McGill University
School of Computer Science
Sable Research Group

Efficient Web-Based Parallel Sparse Matrix-Vector Multiplication

Sable Technical Report No. McLAB-2020-11

Prabhjot Sandhu, Clark Verbrugge and Laurie Hendren

April 22, 2021

Contents

List of Figures

List of Tables

Abstract

Sparse Matrix-Vector Multiplication (SpMV) is a computational kernel of a wide range of scientific computations, including popular targets like machine learning. Web-based implementations simplify access to such computations, but are limited by the performance of web-based programming languages. In this work we describe a scalable and highly optimized parallel SpMV implementation based on WebAssembly. Our design builds on a stack of optimizations, exploiting different sparse storage formats as well as novel matrix properties and machine-level performance characteristics. We perform exhaustive experiments with 2000 real-life sparse matrices to evaluate the effect of our optimizations and performance relative to a static, C baseline. Using our stack of optimizations, we achieve similar performance to the Intel MKL C library, showing that a web-based design can offer performance competitive with even highly tuned and well established native implementations.

Efficient Web-Based Parallel Sparse Matrix-Vector Multiplication

Prabhjot Sandhu, Clark Verbrugge and Laurie Hendren

April 22, 2021

1 Introduction

Many scientific applications have been ported to web-based implementations, allowing for easier and more open access to useful, but non-trivial computations; examples can be found in image editing [6], computer-aided design [5], augmented reality [1, 2], text classification [17, 3, 4] and deep learning [24]. Large and sparse matrices are frequently encountered in this context, with *sparse matrix-vector multiplication* (SpMV) being a core operation [35, 32], computing $y = Ax$, where matrix A is sparse and vector x is dense. Efficient, client web-based execution, however, requires significant optimization, made more difficult by limitations of dynamic web languages like JavaScript, and the high-level execution context provided by web browsers. This potentially affects the choice of sparse matrix storage format, and limits the ability to exploit shared-memory parallelism and generally optimize for low-level machine characteristics, all of which are known to strongly affect SpMV performance [28, 29].

In this work we describe and analyze a web-based SpMV implementation. Our system leverages *WebAssembly* [12] (abbreviated *wasm*), a low-level web language designed to augment JavaScript and intended to offer near-native performance. This allows us to explore use of shared-memory, and enables multiple, low-level approaches to optimization, including use of vector instructions. We consider the most common storage formats, specializing our optimization and parallelization strategies accordingly. We show that most language and execution-context limitations can be overcome, with our implementation achieving performance similar to a highly optimized C implementation. Specific contributions of our work include:

- We describe a highly optimized and parallel implementations of SpMV in a web context. Our approach exploits multiple storage formats (COO, CSR, DIA, ELL) [27], thread partitioning, and machine-level optimization opportunities afforded by the low-level wasm interface.
- We verify performance using data from around 2000 real-life sparse matrices from The SuiteSparse Matrix Collection [10]. We compare performance with native implementations, the *taco* C++ library [18, 9] and the Intel MKL C library [34]. We achieve significant improvement over the former, and competitive performance with the latter.

2 Motivation and Related Work

Research work on analyzing SpMV dates back to nearly three decades ago, and provides considerable evidence that SpMV performance depends on the structure of the matrix [7] and the memory system [30, 13].

A plethora of research work has since been dedicated towards accelerating SpMV performance. Some proposed new sparse storage formats [22, 15, 33, 19, 21], and some applied different optimization techniques for the basic sparse matrix storage formats, with *CSR* format being a particular focus [31, 36, 26, 37]. More recently, the growing importance of SpMV in a web-context, fueled by a wide variety of use cases, including big-data analytics, image processing [35] and machine learning algorithms like Support Vector Machines (SVM) [25], Sparse Convolutional Neural Networks (CNNs) [8], logistic regression and more, motivated several studies [14, 28, 29] that analyzed the web-based SpMV performance.

Experiments by Herrera et. al [14] using the Ostrich Numerical Benchmark set, which included SpMV CSR kernel as one of their "dwarfs" showed the WebAssembly performance within 2x of native C across many devices, with one device achieving better overall performance than native C. Sandhu et al. [28] presented serial SpMV performance numbers for JavaScript and Emscripten-generated WebAssembly as compared to C for both single- and double-precision floating point representations. They introduced the notion of x%-affinity to identify which, if any, storage format performs at least x% better than all other formats for a given sparse matrix. They also highlighted the differences in the choice of storage format between native C and web environments. All of these have inspired our research towards parallel SpMV performance analysis and optimizations on modern systems and web-based execution environments. The development of such web-based scientific computing framework which provides optimized and parallel Sparse BLAS (Basic Linear Algebra Subprograms) routines like SpMV has clear applicability as a high-performance backend for web-based ML frameworks like TensorFlow.js [24] which train and deploy models in the browser.

Despite the numerous attempts at developing new sparse storage formats, there is no one-size-fits-all storage format for different sparse matrices. Interesting work, however, has been proposed to predict the best storage format for SpMV computation based on the matrix structure and hardware features using machine learning techniques [23, 20]. The features used have an obvious impact on the effectiveness of these approaches. Sandhu et al. [29] investigated and presented a number of matrix-structure features that dictate the performance and the choice of storage format on a uniprocessor web-based system, comparing hand-tuned WebAssembly to C. Our approach instead focuses on analyzing how the matrix structure affects the choice of storage format in the presence of many optimizations and parallel techniques.

Several specific optimizations have been explored in the past to fully unleash the potential of different systems. For instance, Williams et al. [37] presented many optimizations especially for emerging multicore platforms like cache and register blocking, TLB blocking, loop optimizations, software prefetching etc. to show significant improvements on a set of 14 sparse matrices collected from a variety of actual applications. The use of different optimizations in different cases continues to inspire work on identifying relevant sparse matrix features. Elafrou et al. [11], for example, classify SpMV performance bottlenecks in terms of being memory-bandwidth bound, memory-latency bound, having thread imbalance, or due to computational bottlenecks within the CSR format, and apply supervised learning to build a feature-guided classifier, trained on a set of 210 matrices. In relation to that, our work presents the design of specific low-level code and data optimizations that have a major impact on web-based systems.

3 WebAssembly Parallel SpMV Overview

In this section we present our basic approach to a parallel implementation infrastructure for SpMV using WebAssembly and JavaScript. This includes initial experiments to examine baseline performance and scalability.

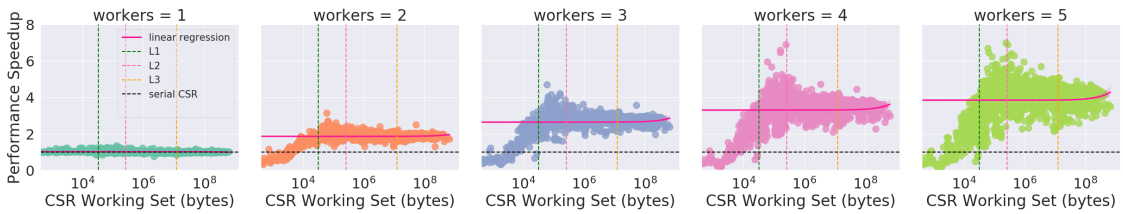


Figure 1: Parallel SpMV performance speedup over serial SpMV for different numbers of threads. Each circle represents a matrix, and the x-axis shows the working set size, with dashed vertical lines marking the size of the L1, L2, and L3 caches.

3.1 WebAssembly SpMV Parallel Infrastructure

WebAssembly [12] is a new stack-based virtual ISA binary code format with a corresponding textual format. It is a low-level, assembly-like execution context that is simpler to optimize and more directly maps to the instructions of common hardware architectures. It has been designed to complement and run alongside JavaScript, and aims to overcome the performance limitations of JavaScript by bringing near-native speeds to the web.

A number of recent research contributions explored the performance of compute-intensive tasks in WebAssembly. Haas et. al [12] reported the performance within 2x of native C for all benchmarks in the PolyBenchC suite, with 7 of them within 10% of native C. Jangda et al. [16] 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.

In this work, we have enabled task parallelism for WebAssembly via JavaScript's web workers, which map onto OS-level threads and provides a shared memory paradigm, an important consideration for parallel, data-intensive computations. First, the main JavaScript thread creates a shared WebAssembly memory which is basically a resizable SharedArrayBuffer object. Memory is managed using the malloc/free WebAssembly module extracted from Emscripten [38] (a C $\rightarrow$ wasm translator) source code to simplify memory allocation. The main thread then creates WebAssembly threads by spawning web workers, which in turn instantiate our WebAssembly module containing the SpMV implementation. After allocating and loading the input and output vectors, along with the input sparse matrix in the desired internal storage format, these WebAssembly threads perform SpMV in parallel. Note that WebAssembly does not (yet) provide fine-grain synchronization primitives, and thus threads require independent work.

3.2 Experimental Testbed

We conducted our experiments on an Intel Core i7-3930K with 6 $\times$ 3.20GHz cores, 12MB last-level cache and 16GB memory, running Ubuntu Linux 18.04.5. We translated our hand-tuned SpMV implementations in the wasm textual format to the corresponding binary format using the WebAssembly Binary Toolkit (wabt) version 1.0.13. Our WebAssembly execution environment is Chrome 80 browser (Official build 80.0.3987.149 with V8 JavaScript engine 8.0.426). We run headless Chrome with a flag `--experimental-wasm-simd` to enable the use of SIMD (Single Instruction Multiple Data) instructions for loop vectorizations in our 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.

Our set of sparse matrix benchmarks, to evaluate the performance of our SpMV implementations, consists

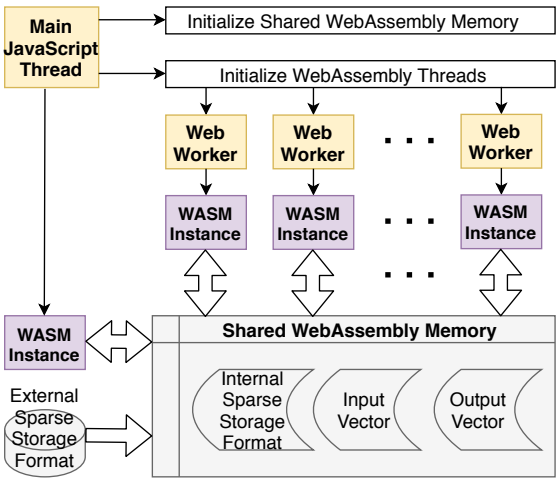


Figure 2: Overview of our SpMV parallel implementation infrastructure on WebAssembly and JavaScript

of 1,962 real-life square sparse matrices from The SuiteSparse Matrix Collection [10].

3.3 SpMV Performance and Scalability

Our baseline scalability experiments are aimed at the most popular format, CSR, where we tested single-precision parallel SpMV performance using 1 to 5 threads. Work can be distributed using simple *row-partitioning*, giving the same number of rows to each thread. Workload balance, however, depends on the number of non-zero entries each thread processes, and thus we divide the sparse matrix among the threads using an *nnz-partitioning* approach, with each thread given a contiguous blocks of rows containing an almost equal number of non-zeros.

Figure 1 shows the performance speedup of our parallel CSR SpMV for different number of threads in comparison to serial WebAssembly CSR SpMV [29] using all our benchmark matrices. The x-axis shows the working set size of matrices when stored in CSR format (detailed definition in Section 4). We can clearly observe that SpMV parallel performance scales well with the increase in number of threads, except for the small matrices, which are likely affected by the parallelism overhead. Even for larger matrices, however, speedup has significant variance, mostly around 3x–5x with 5 threads (with a few super-linear outliers). We attribute reduced performance to other kinds of imbalance between the threads: locality index [29] imbalance, branch mispredictions imbalance and loop overhead imbalance etc. In the next section we describe several optimization strategies to lessen the effect of these imbalances, and analyze their effect on the SpMV performance.

4 SpMV Optimizations and Sparse Formats

In this section we describe our set of optimizations for web-based parallel SpMV for four different sparse storage formats (DIA, ELL, COO, CSR) [27]. We examine the SpMV performance for a number of optimizations including SIMD, software gather/scatter vectorization, reordering of matrix rows and loop optimizations like loop blocking, inner loop unrolling and outer loop unroll-peel-jamming. Although many of these optimizations are known and proven beneficial for SpMV in the native context, their applicability with WebAssembly is not straightforward. For example, WebAssembly incorporates SIMD instructions,

```
(loop $inner_loop
  (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_loop)
)
```

Listing 1: SIMD portion of single-precision Parallel SpMV DIA implementation in WebAssembly

but it does so in an incomplete fashion, lacking hardware gather/scatter, which motivated our software-based gather/scatter approach. Alongside this, it is not known whether making these optimizations applicable with the available WebAssembly instructions will be as impactful as their native counterparts on the given architecture. We evaluate the performance speedup after the application of several optimizations over the baseline SpMV for each storage format. In these evaluations, we use the working set size, the value of which depends on the $N \times N$ size of the sparse matrix, its number of non-zeros (nnz), and storage format.

4.1 DIA Format

The diagonal format (DIA) only stores the diagonals that include non-zeros. The `data` array stores the diagonal values, and the `offset` array stores the distance of each diagonal from the main diagonal, with positive offsets representing the upper diagonals and vice versa.

Data Partitioning : It is quite straightforward to parallelize the SpMV for DIA format. Similar to CSR, we have implemented the static distribution of workload among the processing elements by row decomposition method, in which the `data` array in DIA format is partitioned into row blocks to avoid false sharing. We tested both *row-partitioning* and *nnz-partitioning* strategies for our benchmark matrices. In the *row-partitioning* strategy, the load imbalance can exist among the workers if the padded zeros around the diagonals (either positive or negative offsets) are significant in comparison to the ideal number of non-zeros per worker.

SIMDization : The SpMV DIA kernel consists of nested loops, where the outer loop iterates across all diagonals, and the inner loop iterates across the elements of each diagonal. The contiguous access to the values of `data` array, input vector `x` and output vector `y` and no data dependencies between the iterations provide a perfect opportunity to vectorize the SpMV DIA computation. So, we applied *SIMDization* using the new WebAssembly 128-bit vector instructions as illustrated in Listing 1. We observed magnificent performance improvements as shown in Figure 3 with *DIA Working Set* on the x-axis, which is ($ndiag\_elems + num\_diags + 2 * N$) * 4, where *ndiag_elems* is the number of elements in the diagonals and *num_diags* is the number of diagonals.

Loop Blocking : It is known that the inherent diagonal structure of DIA matrices allows access to the values of input vector `x` and output vector `y` contiguously, providing both spatial and temporal locality. But at the same time, the column-wise access to the `data` array requires both `x`-vector and

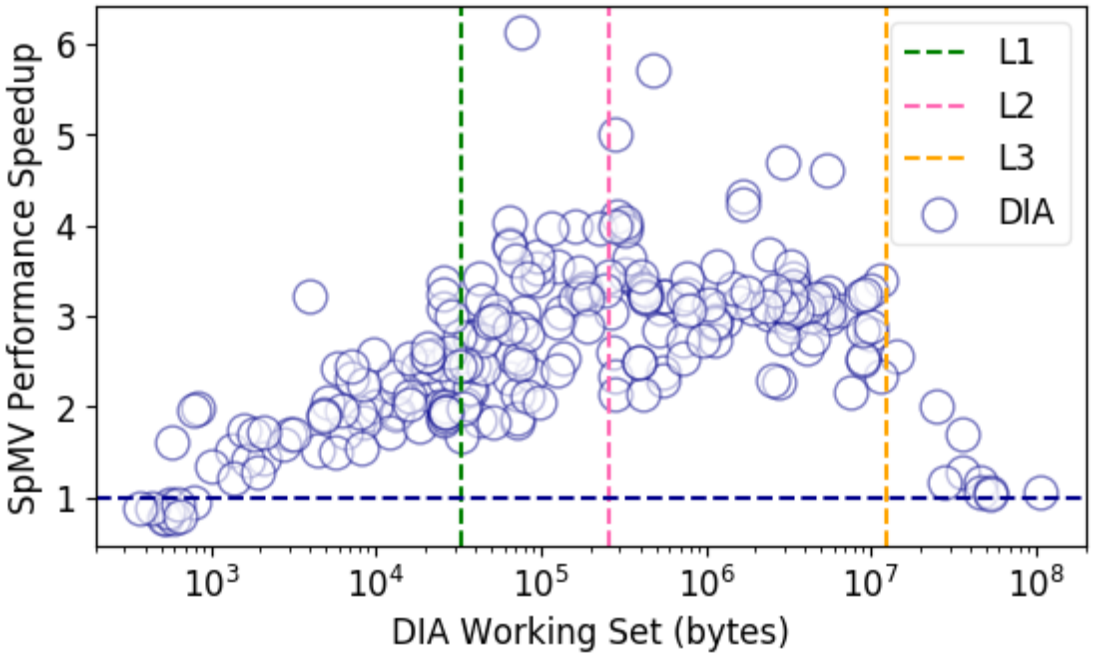


Figure 3: Performance speedup for DIA SpMV with SIMD

y-vector in the cache, repeatedly for up to *num_diags* for each worker. Therefore, for very large diagonal matrices, where both x-vector and y-vector won't fit in the L3 cache, it can cause numerous capacity cache misses. As a result, it can degrade the DIA SpMV performance, likely leading to not choosing DIA as the optimal storage format for the potential DIA matrix.

Thus, we have applied *loop blocking* optimization with block size $1K \times$ *num_diags*, especially targeted at such sparse matrices, and observed up to 2.6x performance gain on very large diagonal matrices as shown in Table 1. We chose to include all the diagonals in each block because the number of diagonals in these matrices are quite few in comparison to the number of rows. Also, to carry out this optimization, a small extra space is required to store the starting and ending row index of each diagonal for each worker. We note interesting performance differences between two matrices (*ecology1* and *ecology2*) which have almost equal N and *nnz*. This is apparently due to the difference in number of diagonals between those two matrices: *ecology2* has more diagonals than *ecology1*, leading to greater computation time.

Table 1: Parallel SpMV DIA performance before and after loop blocking optimization for very large diagonal matrices

Name	**N x N**	**nnz**	**Before (GFLOPS)**	**After (GFLOPS)**
ecology1	1M x 1M	4.9M	8.6	11.4
ecology2	1M x 1M	4.9M	6.5	9.0
atmosmodd	1.3M X 1.3M	8.8M	6.3	12.4
atmosmodl	1.5M X 1.5M	10.3M	5.3	11.9
Transport	1.6M x 1.6M	23.4M	5.1	13.3

```
(i32x4.splat(local.get $x))
(v128.load (i32.add (local.get $indices) (i32.shl (i32.add (local.get $exp) (
    local.get $row)) (i32.const 2))))
(i32.const 2)
(i32x4.shl)
(i32x4.add)
(local.set $x_index)
(f32x4.replace_lane 3
  (f32x4.replace_lane 2
    (f32x4.replace_lane 1
      (f32x4.replace_lane 0
        (f32x4.splat(f32.const 0.0))
        (f32.load (i32x4.extract_lane 0 (local.get $x_index)))
      )
      (f32.load (i32x4.extract_lane 1 (local.get $x_index)))
    )
    (f32.load (i32x4.extract_lane 2 (local.get $x_index)))
  )
  (f32.load (i32x4.extract_lane 3 (local.get $x_index)))
)
```

Listing 2: Software gather vectorization portion of single-precision SpMV ELL implementation in WebAssembly

4.2 ELL Format

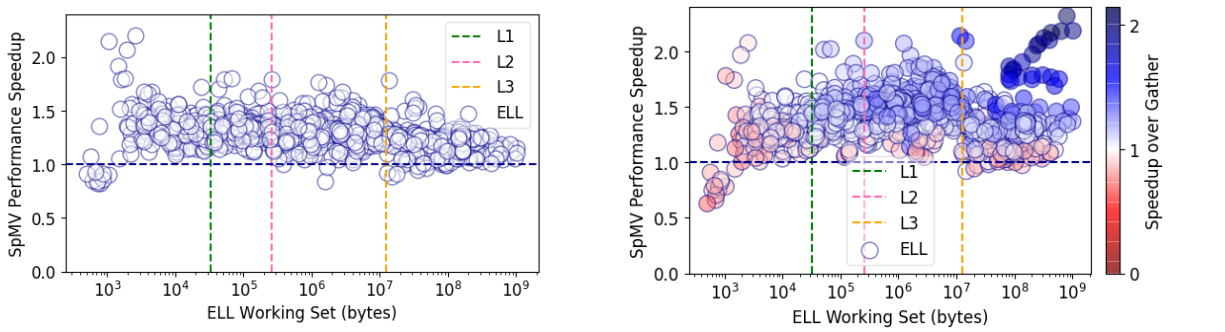
The ELLPACK format (ELL) stores a fixed (maximum) number of non-zeros per row, *max_nnz_row*. The `data` array stores the values for each row, with corresponding column indices given by the `indices` array.

Data Partitioning : Similar to DIA, the `data` array in ELL format is partitioned into row blocks, avoiding the `y`-vector conflicts. Since the `data` array in ELL is a packed 2D array, in which all the elements are accessed and processed (even for stored zeros) for SpMV computation, there is essentially no difference between *row-partitioning* and *nnz-partitioning* strategies.

Software Gather Vectorization : The SpMV ELL kernel consists of nested loops, where the outer loop iterates across all columns of `data` array, and the inner loop iterates across the elements of each column, leading to contiguous access to the values of `data` array (stored in column-wise fashion) and output vector `y`. It provides a perfect opportunity for these arrays to apply SIMD vector instructions similar to DIA, although the access to the values of input vector `x` (indirect addressing) depends on the structure of the matrix.

In order to improve the performance of such a computation via vectorization, we need hardware gather support (enabled in AVX2 for the first time) through which the non-contiguous `x`-vector values can be loaded into the SIMD registers. However, these instructions are not available for WebAssembly environment at the moment. Therefore, we have implemented *software gather vectorization* functionality using the new WebAssembly 128-bit vector instructions, to perform the indirectly indexed reads on input vector `x` as shown in Listing 2. After applying this optimization, we observed up to 1.6x performance gain for large sparse matrices as shown in Figure 4a, with *ELL Working Set* on the x-axis, which is $(2 * nell\_elems + 2 * N) * 4$, where *nell_elems* is $(max\_nnz\_row * N)$.

Loop Blocking : Similar to DIA, the column-wise access to the `data` array requires contiguously accessed `y`-vector and indirectly accessed `x`-vector in the cache, repeatedly up to *max_nnz_row* for each



(a) Software Gather Vectorization

(b) Software Gather Vectorization and Loop Blocking

Figure 4: Performance speedup for ELL SpMV with different optimizations

worker. Therefore, for matrices with very large dimensions, where `y`-vector and `x`-vector won't fit in the L3 cache, it can cause numerous capacity cache misses. We applied *loop blocking* optimization with block size $1K \times max\_nnz\_row$ along with the software gather optimization, and observed up to 2.3x combined performance gain on large sparse matrices as shown in Figure 4b.

4.3 COO Format

The coordinate format (COO) consists of three arrays, `row`, `col` and `val` to store the row and column indices and corresponding values of each non-zero.

Data Partitioning : We equally distributed the non-zeros among the workers to parallelize SpMV COO computation. In this there is never a chance of load imbalance, but there could be `y`-vector conflicts between the workers. Therefore, we allocated separate copies of `y`-vector for each worker, and accumulated the results in parallel at the end.

Software Gather and Scatter Vectorization : The SpMV COO kernel consists of a single loop that iterates through all the non-zeros, leading to the contiguous access of the values of `row`, `col` and `val` arrays. However, both input vector `x` and output vector `y` are indirectly addressed via these arrays. Therefore, to vectorize this SpMV computation and perform indirectly indexed reads and writes, we applied *software scatter vectorization* for the output vector `y` as shown in Listing 3, in addition to the previously described *software gather vectorization* on the input vector `x`.

We observed only up to 1.25x performance improvements on large sparse matrices as shown in Figure 5, with *COO Working Set* on the x-axis which is $(3 * nnz + 2 * N) * 4$. It is likely that vectorization performance benefit was discounted by the overhead of software gather/scatter instructions per non-zero.

4.4 CSR Format

In compressed sparse row (CSR) format, `row_ptr` array stores only one entry per row. This array keeps track of the starting position of each row for `col` and `val` arrays.

Software Gather Vectorization : The SpMV CSR kernel consists of nested loops, where the outer loop iterates across all matrix rows, and the inner loop iterates across the non-zeros of each row. Similar

```
(i32x4.splat(local.get $y))
(v128.load (local.get $coo_row))
(i32.const 2)
(i32x4.shl)
(i32x4.add)
(local.set $y_index)
(i32x4.extract_lane 0 (local.get $y_index))
(f32.load (i32x4.extract_lane 0 (local.get $y_index)))
(f32x4.extract_lane 0 (local.get $temp))
(f32.add)
(f32.store)
...
(i32x4.extract_lane 3 (local.get $y_index))
(f32.load (i32x4.extract_lane 3 (local.get $y_index)))
(f32x4.extract_lane 3 (local.get $temp))
(f32.add)
(f32.store)
```

Listing 3: Software scatter vectorization portion of single-precision SpMV COO implementation in WebAssembly

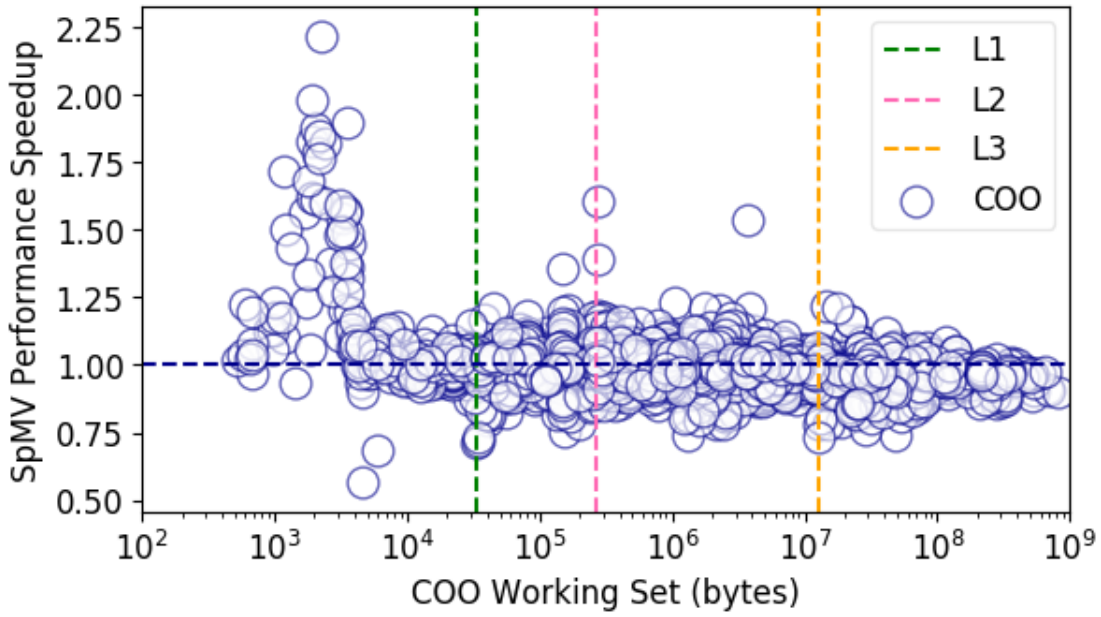


Figure 5: Performance speedup for COO SpMV with software gather/scatter vectorization

to ELL, we have applied *software gather vectorization* to perform the indirectly indexed reads on input vector `x`. Unlike ELL, we have performed sum reduction on the vectorized output for each row as shown in Listing 4, to finally store the result into the output vector `y`.

This vectorization can only be applied to the rows which have more than 4 number of non-zeros. In fact, the rows which have more than 4 number of non-zeros, the performance benefit is mostly observed when the inner loop runs for a number of iterations. Otherwise, the inner loop overhead and high number of software gather instructions per non-zero degrade the SpMV performance as shown in Figure 6a, with *avg_nnz_row*, average number of non-zeros per row on the colorbar, which is calculated as (nnz / N) and

```
f32x4.mul
(local.get $temp_v)
f32x4.add
(local.set $temp_v)
...
(local.get $y)
(local.get $temp)
(f32x4.extract_lane 0 (local.get $temp_v))
(f32.add)
(f32x4.extract_lane 1 (local.get $temp_v))
(f32.add)
(f32x4.extract_lane 2 (local.get $temp_v))
(f32.add)
(f32x4.extract_lane 3 (local.get $temp_v))
(f32.add)
(f32.store)
```

Listing 4: Reduction portion of single-precision SpMV CSR implementation in WebAssembly

CSR Working Set on the x-axis which is $((N + 1) + 2 * nnz + 2 * N) * 4$. We observed up to 2x performance speedup with this optimization for the matrices with high *avg_nnz_row*.

Reorder Matrix Rows : We have reordered the matrix rows based on the number of non-zeros per row, and observed up to 2x performance gain as shown in Figure 6b.

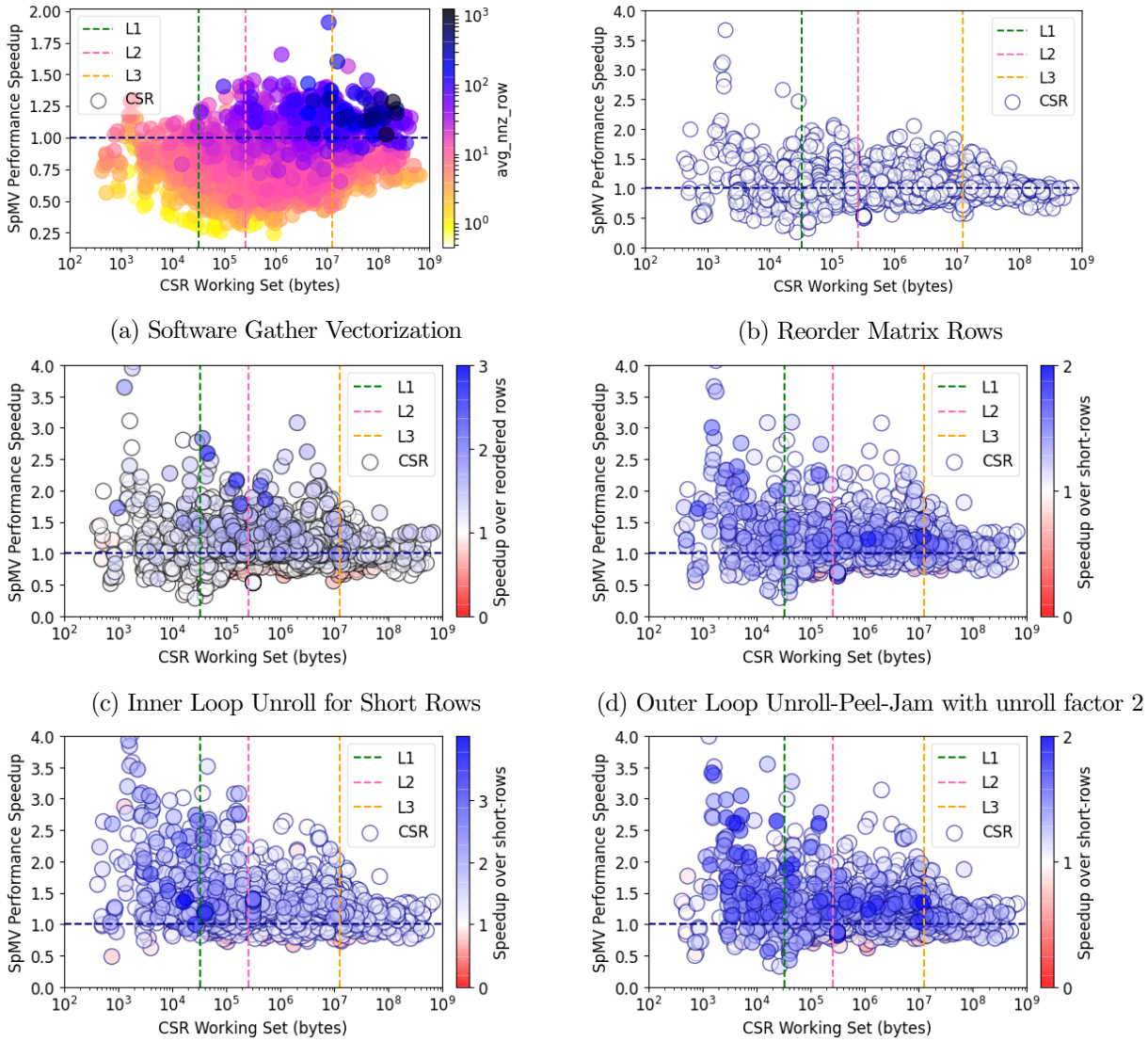
This strategy brings the rows with equal number of non-zeros together, and reduces the branch mispredictions for the matrices which otherwise have highly unequal number of non-zeros per row. In order to perform this optimization, some extra space is needed to store the permutation vector for the rows, which permutes the output vector `y` at the end of the SpMV computation.

Inner Loop Unroll for Short Rows : We have employed inner loop unrolling for the short rows which included the rows with number of non-zeros equal to 0, 1, 2 and 3, and observed up to 3x combined performance gain as shown in Figure 6c.

Following the application of reordering optimization, this strategy fully unrolls the inner loop for short rows, and processes the set of rows with equal number of non-zeros together with a single loop, while entirely bypassing the empty rows. In this way, inner loop overhead problems are avoided for the matrices which have high number of short rows.

Outer Loop Unroll-Peel-Jam : We generalized the popular "Unroll and Jam" [22] optimization for SpMV by combining it with loop peeling to observe the benefit of this customized optimization on a much wider set of matrices with different features. To effectively apply this optimization, we stacked it on top of the previous short rows strategy.

In this technique, we unroll the outer loop for different unroll factors (2, 3, 4 and 6), replicating the inner loop accordingly. Next, we peel these inner loops such that the number of iterations of these inner loops become equal. Finally, we jam these inner loops together to create one big inner loop. The peeled loop iterations are processed separately at the end. In this way, CSR SpMV computation is performed in column-wise fashion within the row blocks, with each block size equal to the unroll factor. This optimization is expected to improve the SpMV performance due to increased temporal locality for the input vector `x` and also due to reduction in the number of inner loop iterations. It especially seems beneficial for the matrices whose neighbouring rows have similar and wide `x-vector` access pattern. After applying this optimization using unroll factors 2, 3 and 4, we observed up to 3x combined performance gain as shown in Figures 6d, 6e and 6f.



(a) Software Gather Vectorization

(b) Reorder Matrix Rows

(c) Inner Loop Unroll for Short Rows

(d) Outer Loop Unroll-Peel-Jam with unroll factor 2

(e) Outer Loop Unroll-Peel-Jam with unroll factor 3

(f) Outer Loop Unroll-Peel-Jam with unroll factor 4

Figure 6: Performance speedup for CSR SpMV with different optimizations

5 Sparse Formats and Matrix Structure

Previous work has shown that matrix structure affects choice of optimal storage format [29]. Optimizations, however, change relative performance, and may thus modify the choice of format. In this section we consider why such changes occur, using Sandhu et al.'s 10%-affinity criteria in selecting the best format for each matrix [28].

Typically, the specialized sparse storage formats like DIA and ELL, if better suited for the given sparse matrix, tend to have higher SpMV computational capability than CSR or COO. Given the sparsity structure of the matrix, the availability of applicable structure-based optimizations for a particular format can further increase its SpMV computational capability. As a result, a sparse matrix shows affinity towards a particular format based on the values of its various structure features, whose threshold varies

with the relative degree of increase in SpMV computational capability of that format to other candidate formats in the presence of several optimizations.

In order to understand this better we first define *efficiency ratio (ER)* as the ratio of the amount of work done, w_d to the amount of work required to be done, w_{rd}. The value of w_{rd} for SpMV computation is the total number of non-zeros in the matrix, whereas the formula for w_d varies with the choice of storage format, and further depends on the sparsity structure of the matrix.

$$\begin{aligned} ER(f_i) &= \frac{w_d(f_i)}{w_{rd}} = \frac{w_d(f_i)}{nnz} \\ w_d(DIA) &= ndiag\_elems \\ w_d(ELL) &= max\_nnz\_row \times N \end{aligned} \tag{1}$$

Both DIA and ELL store less auxiliary information than CSR, and have the potential to be chosen as an optimal storage format for a large matrix if its *ER* value is 1 as shown in Figure 7 for DIA format.

On the other hand, an *ER* value greater than 1 means that the number of floating-point operations performed are higher than the required ones, if the matrix is stored in that format. This affects the SpMV performance and the optimal format choice. The application of SIMD vectorization and loop blocking optimization, for example, results in many of our benchmark matrices with *ER(DIA)* value up to 3 showing affinity towards the DIA format. For instance, a very large DIA matrix named *Transport* with *ER(DIA)* value equal to 1.006 (also shown in Table 1 of Section 4) shows affinity towards CSR format without the application of loop blocking optimization. Relatively small matrices also tend to suffer from inner loop overhead, and thus due to the distribution of workload for parallel and vectorized SpMV DIA computation, tend to choose other optimized formats over DIA.

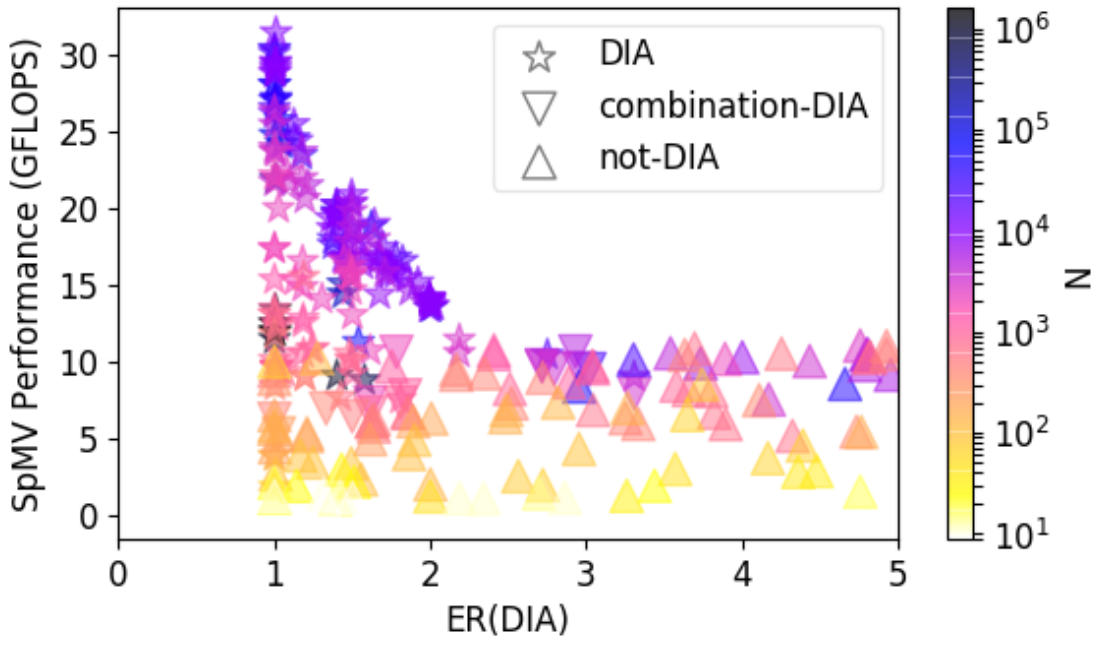


Figure 7: Effect of DIA efficiency ratio and matrix dimension on the choice of storage format using 10%-affinity

Next, we look at the `data` array width for DIA and ELL which is *num_diags* and *max_nnz_row* respec-

tively. This width is the count of repeated access of input vector `x` and output vector `y` at some regular interval for the SpMV computation. The high value of this structure feature for DIA or ELL may cause more capacity misses at lower-level caches, in comparison to CSR which accesses each value of the output vector `y` only once. For instance, a matrix named *piston* with $N = 2025$, $nnz = 100015$, *ER(DIA)* $= 1.78$ and *num_diags* $= 89$ shows affinity towards both DIA and CSR format, leading to choose *combination-DIA* category based on the 10%-affinity criteria. We found that the application of software gather and outer loop unrolling significantly improved its performance for CSR SpMV. Hence, it is a complex combination of these structure features and applicable optimizations which greatly affects the choice of storage format.

Finally, we observed only few small matrices from our benchmark set that show affinity towards the COO format. We argue that the application of our stack of optimizations has potentially improved the SpMV computational capability of CSR against COO. Otherwise, owing to the low values of *avg_nnz_row* (leads to inner loop overhead) and uneven number of non-zeros per row (leads to branch mispredictions), the matrices would usually show affinity towards COO over CSR.

6 SpMV Performance Comparison

In this section, we evaluate the SpMV performance of our benchmark matrices using our WebAssembly implementations relative to Intel MKL C and taco C++ libraries.

6.1 Intel MKL C

SpMV is one of the Sparse BLAS operations supported by Intel MKL C library. Before the computation, their inspector-executor two-stage algorithm first analyzes the matrix structure, and then performs optimizations including the conversion of CSR format into some undisclosed internal representation if needed. Figure 8 shows the performance speedup of our unoptimized WebAssembly CSR SpMV implementations over Intel MKL inspector-executor SpMV for all our benchmark matrices. Except for the small matrices, it is quite evident that there is a range of performance slowdowns of up to 4x for our WebAssembly SpMV. Next, we perform the format-wise comparison of our optimized SpMV with Intel MKL. For simplification, we have included the combination-format matrices into one of their corresponding format sets. The performance evaluation for COO format is not shown, on account of only few and small matrices in this category, which show high performance gain and are likely not optimized by Intel MKL.

6.1.1 DIA Format

Figure 9a shows that in our implementation the majority of DIA matrices have almost equal or better performance than Intel MKL inspector-executor. We attribute this mainly to SIMD optimizations and also the loop blocking optimization for very large matrices.

At the same time, we found performance slowdown for some of the DIA matrices. Hence, we evaluated a number of structure features that could impact the SpMV performance of these matrices. First of all, we recognized that the performance decreases with increase in the *ER(DIA)* value which is due to wasted computation cycles on stored zeros within the diagonals. Next, we found that the *num_diags* value also impacts the performance which is likely due to repeated access of the contiguous values of input vector x and output vector y. The combination of these features is especially impactful, and is found in the matrices for which our SpMV DIA performance is around 2x slower than Intel MKL inspector-executor SpMV. We

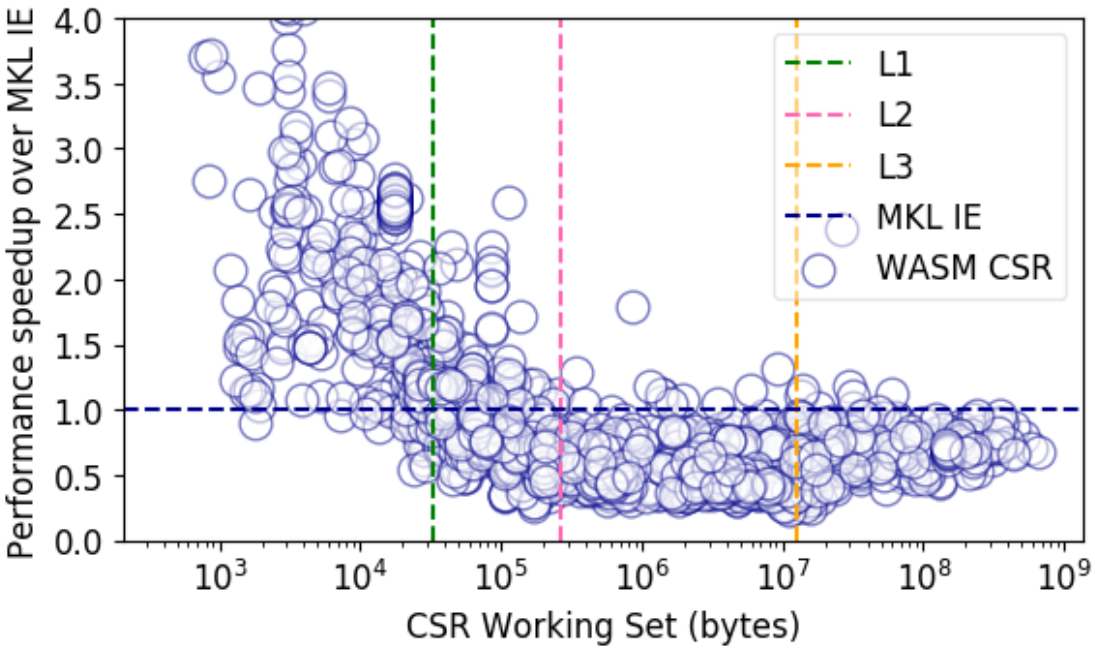


Figure 8: Unoptimized WebAssembly SpMV CSR performance speedup over MKL inspector-executor

expect that Intel MKL, if using DIA as their internal format for such matrices, has overcome these potential computation bottlenecks by using wider (256-bit) SIMD, which is currently not available for WebAssembly.

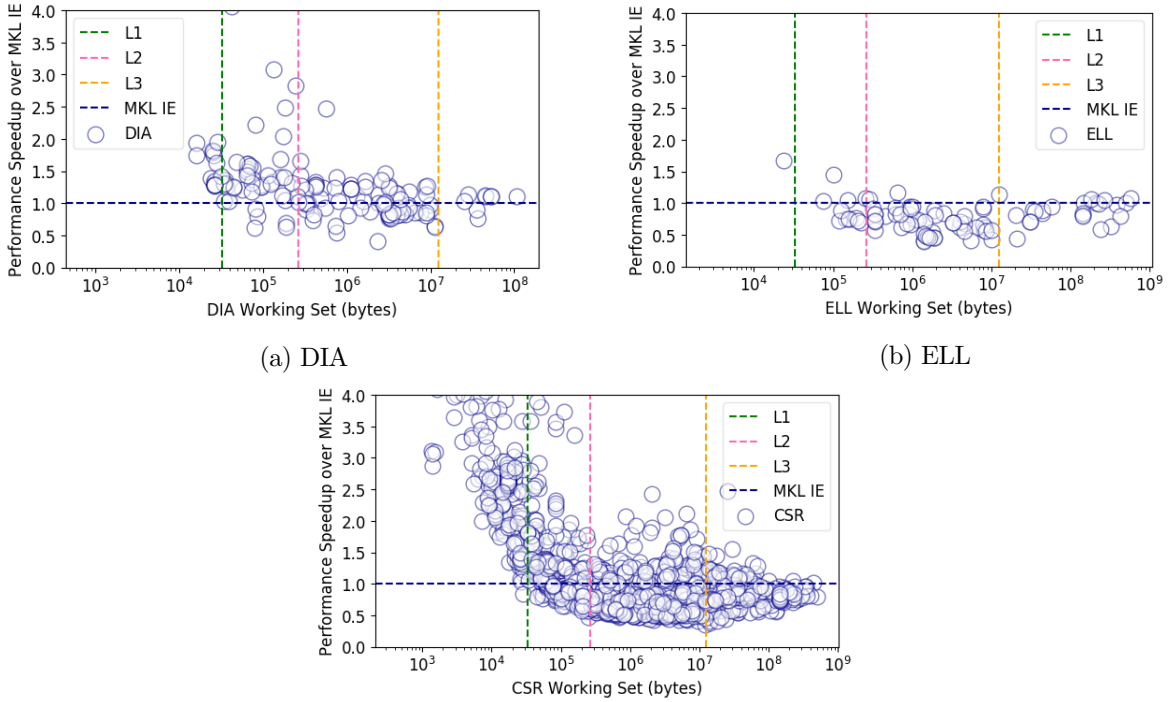
6.1.2 ELL Format

Figure 9b shows that using our implementations, the performance of some of the ELL matrices is similar to Intel MKL inspector-executor. We have seen that our software gather vectorization and loop blocking optimization have significantly improved the performance of ELL marices. However, we still observe performance slowdown relative to Intel MKL. Similar to DIA, we argue that the high *ER(ELL)* value impacts the performance by wasting computation cycles on the stored zeros for padding. Again, wider SIMD has the potential to further improve the SpMV performance.

We also recognize that despite our highly optimized implementations, several WebAssembly intricacies still affect the performance in comparison to the native implementations. For instance, the V8 compiler tends to favour use of registers in generated code instead of the memory addressing modes available in the x86-64 instruction set architecture. As a result, there are more load and store instructions and high register pressure. In addition, an extra operation is performed to calculate the effective address using a fixed offset for every array access from the base of WebAssembly's linear memory.

6.1.3 CSR Format

Figure 9c shows quite interesting performance comparison results of CSR matrices between our implementations and Intel MKL inspector-executor, leading to a geometric mean of 1.02. First of all, we observe very high performance gain for small CSR matrices, which are likely not considered for optimizations by Intel MKL due to their small size. Next, it can be seen that for a lot of large CSR matrices, we have up



(a) DIA

(b) ELL

(c) CSR

Figure 9: Performance speedup of optimized WebAssembly SpMV with different formats over MKL inspector-executor

to 2x performance gain over Intel MKL, which is due to our stack of various optimizations which targeted inner loop overhead for short row lengths, branch misprediction issues for uneven row lengths, and poor cache reuse due to highly scattered non-zeros in each row along with vector `x`'s indirect addressing.

On the other hand, we also find a range of performance slowdowns of up to 2x for a number of CSR matrices. It is important to note that this set mostly consists of the matrices for which we were able to improve the performance over our baseline CSR by identifying the potential performance bottlenecks. Wider SIMD and better instruction selection, however, has a significant impact on performance, especially for the CSR matrices with high number of non-zeros per row. Few of those matrices chose our baseline SpMV CSR as their best performance due to the absence of above mentioned shortcomings. Another potential optimization that may have also improved the SpMV performance for Intel MKL is software prefetching on the input vector `x`. This is used to overcome indirect addressing problems, but the current WebAssembly instruction set doesn't offer that functionality.

6.2 taco C++

The tensor algebra compiler (taco) C++ library [18, 9] uses compiler-based techniques to generate optimized sparse kernels like SpMV, and showed competitive performance with an earlier version of Intel MKL. Figure 10 shows the performance speedup of our optimized WebAssembly CSR SpMV implementations over taco-generated C++ CSR SpMV for all the matrices currently supported by taco (precisely 1333 matrices from our benchmark suite). It is quite impressive to observe up to 2x performance speedup over taco CSR for the majority of our large matrices. Some of the potential reasons for this are the application of gath-

er/scatter vectorization by taco for all the benchmark matrices including the ones with small number of non-zeros per row and high number of short-length rows, and unequal distribution of work among the threads.

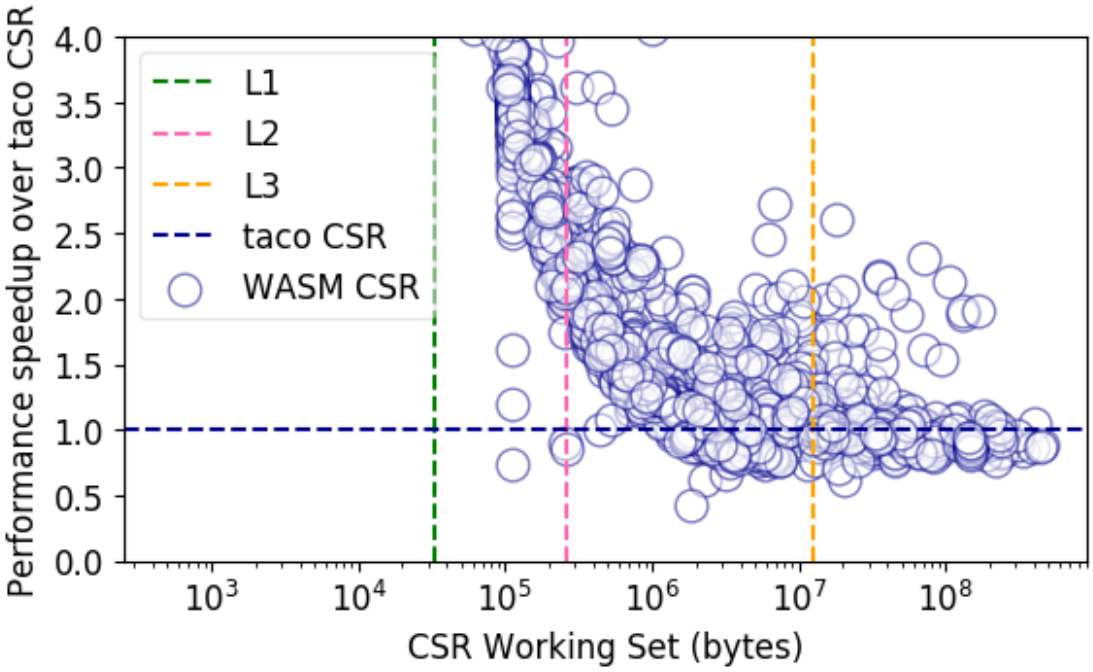


Figure 10: Optimized WebAssembly SpMV CSR performance speedup over taco C++ CSR

7 Conclusion and Future Work

We are witnessing a surge of scientific and compute-intensive applications involving SpMV on the web for its interactiveness and easy accessibility. It is of utmost importance to understand the effectiveness and design of the optimization techniques which bring out the best from these web-based systems. Our stack of well-designed and effective low-level optimizations, targeting several distinct SpMV performance bottlenecks, provides valuable insights about the factors that have a major impact on the WebAssembly SpMV performance. Our evaluations with scalable and hand-tuned parallel SpMV WebAssembly implementations, running on a modern web browser, demonstrate the potential to deliver competitive performance in comparison to the native and highly-tuned Intel MKL inspector-executor and taco-generated SpMV routines.

In our future work we wish to explore some reordering techniques based on the locality index [29] to improve the access pattern of input vector `x`. We recognize that wider SIMD and software prefetching are among the future optimization opportunities needed to be analyzed for web-based sparse computations. Along with these enhancements, we want to further explore more sparse storage formats and optimization strategies to finally build an efficient and user-friendly web-based sparse library, supporting several sparse matrix operations.

References

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

[2] Ar.js - augmented reality for the web, 2018. URL: https://github.com/jeromeetienne/ar.js.

[3] brain.js - neural networks in javascript, 2018. URL: https://github.com/BrainJS/brain.js.

[4] ml.js - a k-nearest neighbour classifier algorithm, 2018. URL: https://github.com/mljs/knn.

[5] Autocad web app, 2019. URL: https://web.autocad.com/.

[6] Photo editor — online photoshop lightroom, 2019. URL: https://lightroom.adobe.com.

[7] R. C. Agarwal, F. G. Gustavson, and M. Zubair. 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, pages 32–41, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press. URL: http://dl.acm.org/citation.cfm?id=147877.147901.

[8] Baoyuan Liu, Min Wang, H. Foroosh, M. Tappen, and M. Penksy. Sparse convolutional neural networks. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 806–814, 2015. doi:10.1109/CVPR.2015.7298681.

[9] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. Format abstraction for sparse tensor algebra compilers. *Proc. ACM Program. Lang.*, 2(OOPSLA):123:1–123:30, October 2018. URL: http://doi.acm.org/10.1145/3276493, doi:10.1145/3276493.

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

[11] A. Elafrou, G. Goumas, and N. Koziris. Performance analysis and optimization of sparse matrix-vector multiplication on intel xeon phi. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1389–1398, May 2017. doi:10.1109/IPDPSW.2017.134.

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

[13] D.B. Heras, J.C. Cabaleiro, and F.F. Rivera. Modeling data locality for the sparse matrix-vector product using distance measures. *Parallel Computing*, 27(7):897 – 912, 2001. URL: http://www.sciencedirect.com/science/article/pii/S0167819101000898, doi:https://doi.org/10.1016/S0167-8191(01)00089-8.

[14] David Herrera, Hanfeng Chen, Erick Lavoie, and Laurie Hendren. Numerical computing on the web: Benchmarking for the future. In *Proceedings of the 14th ACM SIGPLAN International Symposium on Dynamic Languages*, DLS 2018, pages 88–100, New York, NY, USA, 2018. ACM. URL: http://doi.acm.org/10.1145/3276945.3276968, doi:10.1145/3276945.3276968.

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

[16] Abhinav Jangda, Bobby Powers, Emery D. Berger, and Arjun Guha. 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, pages 107–120, Berkeley, CA, USA, 2019. USENIX Association. URL: http://dl.acm.org/citation.cfm?id=3358807.3358817.

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

[18] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proc. ACM Program. Lang.*, 1(OOPSLA):77:1–77:29, October 2017. URL: http://doi.acm.org/10.1145/3133901, doi:10.1145/3133901.

[19] Kornilios Kourtis, Vasileios Karakasis, Georgios Goumas, and Nectarios Koziris. 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, pages 247–256, New York, NY, USA, 2011. ACM. URL: http://doi.acm.org/10.1145/1941553.1941587, doi:10.1145/1941553.1941587.

[20] Jiajia Li, Guangming Tan, Mingyu Chen, and Ninghui Sun. Smat: An input adaptive auto-tuner for sparse matrix-vector multiplication. In *PLDI'13*, pages 117–126. ACM, 2013.

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

[22] John Mellor-Crummey and John Garvin. Optimizing sparse matrix-vector product computations using unroll and jam. *The International Journal of High Performance Computing Applications*, 18(2):225–236, 2004. doi:10.1177/1094342004038951.

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

[24] Nikhil Thorat, Daniel Smilkov, and Charles Nicholson. TensorFlow.js - A WebGL accelerated browser based JavaScript library for training and deploying ML models. URL: https://js.tensorflow.org.

[25] E. Nurvitadhi, A. Mishra, and D. Marr. A sparse matrix vector multiply accelerator for support vector machine. In *2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, pages 109–116, 2015. doi:10.1109/CASES.2015.7324551.

[26] Ali Pinar and Michael T. Heath. Improving performance of sparse matrix-vector multiplication. In *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*, SC '99, New York, NY, USA, 1999. ACM. URL: http://doi.acm.org/10.1145/331532.331562, doi:10.1145/331532.331562.

[27] Yousef Saad. Sparskit: a basic tool kit for sparse matrix computations. https://www-users.cs.umn.edu/~saad/software/SPARSKIT/, 1994.

[28] Prabhjot Sandhu, David Herrera, and Laurie Hendren. 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, pages 6:1–6:13, New York, NY, USA, 2018. ACM. URL: http://doi.acm.org/10.1145/3237009.3237020, doi:10.1145/3237009.3237020.

[29] Prabhjot Sandhu, Clark Verbrugge, and Laurie Hendren. A fully structure-driven performance analysis of sparse matrix-vector multiplication. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, ICPE '20, pages 108–119, New York, NY, USA, 2020. Association for Computing Machinery. URL: https://doi.org/10.1145/3358960.3379131, doi:10.1145/3358960.3379131.

[30] O. Temam and W. Jalby. Characterizing the behavior of sparse algorithms on caches. In *Proceedings of the 1992 ACM/IEEE Conference on Supercomputing*, Supercomputing '92, pages 578–587, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press. URL: http://dl.acm.org/citation.cfm?id=147877.148091.

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

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

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

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

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

[36] James B. White, III and P. Sadayappan. On improving the performance of sparse matrix-vector multiplication. In *Proceedings of the Fourth International Conference on High-Performance Computing*, HIPC '97, pages 66–, Washington, DC, USA, 1997. IEEE Computer Society. URL: http://dl.acm.org/citation.cfm?id=523991.938962.

[37] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, SC '07, pages 38:1–38:12, New York, NY, USA, 2007. ACM. URL: http://doi.acm.org/10.1145/1362622.1362674, doi:10.1145/1362622.1362674.

[38] Alon Zakai. Emscripten: an llvm-to-javascript compiler. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 301–312. ACM, 2011.