significant differences. In particular, the DIA format seems to work very poorly in JavaScript compared to C. This appears to be due to the lack of SIMD optimizations in JavaScript. This means that techniques for choosing the best formats must be tailored for the web-based SpMV, one cannot use the same predictions as for C. For current implementations of JavaScript, CSR seems to be a reasonable choice, and the benefit of DIA and ELL seems to be very minimal.

In our future work we wish to examine how the web-based sparse matrix computations can be further improved. Upcoming browser-based technologies, like WebAssembly, and new support for SIMD and multi-threading, will provide new opportunities for more optimized and parallelized versions of SpMV. We intend to use the methodologies developed in this paper to examine these new technologies as they become available. We also plan to explore SpMV optimization opportunities that will be particularly beneficial for the SpMV implementation in Javascript because of its dynamic nature. These SpMV optimizations should typically improve the temporal data locality in access to the source vector x and output vector y. Some traditional examples of such optimizations are cache blocking, register blocking and reordering of rows or columns. It has been observed in some applications like Hypre AMG solver that the structure of sparse matrix changes a number of times during the execution, and it requires frequent conversions between different formats for the optimal performance. So, we also want to improve upon the overhead cost of conversion among the storage formats.

References

[1] Features to add after the MVP.

[2] Alon Zakai. Chrome Perf Issues. `https://github.com/kripken/emscripten/wiki/Chrome-Perf-Issues`. [Accessed: 2018-01-19].

[3] R. N. Alon Zakai. Gap between asm.js and native performance gets even narrower with float32 optimizations.

[4] Benjamin Bouvier . Efficient float32 arithmetic in JavaScript. `https://blog.mozilla.org/javascript/2013/11/07/efficient-float32-arithmetic-in-javascript/`. Accessed: 2018-01-19.

[5] R. F. Boisvert, R. Pozo, K. Remington, R. F. Barrett, and J. J. Dongarra. Matrix market: a web resource for test matrix collections. In *Quality of Numerical Software*, pages 125–137. Springer, 1997.

[6] T. A. Davis and Y. Hu. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1, 2011.

[7] J. Duda and W. Dlubacz. Distributed Evolutionary Computing System Based on Web Browsers with Javascript. In *PARA'12*, pages 183–191, 2013.

[8] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and

M. Franz. Trace-based just-in-time type specialization for dynamic languages. In *PLDI'09*, pages 465–478. ACM, 2009.

[9] E.-J. Im, K. Yelick, and R. Vuduc. Sparsity: Optimization framework for sparse matrix kernels. *The International Journal of High Performance Computing Applications*, 18(1):135–158, 2004.

[10] I. Jibaja, P. Jensen, N. Hu, M. R. Haghighat, J. McCutchan, D. Gohman, S. M. Blackburn, and K. S. McKinley. Vector parallelism in javascript: Language and compiler support for simd. In *PACT'15*, pages 407–418. IEEE Computer Society, 2015.

[11] E. Jones, T. Oliphant, P. Peterson, et al. SciPy: Open source scientific tools for Python, 2001–. [Online; accessed ¡today¿].

[12] A. Karpathy. Convnetjs: Deep learning in your browser (2014). *URL http://cs. stanford. edu/people/karpathy/convnetjs*, 2014.

[13] F. Khan, V. Foley-Bourgon, S. Kathrotia, E. Lavoie, and L. J. Hendren. Using JavaScript and WebCL for numerical computations: a comparative study of native and web technologies. In *DLS'14*, pages 91–102, 2014.

[14] K. Kourtis, V. Karakasis, G. Goumas, and N. Koziris. Csx: An extended compression format for spmv on shared memory systems. In *PPoPP'2011*, pages 247–256. ACM, 2011.

[15] P. Langhans, C. Wieser, and F. Bry. Crowdsourcing MapReduce: JSMapReduce. In *WWW '13 Companion*, pages 253–256, 2013.

[16] J. Li, G. Tan, M. Chen, and N. Sun. Smat: An input adaptive auto-tuner for sparse matrix-vector multiplication. In *PLDI'13*, pages 117–126. ACM, 2013.

[17] W. Liu and B. Vinter. Csr5: An efficient storage format for cross-platform sparse matrix-vector multiplication. In *ICS'15*, pages 339–350. ACM, 2015.

[18] M. Mayo. Introducing the New Firefox: Firefox Quantum.

[19] E. Meeds, R. Hendriks, S. al Faraby, M. Bruntink, and M. Welling. Mlitb: machine learning in the browser. *PeerJ Computer Science*, 1:e11, 2015.

[20] B. Neelima, G. R. M. Reddy, and P. S. Raghavendra. Predicting an optimal sparse matrix format for spmv computation on gpu. In *IPDPSW'14*, pages 1427–1436. IEEE Computer Society, 2014.

[21] Nikhil Thorat, Daniel Smilkov, and Charles Nicholson. deeplearn.js – a hardware-accelerated machine intelligence library for the web. [Online; accessed ¡today¿].

[22] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of javascript programs. In *PLDI'10*, pages 1–12. ACM, 2010.

[23] Y. Saad. Sparskit: a basic tool kit for sparse matrix computations, 1994.

[24] M. Selakovic and M. Pradel. Performance issues and optimizations in javascript: An empirical study. In *ICSE'16*, pages 61–72. ACM, 2016.

[25] C. Severance. JavaScript: Designing a Language in 10 Days. *Computer*, 45(2):7–8, Feb. 2012.

[26] W. T. Tang, R. Zhao, M. Lu, Y. Liang, H. P. Huynh, X. Li, and R. S. M. Goh. Optimizing and auto-tuning scale-free sparse matrix-vector multiplication on intel xeon phi. In *CGO'15*, pages 136–145, 2015.

[27] M. F. Team. Firefox Quantum is super fast, while still conserving memory.

[28] V8 group. Add support for float32 to TurboFan. `https://bugs.chromium.org/p/v8/issues/detail?id=3589`. Accessed: 2018-01-19.

[29] R. Vuduc, J. W. Demmel, and K. A. Yelick. Oski: A library of automatically tuned sparse matrix kernels. In *Journal of Physics: Conference Series*, volume 16, page 521. IOP Publishing, 2005.

[30] E. Wang, Q. Zhang, B. Shen, G. Zhang, X. Lu, Q. Wu, and Y. Wang. Intel math kernel library. In *High-Performance Computing on the Intel® Xeon Phi*, pages 167–188. Springer, 2014.