A fully structure-driven performance analysis of sparse matrix-vector multiplication

Prabhjot Sandhu
McGill University
School of Computer Science
Montreal, QC, Canada
prabhjot.sandhu@mail.mcgill.ca

Clark Verbrugge
McGill University
School of Computer Science
Montreal, QC, Canada
clump@cs.mcgill.ca

Laurie Hendren
McGill University
School of Computer Science
Montreal, QC, Canada
hendren@cs.mcgill.ca

ABSTRACT

Sparse matrix-vector multiplication (SpMV) is an important kernel in many scientific, machine-learning, and other compute-intensive applications. Performance characteristics, however, depend on a complex combination of storage format, machine capabilities, and choices in code-generation. A deep understanding of the relative impact of these properties is important in itself, and also to better understanding the performance potential of alternative execution contexts such as web-based scientific computing, where the recent introduction of *WebAssembly* offers the potential for low-level, near-native performance within a web browser.

In this work we characterize the performance of SpMV operations for different sparse storage formats based on the sparse matrix structure and the machine architecture. We extract structural properties from 2000 real-life sparse matrices to understand their impact on the choice of storage format and also on the performance within those storage formats for both WebAssembly and native C. We extend this with new matrix features based on a "reuse-distance" concept to identify performance bottlenecks, and evaluate the effect of interaction between the matrix structure and hardware characteristics on SpMV performance. Our study provides valuable insights to scientific programmers and library developers to apply best practices and guide future optimization for SpMV in general, and in particular for web-based contexts with abstracted hardware and storage models.

CCS CONCEPTS

• **Mathematics of computing** → *Computations on matrices*; • **Software and its engineering** → *Dynamic compilers.*

KEYWORDS

Sparse Matrix-Vector Multiplication, SpMV, Matrix Structure, Reuse-Distance, WebAssembly, C, Performance, Scientific Computing

ACM Reference Format:
Prabhjot Sandhu, Clark Verbrugge, and Laurie Hendren. 2020. A fully structure-driven performance analysis of sparse matrix-vector multiplication. In *Proceedings of International Conference on Performance Engineering (ICPE'20).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Recently, the performance capability of the web has been elevated with the addition of WebAssembly [15] to the world of JavaScript. It provides opportunities for the efficient execution of sophisticated and compute-intensive applications on the web, such as image editing [7], computer-aided design [6], augmented reality [1, 2], text classification [3, 5, 21] and deep learning [25]. A number of these applications [32, 35] involve sparse matrix computations which are considered important for their performance. Sparse matrix-vector multiplication (SpMV) is one such critical operation which computes $y = Ax$, where matrix A is sparse and vector x is dense. It dominates the overall application run-time through its recurring nature which makes it a good candidate for optimization. Since the matrices in these applications are large and sparse with different sparsity characteristics, a single sparse storage format is not appropriate for all of them. The choice of storage format, however, can have a significant impact on the SpMV performance based on the sparsity structure of the matrix and the available code optimization opportunities. Therefore, selecting an optimal format to store the input matrix at run-time is one way to optimize SpMV. Another way is to apply different data structure optimizations and low-level code optimizations to a single format. It sometimes results in the derivation of new formats tailored for even more specific types of matrices [18, 22, 24, 27]. However, both of these techniques require a thorough understanding of how the performance of SpMV operation is affected by the structure of the matrix and the machine characteristics.

Previous work on the performance and storage format choice for SpMV has either focused on native languages like C or only been able to highlight general differences between native C and web languages, JavaScript and WebAssembly. In this paper we focus on understanding the performance of the SpMV operation solely based on the structural properties of the matrices for both C and WebAssembly. We performed rigorous experiments on a set of almost 2000 real-life sparse matrices, storing them in the four most popular storage formats (COO, CSR, DIA and ELL).

Our investigation is structured in terms of three main research questions. Our first research question, **RQ1**, evaluates the impact of the structure of the matrix on the choice of the storage format for the SpMV operation for both WebAssembly and C. We employ an *x%-affinity* criteria [28] to obtain the set of matrices best suited for each format category. We extract some useful sparse structure features which contribute to build the affinity of our benchmark matrices towards the given formats. The results from **RQ1** showed that these structure features indeed govern the choice of storage format. However, one storage format can have a priority over the other format due to the availability of some interesting code optimizations in modern architecture.

Unanswered questions about the variation in SpMV performance of the matrices with similar feature values led us to our second research question, **RQ2**, which explores more structure features to explain the effect of the sparse matrix structure on the SpMV performance within each storage format for both WebAssembly and C. We modeled data locality for SpMV ELL and CSR execution using the *reuse-distance* concept [12] and introduced some new features like *ELL Locality Index* and *CSR Locality Index* to explain the performance bottlenecks for the given matrices based on their structure.

Features from **RQ1** and **RQ2** have their foundation in the hardware architecture. This drives our final research question, **RQ3**, aimed at measuring hardware performance properties to understand the interaction between the sparse matrix structure and hardware features. The results of this question validate our evaluations and parameter choices from **RQ1** and **RQ2**.

2 BACKGROUND AND RELATED WORK

Our work requires background knowledge of the core sparse matrix storage formats, as well as the basics of WebAssembly. After a brief presentation of these concepts, we describe related work on SpMV performance analysis.

2.1 Sparse Matrix Formats

Sparse matrices are characterized by an abundance of 0-entries relative to a small number of non-zero values. Storage formats aim to minimize space requirements, with different formats aimed at different kinds of matrix structure. Figure 1 uses a small example to illustrate encodings in the four core formats we address in this study, with details discussed below.

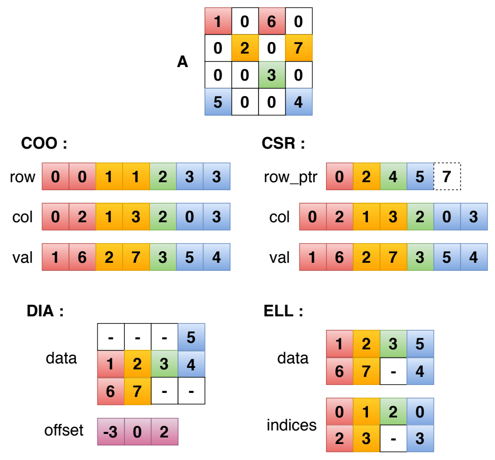


Figure 1: A simple example for sparse matrix formats

Coordinate Format (COO) is the simplest storage format. It consists of three arrays, `row`, `col` and `val` to store the row and column indices and corresponding values of each non-zero entry.

Compressed Sparse Row Format (CSR) is the most widely used format. It is the compressed version of COO format, using a `row_ptr` array with only one entry per row. Each entry in this array gives the offset in the `col` array of the first non-zero column index of that row.

Diagonal Format (DIA) only stores the diagonals that include non-zeros. The `data` array stores the diagonal values, and the `offset` stores the relative offset of each diagonal from the main diagonal.

ELLPACK Format (ELL) stores a fixed (maximum) number of non-zeros per row. The `data` array stores the values for each row, with corresponding column indices given by the `indices` array.

2.2 WebAssembly

WebAssembly [15] or *wasm* is a new virtual instruction set architecture. This binary code format for the web has been designed to complement and run alongside JavaScript, overcoming the performance limitations of JavaScript by offering a low-level, assembly-like execution context that is simpler to optimize and more directly maps to the instructions of common hardware architectures. The latter also simplifies deployment of programs written in other, statically typed languages such as C or C++, which can easily target wasm as another compilation architecture. Current versions provide some low-level machine features, such as 128-bit SIMD support, with ongoing development aimed at extending this to longer SIMD types and adding task parallelism through a thread model [4]. V8, the open-source JavaScript engine for Chrome browser, is reusing its top-tier optimizing JITs from JavaScript for WebAssembly [15].

In terms of performance, Haas et. al [15] reported a performance within 2x of native C for all benchmarks in the PolyBenchC suite, with 7 of them within 10% of native C. Further experiments by Herrera et. al [17] using the Ostrich Numerical Benchmark set showed similar performance numbers across many devices, with one device achieving better overall performance than native C. Recently, Jangda et al. [19] analyzed the performance across the SPEC CPU benchmark suite and reported a mean slowdown factor less than 2x along with the causes of these performance gaps.

2.3 Related Work

The feature-based aspect of our work of SpMV performance analysis on modern systems and different execution environments is inspired from the research which dates back to nearly three decades ago. In the early 1990s, Agarwal et al. [8] presented a feature extraction scheme for SpMV with nearly dense submatrices, nearly dense diagonals and rows with a large number of non-zeros as its features to examine sparsity structure in the pre-processing phase which could be exploited during the computation phase. During the same time, Temam and Jalby [29] analyzed the behaviour of cache for SpMV by modelling the irregular references using probabilistic methods. After a while, Heras et al. [16] modeled the data locality in the execution of SpMV by taking into account the entry matches and block matches between the pairs of rows of sparse matrices and experimented with a set of 10 sparse matrices. This groundwork provided considerable evidence that the SpMV performance depends on the structure of the sparse matrix and the memory system. Thenceforth, many research contributions have been made towards accelerating the SpMV performance which includes applying different optimization strategies [26, 31, 36, 37] and proposing various new sparse storage formats [18, 22, 24, 33]. Toledo [31] proposed reordering, blocking and prefetching to improve SpMV performance on superscalar RISC processors. Later, Pinar and Heath [26] proposed one-dimensional blocking and also stated the reordering problem as the traveling salesman problem, and used associated heuristics to permute the non-zeros of the matrix into contiguous locations. Eun-Jin Im et al. [18] proposed BCSR (Blocked CSR) format to take the advantage of the performance of dense blocks in a sparse matrix, and evaluated the strategy over a set of 40 matrices. Following this, Vuduc et al. [33] 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. However, all of these techniques were tested on quite small sets of matrices and used previous generation microarchitectures. In light of that, Goumas et al. [14] conducted experiments on a suite of 100 matrices to understand the SpMV CSR performance issues by looking into the interaction between architecture characteristics and previously reported factors like indirect memory references, irregular memory access for vector x and short row lengths.

Over the last decade as well, several other attempts towards SpMV performance optimizations have been testing their techniques only on limited sets of matrices which may or may not cover a wide variety of properties related to SpMV performance. Williams et al. [37] presented several optimization strategies like cache blocking, TLB blocking, register blocking, loop optimizations, software prefetching and more to show significant improvements on their set of 14 sparse matrices which were collected from a variety of actual applications. Kourtis et al. [22] developed CSX (Compressed Sparse eXtended) to compress metadata by exploiting dense structures like dense blocks, 1-D bars and dense diagonals, and tested the technique on a set of 15 matrices from The SuiteSparse Matrix Collection. Liu et al. [24] proposed a new storage format, CSR5, that has been claimed to be insensitive to the sparsity structure of the matrix, and needs some extra space to store two more groups of data than classic CSR, and tested it on the benchmark suite of 24 regular and irregular matrices. To that extent, one of our objectives in this paper is to fully understand the effect of matrix structure properties on SpMV performance on modern architecture by conducting comprehensive experiments on around 2000 real-life sparse matrices from The SuiteSparse Matrix Collection for four different storage formats.

Recently, some interesting work has been proposed to predict the best storage format and performance bottlenecks for SpMV computation using machine learning approaches. SMAT [23] focused on automatically determining the best storage format among COO, CSR, DIA and ELL by applying machine learning techniques which use matrix structure and hardware features as their parameter values with more than 2000 matrices from The SuiteSparse Matrix Collection for their testing and training set. Elafrou et al. [13] presented an approach to identify some selected performance bottlenecks which include memory-bandwidth bound, memory-latency bound, thread imbalance and computation bottleneck for CSR format by leveraging supervised learning technique to build a feature-guided classifier with a training set of 210 matrices from The SuiteSparse Matrix Collection. To enhance the feature selection process and achieve better accuracy for such machine learning models, we provide a concise set of structure features which indeed determine how the matrix structure affects the performance and the choice of the storage format. We also introduce some new features and evaluate the interaction between the matrix structure and hardware features since they jointly influence the SpMV performance.

In terms of web-based SpMV execution, Sandhu et al. [28] laid the foundation by presenting serial SpMV performance numbers for JavaScript and WebAssembly as compared to C for both single- and double-precision floating point representations. They introduced the notion of x%-affinity to identify with certainty that the best storage format is at least x% better performing than all other formats. They also highlighted the differences in the choice of storage format between native C and web environments. Our paper extends their study by understanding the influence of matrix structure and machine characteristics on the performance and choice of optimal storage format for Serial SpMV for web-based language, WebAssembly in comparison to native language, C.

3 EXPERIMENTAL SETUP

In this section we describe our experimental layout which covers information about our target languages, execution environments and the set of matrices used and also provides some flavour of our reference WebAssembly SpMV implementations.

3.1 Target Languages and Runtime

We conducted our experiments on an Intel Core i7-3930K with 6 3.20GHz cores, 12MB last-level cache and 16GB memory, running Ubuntu Linux 16.04.2. We compiled our C implementations with gcc version 7.2.0 at optimization level -O3. For WebAssembly, used the Chrome 74 browser (Official build 74.0.3729.108 with V8 JavaScript engine 7.4.288.25) as the execution environment. We run Chrome with a flag `--experimental-wasm-simd` to enable the use of SIMD (Single Instruction Multiple Data) instructions for loop vectorization optimizations in some of the SpMV WebAssembly implementations. We also enable two more flags, `--wasm-no-bounds-checks` and `--wasm-no-stack-checks` to avoid memory bounds checks and stack guards for performance testing. For C, we used the `clock()` method from `time.h`, while for WebAssembly, we made use of the `Date.now()`, a JavaScript method to measure the execution time.

3.2 Input Matrices

We used 1,979 real-life square sparse matrices from The SuiteSparse Matrix Collection which served as the set of sparse matrix benchmarks for our experiments [11].[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 a library [10] for Matrix Market I/O. In order to avoid SpMV performance degradation from the presence of subnormal floating point values, we substitute them with the normal values for both C and WebAssembly.

3.3 Reference WebAssembly Implementation

We developed a hand-tuned reference set of single-precision sequential WebAssembly implementations of Sparse Matrix-Vector Multiplication (SpMV) for each of the four formats. Our implementations[2] closely follow the conventional implementations of SpMV that target cache-based superscalar uniprocessor machines. Our WebAssembly

[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 implementations can be found in the Github repository – https://github.com/Sable/Sparse.SciWasm/tree/master/src/sequential

implementations algorithmically follow the reference C implementation versions from previous work [28] as illustrated in `spmv_coo` in Listing 1 and 2. In order to demonstrate that the reference C implementation is reasonable, it was compared to two popular libraries, Intel MKL [34] and Python SciPy [20] in [28] and shown that the performance of reference C implementation is close to both of them.

For SpMV DIA implementations, we apply SIMD optimizations using the new WebAssembly vector instructions as illustrated in the portion of `spmv_dia` in Listing 3.

Listing 1: Single-precision SpMV COO implementation in C

```
void spmv_coo(int *coo_row, int *coo_col,
      float *coo_val, int nnz, int N, float *x, float *y)
{ int i;
  for(i = 0; i < nnz ; i++)
    y[coo_row[i]] += coo_val[i] * x[coo_col[i]];
}
```

Listing 2: Single-precision SpMV COO implementation in WebAssembly

```
(func $spmv_coo (export "spmv_coo") (param
      $coo_row i32) (param $coo_col i32) (param $coo_val
      i32) (param $x i32) (param $y i32) (param $nnz i32)
  (local $this_y i32)
  (i32.add (local.get $coo_val) (i32.shl (i32.sub
       (local.get $nnz) (i32.const 1)) (i32.const 2)))
  local.tee $nnz
  local.get $coo_val
  i32.lt_s
  if
    (return)
  end
  (loop $top
    ((local.tee $this_y) (i32.add (local.get $y) (i32.shl
        (i32.load (local.get $coo_row)) (i32.const 2))))
    (f32.mul (f32.load (local.get $coo_val
        )) (f32.load (i32.add (local.get $x) (i32.shl (
        i32.load (local.get $coo_col)) (i32.const 2)))))
    (f32.load (local.get $this_y))
    f32.add
    f32.store
    (local.set $coo_row
         (i32.add (local.get $coo_row) (i32.const 4)))
    (local.set $coo_col
         (i32.add (local.get $coo_col) (i32.const 4)))
    (local.set $coo_val
         (i32.add (local.get $coo_val) (i32.const 4)))
    (local.get $nnz)
    (br_if $top (i32.le_s))
  )
)
```

4 RESULTS AND ANALYSIS

In a previous work [28], SpMV, which is usually evaluated in the traditional context of native implementations (often in C or C++), was evaluated for the first time in the context of web-based engines for JavaScript and WebAssembly. The objective of our experiments is to extend their work by performing a matrix-structure based analysis of the performance and the choice of storage format for serial SpMV computation. We hope that these results will enhance the applicability of the best practices for web-based SpMV while considering the sparsity structure of the matrices and using modern web technologies for scientific, big data and deep learning web applications.

First, in **RQ1**, we analyze how the best choice of storage format for SpMV operation is related to the input sparse matrix structure. Next, **RQ2** further investigates the effect of sparse matrix structure on the SpMV performance within each sparse storage format. Finally, **RQ3**

Listing 3: SIMD portion of single-precision SpMV DIA implementation in WebAssembly

```
(loop $inner_loop1
  (local.get $this_y)
  (f32x4.mul (v128.load (local.get $this_data)) (
      v128.load (local.get $this_x)))
  (v128.load (local.get $this_y))
  (f32x4.add)
  (v128.store)
  (local.set $this_y (i32.add (local.get $this_y) (
      i32.const 16)))
  (local.set $this_data (i32.add (local.get $this_data) (
      i32.const 16)))
  (local.set $this_x (i32.add (local.get $this_x) (
      i32.const 16)))
  (local.tee $n (i32.add (local.get $n) (i32.const 4)))
  (local.get $iend)
  (i32.lt_s)
  (br_if $inner_loop1)
)
```

evaluates the interaction between the sparse matrix structure and the machine characteristics which together influence the performance of an SpMV operation.

4.1 RQ1: What is the Effect of Matrix Structure on the Choice of Storage Format?

In this section we evaluate the sparse matrix structure characteristics to justify the best choice of sparse storage format for SpMV operation for both C and WebAssembly. It has been shown in [28] that there are similarities and differences between native C and WebAssembly implementations for the choice of storage format for SpMV operation. Therefore, we examine if the same matrix features drive the storage format choice for both of them or not. We employ an x%-affinity [28] criteria to obtain the set of matrices best suited to each format category. 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. In addition to *single-format* categories (COO, CSR, DIA and ELL), there exist some cases where more than one format fulfils the *x%-affinity* metric versus the other formats but the performance between them cannot be distinguished, therefore *combination-format* categories were introduced.

4.1.1 DIA. In DIA SpMV kernel implementation, the outer loop iterates through each diagonal, and the inner loop iterates through the elements of a specific diagonal. To represent the matrices in this category, we analyze this feature called *dia_ratio* which is the ratio of the number of elements in the diagonals, *ndiag_elems* to the total number of non-zero elements in the sparse matrix, *nnz*.

$$dia\_ratio = \frac{ndiag\_elems}{nnz} \tag{1}$$

The *ndiag_elems* doesn't include the zeros padded around the diagonals to store them in a 2-dimensional array. These padded zeros are never accessed or processed in the DIA sequential SpMV kernel implementation because the starting point and the ending point of the diagonal are easily evaluated from its offset.

In [28], the unavailability of SIMD support for WebAssembly was described as the potential reason for the difference in the choice of storage format between C and WebAssembly for DIA matrices.

Therefore, our experiments with SIMD support find a lot of similarities in the choice of storage format between C and WebAssembly. Also, it is quite impressive to observe that the SpMV WebAssembly performance for DIA matrices demonstrated a geometric mean slowdown of only 1.14x versus C. The scatter plots in Figure 2 present the choice of storage format for each matrix, *DIA*, *combination-DIA* and *not-DIA* with its *dia_ratio* feature value marked on the x-axis, and its SpMV performance marked on the y-axis for both C and WebAssembly. To obtain these plots, we first categorize the matrices by determining the set of matrices that have 10%-affinity towards the DIA format illustrated as *DIA* in Figure 2. *combination-DIA* constitutes the set of matrices that have 10%-affinity to the DIA format as well as to some other formats. The *not-DIA* set of matrices are the ones which do not have 10%-affinity to the DIA format. We then extract the *dia_ratio* feature values for all the matrices. All the matrices with *dia_ratio* more than 5 belong to the *not-DIA* category, and hence for simplicity, are not shown on the plot.

From these plots, we make the following observations. First, matrices with *dia_ratio* value less than 3 show affinity towards the DIA format, except for a few matrices. After careful evaluations, we found that these exceptional matrices either in *combination-DIA* and *not-DIA* sets are the matrices with small dimensions (Stranke94, Trefethen_20b, cage4, to name a few). This makes the matrix dimension an important feature to consider especially for the small matrices. They fail to perform the best for DIA format because of the increased overhead of the inner loop of SpMV DIA implementation. Second, the SpMV DIA performance decreases with an increase in the *dia_ratio* feature which is as expected. However, the matrices with the same *dia_ratio* value also have different SpMV performance. It means there exist other matrix features which drive performance. This phenomenon will be studied in more depth when we explore the effect of sparse matrix structure on SpMV performance within the DIA format in **RQ2**.

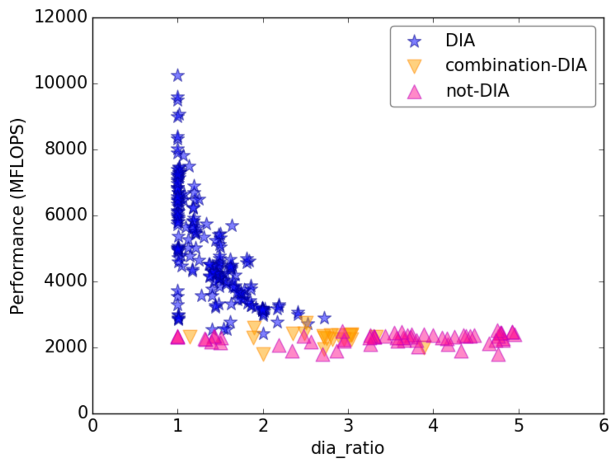
4.1.2 ELL. To represent the matrices in this category, we extract a feature we call *ell_ratio*, the ratio of *nell_elems* to the total number of non-zero elements in the sparse matrix, *nnz*.

$$ell\_ratio = \frac{nell\_elems}{nnz} \tag{2}$$

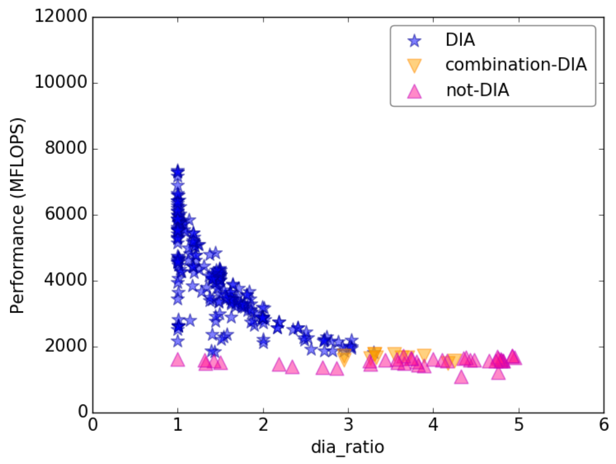
nell_elems is the product of the maximum number of non-zeros per row and the total number of rows in the matrix. Unlike DIA, *nell_elems* includes the padded zeros since they are processed by the ELL sequential SpMV kernel implementation.

In the scatter plots in Figure 3, we present the choice of storage format for each matrix, *ELL*, *combination-ELL* and *not-DIA-not-ELL*, with its *ell_ratio* feature value marked on the x-axis, and its SpMV performance marked on the y-axis for both C and WebAssembly. To obtain these plots, we categorize the matrices by determining the set of matrices that have 10%-affinity to the ELL format illustrated as *ELL* in Figure 3. We then extract the *ell_ratio* feature values for all the matrices. *Combination-ELL* constitutes the set of matrices that have 10%-affinity to the ELL format as well as to some other formats. We found that a number of matrices inside *not-ELL* category had a favourable *ell_ratio* feature value which could align the matrix towards ELL format. But the promising values of *dia_ratio* for these matrices pushed them to choose DIA as their best format. This high priority of *dia_ratio* over *ell_ratio* comes from the fact that the SpMV DIA implementation is a good candidate for loop-vectorization optimization, and the machines with SIMD capabilities tend to perform better for DIA format as compared to ELL format. Therefore, to show the true impact of *ell_ratio* on the matrices which are not DIA, we plot *not-DIA-not-ELL* category instead of *not-ELL*. Also, all the matrices with *ell_ratio* more than 3 belong to the *not-DIA-not-ELL* category, and hence for simplicity, are not shown on the plot.

It was found that a number of matrices which belong to *ELL* for C associate themselves with *combination-ELL* in the case of WebAssembly. We observe from these plots that the *ell_ratio* for all the ELL matrices is always close to 1. However, it is also true for some of the matrices from *combination-ELL* and *not-DIA-not-ELL* category. It means *ell_ratio* being close to 1 is not a sufficient condition to justify the affinity of a matrix towards the ELL format. Therefore, we



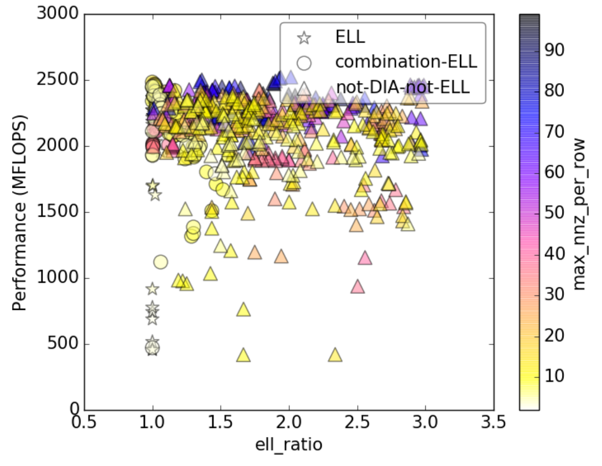
(a) C



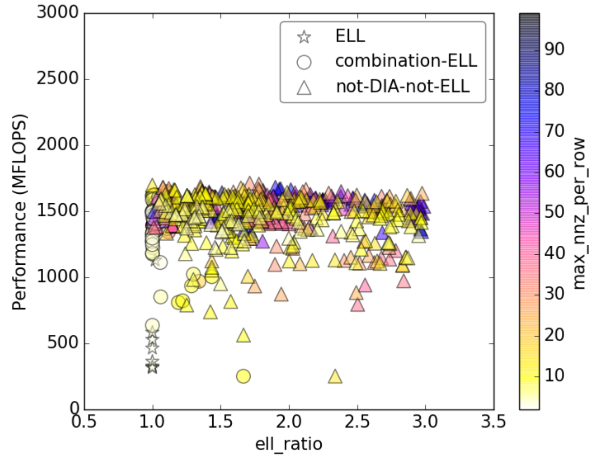
(b) WebAssembly

Figure 2: Effect of dia_ratio on the choice of storage format and perfomance for C and WebAssembly SpMV kernel using the 10%-affinity

explored another feature, *max_nnz_per_row* which is the maximum number of non-zeros per row in a sparse matrix as shown in Figure 3. It is now clear from the plot that the matrices with *ell_ratio* close to 1 show affinity towards ELL format if their value of *max_nnz_per_row* is very small. It increases the overhead of the inner loop in SpMV CSR implementation. It is also noticed that the matrices with similar *ell_ratio* and similar *max_nnz_per_row* values have different SpMV performance. Like DIA, this phenomenon will be studied in more depth based on some more features in **RQ2**.



(a) C

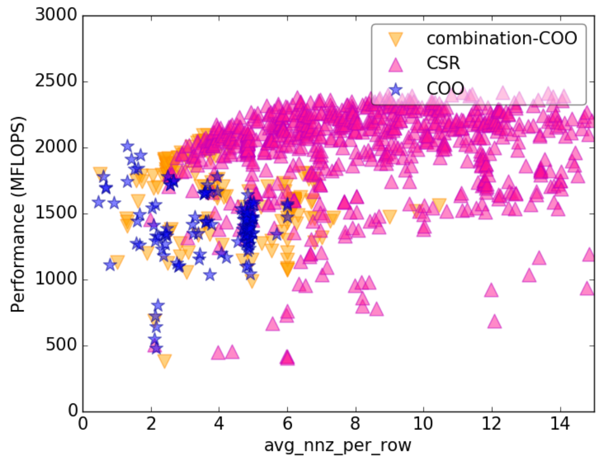


(b) WebAssembly

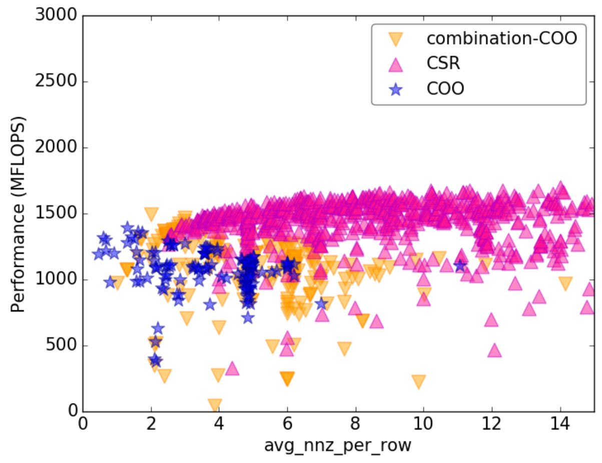
Figure 3: Effect of ell_ratio and max_nnz_per_row on the choice of storage format and perfomance for C and WebAssembly SpMV kernel using the 10%-affinity

4.1.3 COO. Sparse matrices tend to choose COO format over CSR for a number of reasons. In order to demonstrate that, we extract a feature called *avg_nnz_per_row* which is the ratio of the total number of non-zero elements in the sparse matrix, *nnz* to the number of rows of the sparse matrix, *N*.

$$avg\_nnz\_per\_row = \frac{nnz}{N} \tag{3}$$



(a) C



(b) WebAssembly

Figure 4: Effect of avg_nnz_per_row on the choice of storage format and perfomance for C and WebAssembly SpMV kernel using the 10%-affinity

In the scatter plots in Figure 4, we present the choice of storage format for each matrix, *COO*, *combination-COO* and *not-COO-not-DIA-not-ELL*, with its *avg_nnz_per_row* feature value marked on the x-axis, and its SpMV performance marked on the y-axis for both C and WebAssembly. To obtain these plots, we categorize the matrices by determining the set of matrices that have 10%-affinity to the COO format illustrated as *COO* in Figure 4. We then extract the *avg_nnz_per_row* feature values for all the matrices. *Combination-COO* constitutes the set of matrices that have 10%-affinity to the COO format as well as to some other formats. In order to exclude both the ELL and DIA matrices which have promising value for *avg_nnz_per_row* feature, we plot *CSR* category instead of *not-COO-not-ELL-not-DIA*. Also, all the matrices with *avg_nnz_per_row* more than 15 belong to the *CSR* category, and hence for simplicity, are not shown on the plot.

From these plots, we make the following observations. First, for some of the COO matrices, the number of non-zeros is less than the number of rows; e.g., *zenios* matrix, with N = 2873 and nnz = 1314. Such matrices choose COO format over CSR because in this case the size of CSR format becomes greater than the size of COO format, which is usually otherwise. Second, all of the COO matrices from our benchmark set have small values of *avg_nnz_per_row*. The small number of non-zeros per row increases the overhead of the inner loop of sequential SpMV CSR kernel implementation. However, it can also be observed that there are a number of CSR matrices with small values of *avg_nnz_per_row*. So, only based on this feature, one cannot distinguish between CSR and COO matrices. The large variation in the number of non-zeros per row increases the chances of branch mispredictions for the inner loop of the SpMV CSR kernel implementation. Therefore, the COO matrices are mostly with a small and uneven number of non-zeros per row. This phenomenon will be explained more in-depth based on the hardware features in **RQ3**. Also, due to the uneven number of non-zeros per row, ELL alone is not a suitable format for such a matrix. This was essentially the purpose of a new hybrid ELL/COO format proposed by Bell and Garland [9] to store the fixed number of non-zeros per row in the ELL format, and the rest of the non-zeros in the COO format.

4.1.4 CSR.

This format tends to become the best format for the matrices which are highly unsuitable for DIA, ELL and COO formats for the reasons highlighted in Table 1.

Table 1: Summary of relationship between storage format and structure features

Format	Feature(s)	Priority
DIA	*dia_ratio* $\leq$ 3 and large N	1
ELL	*ell_ratio* $\simeq$ 1 and small *max_nnz_per_row*	2
COO	nnz < N or small *avg_nnz_per_row* and uneven number of non-zeros per row	3

Overall, it is interesting to observe for all the formats that the SpMV performance in the case of C is slightly better than WebAssembly despite our efficient and equivalent WebAssembly implementations. Hence, we compared the x86 code generated by the V8's TurboFan compiler and the gcc compiler, and noticed a difference in their code generation strategy: gcc uses memory addressing modes available for x86-64 instruction set architecture, while the V8 compiler uses registers. For the latter this leads to more load and store instructions, and also high register pressure. On top of that, for every array access from linear memory, an extra operation is performed to calculate the effective address using a fixed offset.

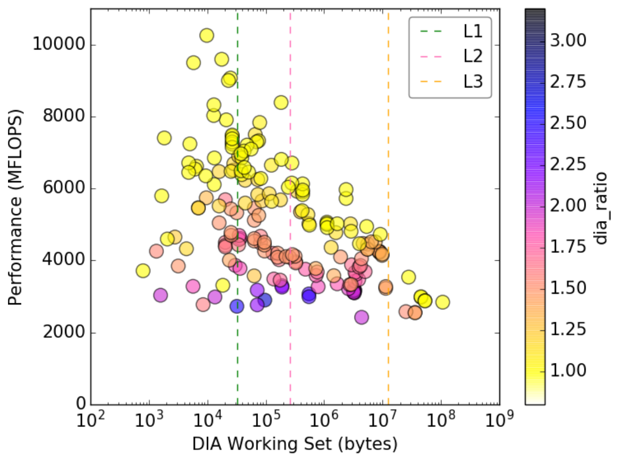
4.2 RQ2: What is the Effect of Matrix Structure on SpMV Performance within a Storage Format?

In this section we evaluate the effect of the sparse matrix structure on the SpMV performance for the different categories of the matrices based on the storage format from the previous section.

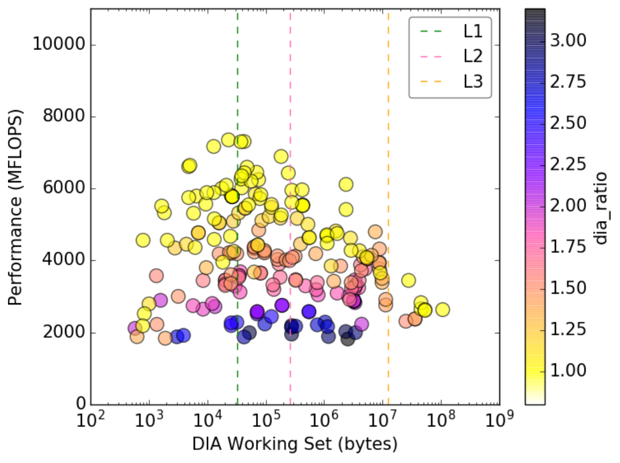
4.2.1 DIA.

The inherent diagonal structure of the DIA matrices allows to access the values of input vector x and output vector y contiguously providing both the spatial and temporal locality. So, this makes the *dia_ratio* a main feature which drives the performance along with the *DIA Working Set* for the matrices having affinity towards this format, where *DIA Working Set* is *ndiag_elems* + *num_diags* + 2 * *N*, *ndiag_elems* is the number of elements in the diagonals, *num_diags* is the number of diagonals, and *N* is the size of the input vector x or output vector y. A good performance is expected if the input vector x and output vector y fit into the cache, and the *dia_ratio* is close to 1.

The scatter plots in Figure 5 shows the combined impact of *DIA Working Set* and *dia_ratio* on the SpMV performance for both C and WebAssembly. It is quite evident that for a set of matrices with similar *dia_ratio*, the SpMV performance decreases with an increase in the working set size.



(a) C



(b) WebAssembly

Figure 5: Performance of the SpMV kernel in relation to the working set size for the DIA matrices using the 10%-affinity

4.2.2 ELL. As observed in **RQ1**, ELL matrices always have *ell_ratio* close to 1 and small *max_nnz_per_row*. Therefore, we evaluate other features in this section to understand how the structure of an ELL matrix affects its SpMV performance. Figure 6 shows the impact of *ELL Working Set* on the SpMV performance for the matrices having 10%-affinity towards this format for both C and WebAssembly. *ELL Working Set* is 2 * *nell_elems* + 2 * *N*, where *nell_elems* is the product of the maximum number of non-zeros per row and the total number of rows in the matrix, and *N* is the size of the input vector x or output vector y. It is clear that most of the matrices perform reasonably well, especially the *small*, *medium* and *large* matrices whose ELL Working Set is below the L3 cache size. For others, a potential reason for the performance degradation comes from the irregularity in their structure. We describe this irregularity via a metric called *ELL Locality Index*.

$$ELLLocalityIndex = \frac{\sum_{p=0}^{15} ELLReuseDistance[p]}{nnz} \times 100 \qquad (4)$$

The matrices with high *ELL Locality Index* tend to perform better than the matrices with low *ELL Locality Index*. The column index of each non-zero of the sparse matrix A is used to calculate this feature. Basically, it is based on the access pattern of the input vector x.

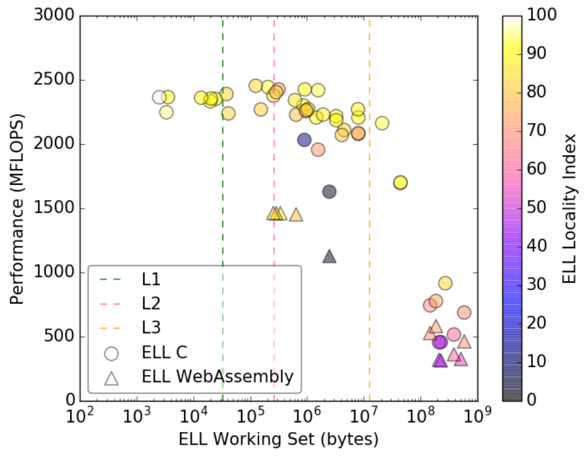


Figure 6: Performance of the SpMV kernel in relation to the working set size and ELL Locality Index for the ELL matrices using the 10%-affinity

Figure 7 shows the calculation of *ELL Locality Index* feature for an example matrix. First of all, the *Reuse Distance* is calculated for each non-zero. *Reuse Distance* is defined for each non-zero as the distance from the last non-zero whose column index corresponds to the same block of the input vector x. The distance here is measured in the unit of non-zero accesses. The size of a block of input vector x is determined based on the cache line size of the given system. Therefore, we use the word block interchangeably with the cache line. In modern systems, the cache line size is typically 64 bytes. We thus determine the number of elements in a cache line for a given floating-point precision. For example, for a single-precision SpMV computation, each cache line consists of 16 x-vector elements, assuming the elements

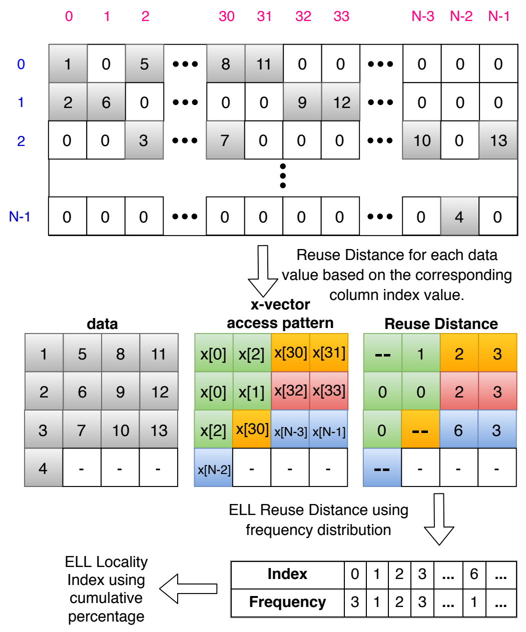
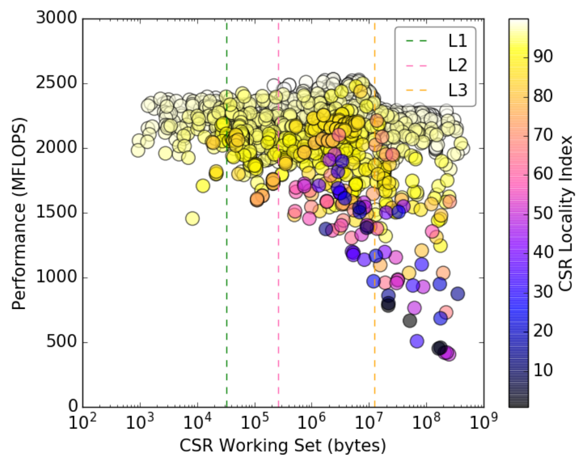

Reuse Distance for each data value based on the corresponding column index value.
data
x-vector access pattern
Reuse Distance
ELL Reuse Distance using frequency distribution
ELL Locality Index using cumulative percentage
Index
Frequency

Figure 7: An example for ELL Locality Index calculation

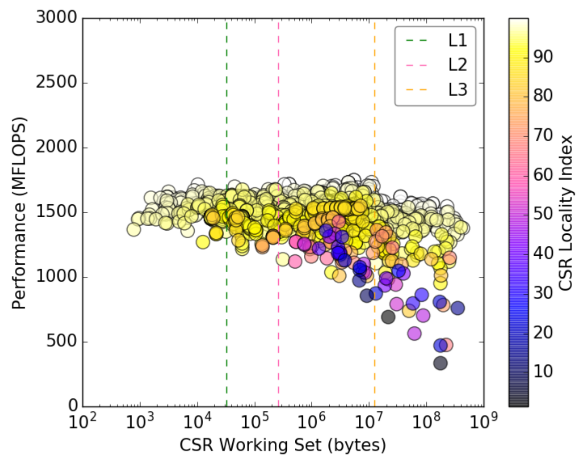
from index 0 to 15 belong to the same cache line and so forth (the same-coloured squares in the x-vector access pattern in Figure 7 belong to the same cache line). For simplification, we also assume here that the matrix dimension *N* is in the power of 2. It is important that the *Reuse Distance* is calculated in the same order as the non-zeros accessed in the ELL SpMV computation. Therefore, it is straightforward to measure it on the `indices` array of the sparse matrix in the ELL format which directly indicates the x-vector access pattern.

For the input vector x, if an element from a block is accessed, and this block is accessed for the first time, then the *Reuse Distance* is not calculated for this access and illustrated as '--' in Figure 7 since this block has not been reused at all. However, we label this block with the position of the respective non-zero for which this x element was used. Thereafter, the *Reuse Distance* will be measured for all the following accesses to this block by subtracting the block label from the position of the corresponding non-zero, with the corresponding block label updating with each non-zero access to keep track of the position at which this block was previously used. Following the calculation of *Reuse Distance* for all the non-zeros, *ELL Reuse Distance* is calculated using frequency distribution. It means the *ELL Reuse Distance[p]* is the number of non-zeros of sparse matrix A stored in the ELL format which access the input vector x with p *Reuse Distance*. This feature counts both temporal and spatial locality for the input vector x. The matrices with a more regular structure will have low values of *Reuse Distance* for most of its non-zeros, while the matrices with irregular structure will have a few non-zeros in the lower range of *ELL Reuse Distance*. Finally, the *ELL Locality Index* is calculated from *ELL Reuse Distance* using cumulative percentage where *Reuse Distance* is from 0 to *Best Reuse Distance*. We chose the *Best Reuse Distance* to be 15 based on our experiments. This data locality based irregular structure is observed for a matrix called `wing`, whose working set lies between L2 and L3 cache, and its SpMV performance is around 1600 MFLOPS in the case of C, which is substantially lower than other matrices with the regular structure in the same working set range.

4.2.3 CSR. In **RQ1** we described CSR as being chosen as the fallback method when no other specialized format can be employed for a given sparse matrix. In short, no structural feature seems to play a role in making CSR as the best choice. However, the structure of a given CSR matrix indeed governs its SpMV performance. Figure 8 shows the impact of *CSR Working Set* on the SpMV performance for the matrices having 10%-affinity towards this format for both C and WebAssembly. The *CSR Working Set* is : $(N + 1) + 2 * nnz + 2 * N$, where *nnz* is the number of non-zeros in the matrix, and *N* is the number of rows/columns. Regardless of the increasing working set size, it is observed that the SpMV performance remains consistently good for most of the matrices, although there exists a set of matrices which do not perform equally well. In order to understand this peculiar behavior, we investigated the difference between the structure of the matrices in these two sets. Like ELL matrices, one potential reason for this performance difference comes from the irregularity in their structure. Therefore, we introduce a new feature called *CSR Reuse Distance* which finally contributes to the calculation of *CSR Locality Index*. The purpose of this feature is to formally define the irregularity or regularity in the matrix structure. The design of this feature is based on the reuse distance concept and the two aspects of regularity in the structure of the matrix: spatial locality for the non-zeros in a row, and temporal locality for the non-zeros in the neighbouring rows.



(a) C

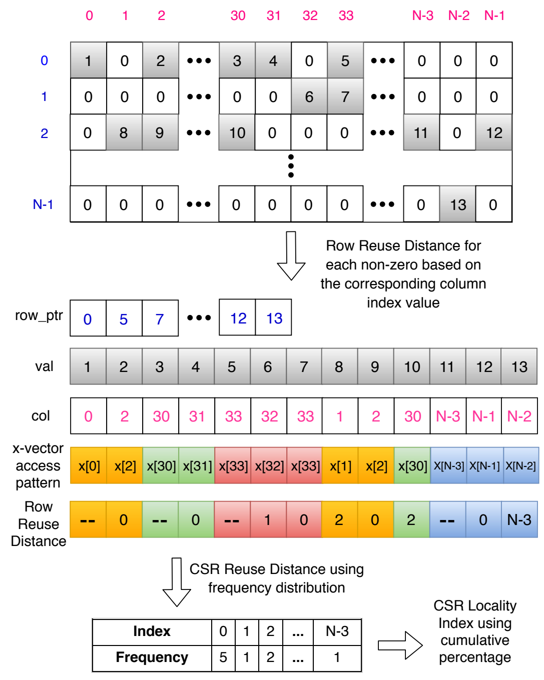


(b) WebAssembly

Figure 8: Performance of the SpMV kernel in relation to the working set size and CSR Locality Index for the CSR matrices using the 10%-affinity

$$CSRLocalityIndex = \frac{\sum_{p=0}^{15} CSRReuseDistance[p]}{nnz} \times 100 \tag{5}$$

Figure 9 shows the calculation of *CSR Locality Index* feature for an example matrix. First of all, the *Row Reuse Distance* is calculated for each non-zero. *Row Reuse Distance* is defined for each non-zero as the distance from the last non-zero whose column index corresponds to the same block of the input vector x. Like ELL, the size of a block of input vector x is determined based on the cache line size of the given system, and the word block is used interchangeably with the cache line. Unlike in ELL, the distance here is measured in the unit of rows. It is important to note that the *Row Reuse Distance* is calculated on the input vector x for each non-zero.



0 1 2 30 31 32 33 N-3 N-2 N-1
0 1 2 N-1
Row Reuse Distance for each non-zero based on the corresponding column index value
row_ptr
val
col
x-vector access pattern
Row Reuse Distance
CSR Reuse Distance using frequency distribution
Index
Frequency
CSR Locality Index using cumulative percentage

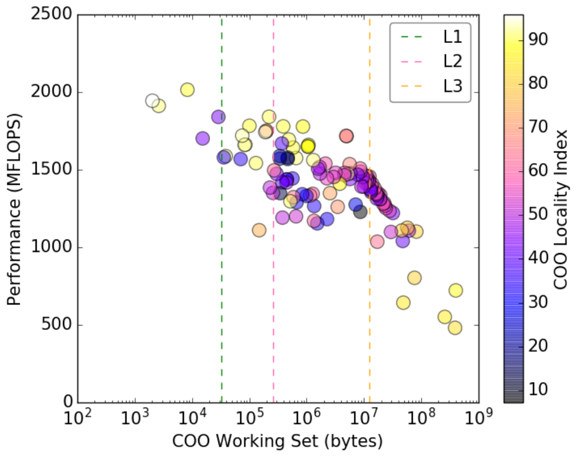
Figure 9: An example for CSR Locality Index calculation

For the input vector x, if an element from a block is accessed, and this block is accessed for the first time, then the *Row Reuse Distance* is not calculated for this access. However, we label this block with the row number of the respective non-zero for which this x element was used. The *Row Reuse Distance* will be measured for all the following accesses to the same block by subtracting the block label from the row number of the corresponding non-zero. Also, the corresponding block label keeps on updating to keep track of the row in which this block was previously used. If two elements from the same block are accessed in a given row, then the second access will have a zero *Row Reuse Distance* because the first access will set the block label to the same row number. The first access could be the first access of this block which in that case will not count towards the *CSR Reuse Distance*. Another possibility is that this cache line was accessed previously in another row which will result in a non-zero *Row Reuse Distance* for the first access. Following the calculation of *Row Reuse Distance* for all the non-zeros, *CSR Reuse Distance* is calculated using frequency distribution. It means the *CSR Reuse Distance[p]* is the number of non-zeros of sparse matrix A stored in the CSR format which access the input vector x with p *Row Reuse Distance*. Finally, the *CSR Locality Index* is calculated from *CSR Reuse Distance* using a cumulative percentage where *Row Reuse Distance* is from 0 to *Best Row Reuse Distance*. We chose the *Best Row Reuse Distance* to be 15 based on our experiments.

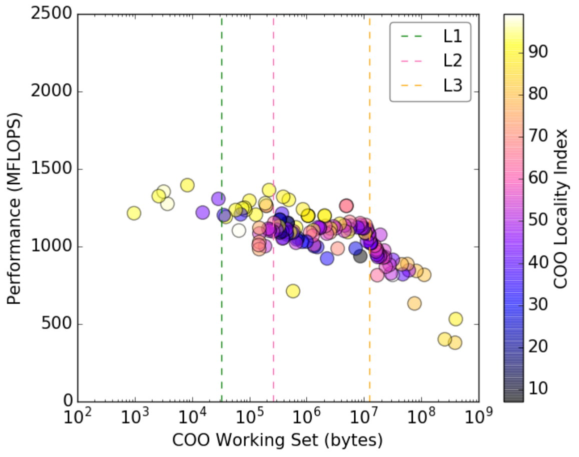
With the *CSR Locality Index* feature in place on Figure 8, we can clearly follow the effect of irregularity based on data locality on the SpMV performance. The power of this parameter is evident from the fact that it makes it easy to identify the reason for the difference in SpMV performance of two sparse matrices of similar working set size. Heras et al. [16] also modeled the data locality for SpMV CSR execution. It was based on the number of entry matches and the number of block matches between the pairs of consecutive rows which only accounted for limited locality between the pairs of consecutive rows. Therefore, it was generalized on the pairs of consecutive group of rows by selecting the best window size ranging between 1 to 64. However, the whole process of calculating the number of entry matches and the number of block matches needs to be repeated for each window size to finally select the best window size. In contrast to that, we proposed a simpler and more efficient approach where the *Row Reuse Distance* is calculated for each non-zero access, and the *Best Row Reuse Distance* that corresponds to twice the best window size is chosen at the end after the calculation of *CSR Reuse Distance* which finally leads to the evaluation of *CSR Locality Index* with just a simple summation. Another factor that affects the SpMV performance of CSR matrices is the irregular number of non-zeros per row which is studied in more detail in **RQ3**.

4.2.4 *COO.*

Figure 10 shows the impact of *COO Working Set* on the SpMV performance for the matrices having affinity towards this format for both C and WebAssembly. The *COO Working Set* is 3 * *nnz* + 2 * *N*, where *nnz* is the number of non-zeros in the matrix, and *N* is the number of rows/columns. It is clear from the plot that the SpMV performance of the COO matrices is directly affected by the working set of the problem. We investigated the performance of COO matrices in relation to the *COO Locality Index* (calculated the same as *CSR Locality Index*). However, we found that a number of matrices have a mediocre value of this feature but their performance is as good as the ones with the higher value of this feature. It means that the *COO Locality Index* doesn't have the same impact on the COO matrices as the *CSR Locality Index* has on the CSR matrices. It is likely due to the higher number of rows with no elements in the COO matrices than CSR matrices. It is also reasonable to point out that at any point of time in COO SpMV computation, there is more memory traffic as compared to other formats.



(a) C



(b) WebAssembly

Figure 10: Performance of the SpMV kernel in relation to the working set size and COO Locality Index for the COO matrices using the 10%-affinity

4.3 RQ3: What is the Effect of Interaction between Matrix Structure and Hardware Characteristics on the SpMV Performance?

In this section we evaluate the relationship between the sparse matrix structure and the hardware features which together influence the SpMV performance. We count the true hardware performance counters through a performance API tool (PAPI) [30] while executing SpMV on our benchmark set for C.

4.3.1 *Caches and Main Memory.*

In **RQ2**, the features designed based on the data locality model have their roots in the hardware features like data cache misses. In order to validate our evaluations, we measure the cache misses for L1, L2 and L3 cache levels using PAPI_L1_DCM, PAPI_L2_DCM and PAPI_L3_TCM PAPI events per SpMV operation. The percentage of the number of cache misses for an SpMV operation over the total number of non-zeros in the sparse matrix provides a good measure for representing data locality.

$$Index = \frac{PAPI\_L1\_DCM \vee PAPI\_L2\_DCM \vee PAPI\_L3\_TCM}{nnz} \times 100 \tag{6}$$

It is important to note here that cache misses from a specific cache level are considered for calculating this percentage index based on the dimension of the sparse matrix. For example, if double the size of the vector x fits the L3 cache but it doesn't fit in L2 cache, then the number of cache misses from L2 cache is used. A low percentage number means fewer cache misses for vector x, and hence a good data locality. In contrast, a high percentage number means poor data locality for a sparse matrix as shown in Figure 11.

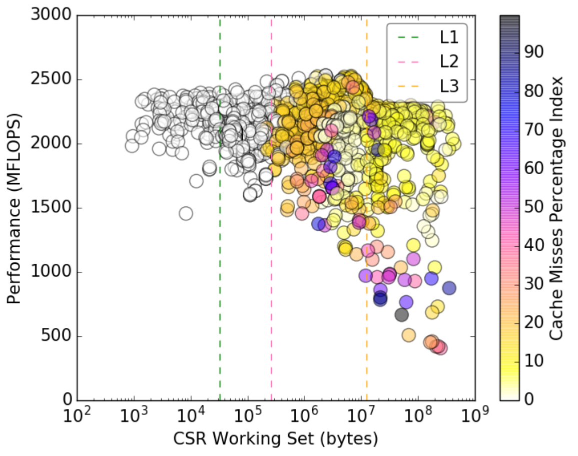


Figure 11: Performance of the SpMV kernel in relation to the working set size and cache misses percentage index for the CSR matrices using the 10%-affinity

4.3.2 Branch Prediction Unit. In **RQ1**, the branch mispredictions is described as the potential reason for some of the matrices to choose COO format over the CSR format for the SpMV computation. When the number of non-zeros per row are uneven, it increases the chances of incorrectly predicting the direction of the conditional branch if the matrices are stored in the CSR format. In order to verify this, we store both COO and CSR matrices in the CSR format and measure the correct branch predictions and branch mispredictions using PAPI_BR_PRC and PAPI_BR_MSP events per SpMV operation. We then compute the percentage of branch mispredictions over the total conditional branch predictions.

$$Index = \frac{PAPI\_BR\_MSP}{PAPI\_BR\_PRC + PAPI\_BR\_MSP} \times 100 \tag{7}$$

Figure 12 shows the SpMV performance of COO and CSR matrices with respect to their *avg_nnz_per_row* and the calculated index. Also, in **RQ2**, the branch misprediction rate in combination with

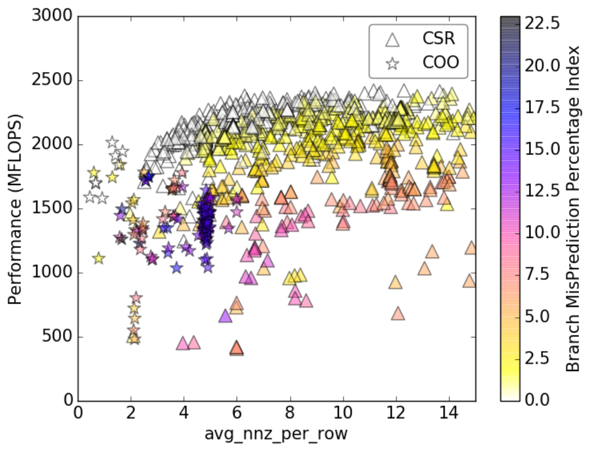


Figure 12: Performance of the SpMV kernel for COO and CSR matrices in relation to the avg_nnz_per_row and branch misprediction percentage index using the 10%-affinity

CSR Locality Index affects the SpMV performance within the CSR matrices as shown in Figure 13. It is quite clear that a high percentage index value, potentially due to the irregular number of non-zeros per row, leads to SpMV performance slowdown.

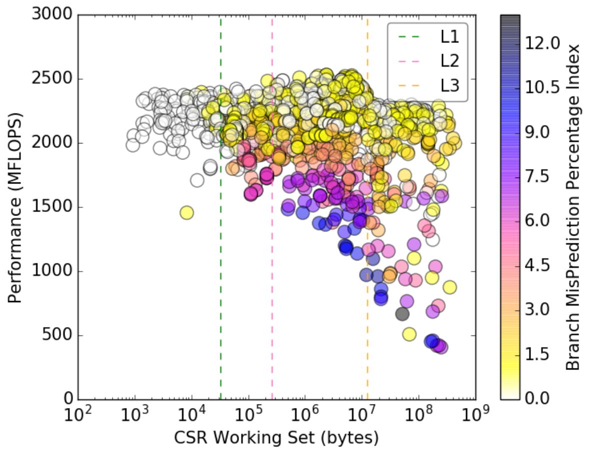


Figure 13: Performance of the SpMV kernel in relation to the working set size and branch misprediction percentage index for the CSR matrices using the 10%-affinity

5 CONCLUSION AND FUTURE WORK

Our work provides valuable insights about the factors which control the SpMV performance. First, we found that some matrices which indeed belong to a *single-format* category can have favourable values for multiple structure features that can indicate their affinity towards more than one format. This is especially true for DIA matrices which have feature values that also fit ELL format criteria. However, the application of SIMD vectorization optimization for SpMV DIA implementation prioritized DIA to be selected as the optimal format. This supports the argument that the optimal choice of storage format is governed both by the structure of the matrix and the code optimization opportunities available for the given programming language and the machine architecture. Second, the SpMV performance suffers in the case of WebAssembly for Chrome browser. V8's code generation currently does not utilize the advantageous memory addressing modes available in the x86 instruction set. Therefore, some of the matrices, which belong to the *single-format* category based on the *10%-affinity* criteria in the case of C, show affinity towards the *combination-format* category when executed for WebAssembly. Third, our structure features modeled based on the data locality, provide the capability of estimating if the SpMV performance for a particular matrix is affected by the irregular memory accesses for vector x or not. Lastly, we validate our evaluations and parameter choices using hardware performance counters.

In our future work we wish to further explore how the impact of additional hardware features on SpMV performance can be quantified via matrix structure features. The future improvements to V8 TurboFan's code generation strategies, and the addition of some new instructions like gather/scatter to the WebAssembly instruction set will provide new optimization opportunities for web-based sparse computations. We also intend to use the methodology established in this paper along with the upcoming browser-based technologies like web workers to examine parallel versions of web-based SpMV.

REFERENCES

[1] 2018. argon.js - A javascript framework for adding augmented reality content to web applications. https://www.argonjs.io

[2] 2018. AR.js - Augmented Reality for the Web. https://github.com/jeromeetienne/ar.js

[3] 2018. brain.js - Neural networks in JavaScript. https://github.com/BrainJS/brain.js

[4] 2018. Features to add after the MVP. http://webassembly.org/docs/future-features

[5] 2018. ml.js - A k-nearest neighbour classifier algorithm. https://github.com/mljs/knn

[6] 2019. AutoCAD Web App. https://web.autocad.com/

[7] 2019. Photo Editor | Online Photoshop Lightroom. https://lightroom.adobe.com

[8] R. C. Agarwal, F. G. Gustavson, and M. Zubair. 1992. A High Performance Algorithm Using Pre-processing for the Sparse Matrix-vector Multiplication. In *Proceedings of the 1992 ACM/IEEE Conference on Supercomputing (Supercomputing '92)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 32–41. http://dl.acm.org/citation.cfm?id=147877.147901

[9] Nathan Bell and Michael Garland. 2009. Implementing Sparse Matrix-vector Multiplication on Throughput-oriented Processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC '09)*. ACM, New York, NY, USA, Article 18, 11 pages. https://doi.org/10.1145/1654059.1654078

[10] Ronald F Boisvert, Roldan Pozo, Karin Remington, Richard F Barrett, and Jack J Dongarra. 1997. Matrix market: a web resource for test matrix collections. In *Quality of Numerical Software*. Springer, 125–137.

[11] Timothy A Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)* 38, 1 (2011), 1.

[12] Chen Ding and Yutao Zhong. 2003. Predicting Whole-program Locality Through Reuse Distance Analysis. In *PLDI '03*. ACM, New York, NY, USA, 245–257. https://doi.org/10.1145/781131.781159

[13] A. Elafrou, G. Goumas, and N. Koziris. 2017. Performance Analysis and Optimization of Sparse Matrix-Vector Multiplication on Intel Xeon Phi. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 1389–1398. https://doi.org/10.1109/IPDPSW.2017.134

[14] G. Goumas, K. Kourtis, N. Anastopoulos, V. Karakasis, and N. Koziris. 2008. Understanding the Performance of Sparse Matrix-Vector Multiplication. In *16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008)*. 283–292. https://doi.org/10.1109/PDP.2008.41

[15] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 185–200.

[16] D.B. Heras, J.C. Cabaleiro, and F.F. Rivera. 2001. Modeling data locality for the sparse matrix-vector product using distance measures. *Parallel Comput.* 27, 7 (2001), 897 – 912. https://doi.org/10.1016/S0167-8191(01)00089-8

[17] David Herrera, Hanfeng Chen, Erick Lavoie, and Laurie Hendren. 2018. Numerical Computing on the Web: Benchmarking for the Future. In *Proceedings of the 14th ACM SIGPLAN International Symposium on Dynamic Languages (DLS 2018)*. ACM, New York, NY, USA, 88–100. https://doi.org/10.1145/3276945.3276968

[18] Eun-Jin Im, Katherine Yelick, and Richard Vuduc. 2004. Sparsity: Optimization framework for sparse matrix kernels. *The International Journal of High Performance Computing Applications* 18, 1 (2004), 135–158.

[19] Abhinav Jangda, Bobby Powers, Emery D. Berger, and Arjun Guha. 2019. Not So Fast: Analyzing the Performance of Webassembly vs. Native Code. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '19)*. USENIX Association, Berkeley, CA, USA, 107–120. http://dl.acm.org/citation.cfm?id=3358807.3358817

[20] Eric Jones, Travis Oliphant, Pearu Peterson, et al. 2001–. SciPy: Open source scientific tools for Python. http://www.scipy.org/ [Online; accessed <today>].

[21] Andrej Karpathy. 2014. ConvNetJS: Deep learning in your browser (2014). *URL http://cs. stanford. edu/people/karpathy/convnetjs* (2014).

[22] Kornilios Kourtis, Vasileios Karakasis, Georgios Goumas, and Nectarios Koziris. 2011. CSX: An Extended Compression Format for Spmv on Shared Memory Systems. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP '11)*. ACM, New York, NY, USA, 247–256. https://doi.org/10.1145/1941553.1941587

[23] Jiajia Li, Guangming Tan, Mingyu Chen, and Ninghui Sun. 2013. SMAT: An Input Adaptive Auto-tuner for Sparse Matrix-vector Multiplication. In *PLDI'13*. ACM, 117–126.

[24] Weifeng Liu and Brian Vinter. 2015. CSR5: An Efficient Storage Format for Cross-Platform Sparse Matrix-Vector Multiplication. In *ICS'15*. ACM, 339–350.

[25] Nikhil Thorat, Daniel Smilkov, and Charles Nicholson. [n.d.]. TensorFlow.js - A WebGL accelerated browser based JavaScript library for training and deploying ML models. https://js.tensorflow.org [Online; accessed <today>].

[26] Ali Pinar and Michael T. Heath. 1999. Improving Performance of Sparse Matrix-vector Multiplication. In *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing (SC '99)*. ACM, New York, NY, USA, Article 30. https://doi.org/10.1145/331532.331562

[27] Yousef Saad. 1994. SPARSKIT: a basic tool kit for sparse matrix computations.

[28] Prabhjot Sandhu, David Herrera, and Laurie Hendren. 2018. Sparse Matrices on the Web: Characterizing the Performance and Optimal Format Selection of Sparse Matrix-vector Multiplication in JavaScript and WebAssembly. In *Proceedings of the 15th International Conference on Managed Languages & Runtimes (ManLang '18)*. ACM, New York, NY, USA, Article 6, 13 pages. https://doi.org/10.1145/3237009.3237020

[29] O. Temam and W. Jalby. 1992. Characterizing the Behavior of Sparse Algorithms on Caches. In *Proceedings of the 1992 ACM/IEEE Conference on Supercomputing (Supercomputing '92)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 578–587. http://dl.acm.org/citation.cfm?id=147877.148091

[30] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. 2010. Collecting Performance Data with PAPI-C. In *Tools for High Performance Computing 2009*, Matthias S. Müller, Michael M. Resch, Alexander Schulz, and Wolfgang E. Nagel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 157–173.

[31] S. Toledo. 1997. Improving the memory-system performance of sparse-matrix vector multiplication. *IBM Journal of Research and Development* 41, 6 (Nov 1997), 711–725. https://doi.org/10.1147/rd.416.0711

[32] K. R. Townsend, S. Sun, T. Johnson, O. G. Attia, P. H. Jones, and J. Zambreno. 2015. k-NN text classification using an FPGA-based sparse matrix vector multiplication accelerator. In *2015 IEEE International Conference on Electro/Information Technology (EIT)*. 257–263. https://doi.org/10.1109/EIT.2015.7293349

[33] Richard Vuduc, James W Demmel, and Katherine A Yelick. 2005. OSKI: A library of automatically tuned sparse matrix kernels. In *Journal of Physics: Conference Series*, Vol. 16. IOP Publishing, 521.

[34] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. 2014. Intel math kernel library. In *High-Performance Computing on the Intel® Xeon Phi*. Springer, 167–188.

[35] Y. Wang, H. Yan, C. Pan, and S. Xiang. 2011. Image editing based on Sparse Matrix-Vector multiplication. In *2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 1317–1320. https://doi.org/10.1109/ICASSP.2011.5946654

[36] James B. White, III and P. Sadayappan. 1997. On Improving the Performance of Sparse Matrix-Vector Multiplication. In *Proceedings of the Fourth International Conference on High-Performance Computing (HIPC '97)*. IEEE Computer Society, Washington, DC, USA, 66–. http://dl.acm.org/citation.cfm?id=523991.938962

[37] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. 2007. Optimization of Sparse Matrix-vector Multiplication on Emerging Multicore Platforms. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing (SC '07)*. ACM, New York, NY, USA, Article 38, 12 pages. https://doi.org/10.1145/1362622.1362674