Performance Analysis of the Parallel Shared-Memory Sparse Matrix-Vector Multiplication on Unstructured Matrices

Kobe Bergmans, Karl Meerbergen, Raf Vandebril
Department of Computer Science, KU Leuven, Belgium
kobe.bergmans@kuleuven.be, karl.meerbergen@kuleuven.be, raf.vandebril@kuleuven.be

***Abstract*—This paper investigates sparse matrix-vector (SpMV) multiplication algorithm performance for unstructured sparse matrices. The development of an SpMV multiplication algorithm for this type of data is hard due to two factors. First, parallel load balancing issues arise because of the unpredictable nonzero structure. Second, SpMV multiplication algorithms are inevitably memory-bound because the sparsity causes a low arithmetic intensity. Three state-of-the-art algorithms for parallel SpMV multiplication on shared-memory systems are discussed. Three new hybrid algorithms are developed which combine optimization techniques of the current algorithms. A modern and high-performance implementation of all discussed algorithms is provided as open-source software. Using this implementation the algorithms are compared to Intel's oneMKL library. Furthermore, SpMV multiplication algorithms require the matrix to be stored in a specific storage format. Therefore, the conversion time between these storage formats is also analyzed. Both tests are performed for multiple unstructured sparse matrices on different machines: two multi-CPU and two single-CPU architectures. We show that one of the newly developed algorithms outperforms oneMKL by 40% on one of the multi-CPU architectures.**

***Index Terms*—Sparse matrix-vector multiplication, shared-memory parallelization, unstructured sparse matrix, high-performance computing, sparse linear algebra, graph analysis**

I. Introduction

The sparse matrix-vector (SpMV) multiplication is an important kernel in many widely used algorithms for scientific computing and graph analysis. For example iterative methods for sparse linear systems such as GMRES, BICGSTAB, and QMR [24]. Algorithms in the graph domain that rely on SpMV multiplication are breadth-first search and the famous PageRank algorithm [3]. Because the SpMV multiplication is at the core of these, and many more algorithms, an efficient parallel implementation is key to fast execution.

Two large sources of sparse matrices are discretizations of partial differential equations and graphs [8]. The discretizations are usually performed on a mesh and as a consequence these matrices usually exhibit structure. Therefore, most sparse matrices resulting from the discretization of a partial differential equation are structured matrices. Sparse problems with structured data can often be transformed into a problem that behaves similarly to a dense problem by using structure exploiting algorithms [25], [28]. A sparse matrix originating from a graph takes the form of an adjacency matrix. Because most graphs do not originate from 2D or 3D structures, the amount of edges per vertex usually varies a lot. For example, it has been observed that the amount of edges per vertex approximately follows a power law for some real-world graphs [1], [10], [21]. This large variation makes that the adjacency matrices of graphs are usually unstructured. In this work, we focus on the second type of sparse matrices because this type of dataset is more frequent than ever due to the trend toward large-scale data collection, and previous research is mostly focussed on structured sparse matrices.

Two main problems need to be taken into consideration when constructing an efficient parallel SpMV multiplication algorithm for unstructured sparse matrices. The first problem is caused by the sparse storage formats. These only store the nonzero elements, and this makes that each arithmetic operation of the SpMV multiplication requires multiple memory operations. This low arithmetic intensity is a problem on modern processors, because while processor speed has increased dramatically, the memory speed remains slow in comparison [23]. To overcome this low arithmetic intensity, it is important to use the cache of modern processors as much as possible. The second problem is parallel load balancing. Unstructured sparse matrices have an unpredictable nonzero structure, so finding an adequate load balance is hard. To this end, strategies need to be found that split up the workload as efficiently as possible. However, the strategies themselves cannot take up too much time during multiplication.

This text reviews, analyzes and improves upon the current state-of-the-art for SpMV multiplication on shared-memory processors for unstructured matrices. The main contributions are as follows:

- Three new hybrid SpMV multiplication algorithms, that combine different aspects of the current state-of-the-art are introduced.
- The performance of the current and new SpMV multiplication algorithms is thoroughly analyzed on different shared-memory architectures. These include two uniform memory access (UMA) and two non-uniform memory access (NUMA) systems [23]. The performance is compared to the Intel oneMKL SpMV multiplication, one of the most widely used libraries for sparse computations [14].
- Most high-performance algorithms require that the matrix

is stored in a specific storage format. The conversion to these storage formats is also a time-consuming step. Therefore, we also optimized and benchmarked the conversion time. To the best of our knowledge, we are the first to thoroughly investigate this preprocessing step.

II. Conventional sparse matrix storage formats

This section introduces four basic storage formats for sparse matrices. These basic storage formats and algorithms will form the basis for the state-of-the-art and hybrid SpMV multiplication algorithms.

The simplest way to store a sparse matrix is the triplet or coordinate (COO) format [5]. The matrix is stored using three arrays: `row_ind`, `col_ind`, and `data`, each with length `nnz`. The first two arrays store the row and column indices respectively and the last array stores the corresponding nonzero elements. The COO format leads to an intuitive SpMV multiplication algorithm that loops over the arrays and performs

$$y[\texttt{row_ind}[i]] \mathrel{+}= \texttt{data}[i] * x[\texttt{col_ind}[i]] \qquad (1)$$

in each iteration.

A more storage optimal scheme is Compressed Row Storage (CRS) [9]. This storage format requires the nonzero elements to be sorted in row-wise order. The `col_ind` and `data` arrays are the same as for the COO format. But, because of the specific ordering, the row indices can be compressed by only storing one element per row. This element points to the start of the row in the `col_ind` and `data` arrays. The array that stores these row pointers, `row_ptr`, will only have $m+1$ entries instead of `nnz`. Algorithm 1 describes the SpMV multiplication algorithm for matrices stored using the CRS format. It proceeds by looping over the `row_ptr` array and performing the multiplication row by row.

Algorithm 1 SpMV multiplication for the CRS format

1: **Input:** `col_ind`, `row_ptr` and `data` arrays corresponding to $A \in \mathbb{R}^{m \times n}$. Dense input vector x.
2: **Output:** Dense output vector y such that $y = Ax$.
3: $y = 0$
4: **for** $i = 0$; $i < m$; $i \mathrel{+}= 1$ **do**
5: **for** $k = \texttt{row_ptr}[i]$; $k < \texttt{row_ptr}[i+1]$; $k \mathrel{+}= 1$ **do**
6: $j = \texttt{col_ind}[k]$
7: $y[i] = y[i] + \texttt{data}[k] * x[j]$
8: **end for**
9: **end for**

The CRS format can be extended to the Incremental Compressed Row Storage (ICRS) format [17]. The `col_ind` array is replaced by a `col_inc` array which does not hold column indices but positive increments between the subsequent column indices. Furthermore, row changes are signaled by letting the column index surpass n. The column index of the first element in the next column is then the calculated value modulo n. Because of this, the `row_ptr` array can be replaced by a `row_jump` array which stores the increments between subsequent row indices. The most significant advantage of this format is that the increments allow pointer arithmetic to be used which is more efficient [17]. Finally, the ICRS format can be generalized into the Bidirectional ICRS (BICRS) format [26]. This format allows negative column and row increments. Overflow of the column index is still used to signal a row change. Because negative increments are possible, the nonzero elements can be ordered arbitrarily.

III. State-of-the-art

This section discusses the current state-of-the-art parallel algorithms for SpMV multiplication. The work of Yzelman and Roose [27] compares a large set of algorithms, and it shows that two methods have the best performance: Compressed Sparse Blocks (CSB) and Row-Distributed Block COH (BCOH). Later Merrill and Garland [19] found that the Merge-Based SpMV algorithm has equal performance to CSB.

A. Compressed Sparse Blocks

The Compressed Sparse Blocks [6] (CSB) algorithm introduces a new storage format. In this CSB format, the matrix is subdivided into sparse blocks. Hence, there are two storage schemes for the matrix. One on the block level and one inside the blocks. The nonzero elements in the sparse subblocks are stored using the COO format. Because the subblocks are smaller than the original matrix, the indices can be compressed. This compression is performed by packing the row and column index into one integer. These compressed indices are then stored in a single array where each subblock is stored contiguously. The nonzero entries are also stored in one contiguous array. On the block level, the matrix is stored as a dense matrix: pointers are stored to each block in row-wise order.

The motivation for sparse blocks is better cache reuse [25]. This is because for the multiplication of a single sparse block only a small part of the input and output vector is needed. Thus, there is a large probability that when an element of the input or output vector is used multiple times, it will remain cached during the block multiplication. So, for efficient SpMV multiplication, an appropriate subblock size needs to be used. Buluç et al. [6] consider two constraints when deciding on the optimal block size. First, storage cost is minimized by capping the compressed index size at 16 bits. This means that the block size β cannot be larger than 2^{16}. Second, for efficient cache utilization, the regions of x and y needed for multiplying the subblock should also fit comfortably in the L2 cache. This leads to an implementation that sets $\log_2(\beta) = 3 + \lceil \log_2(\sqrt{n}) \rceil$ and checks the constraints. If a constraint is not met, $\log_2(\beta)$ is lowered by 1 until all constraints are satisfied.

The efficient SpMV multiplication algorithm corresponding to this storage format uses tasking to achieve parallel execution. The multiplication of each block row corresponds to one task. This is chosen to avoid false sharing. But, if a block row contains too many nonzero elements the task is split and temporary vectors are used to avoid false sharing. Finally,

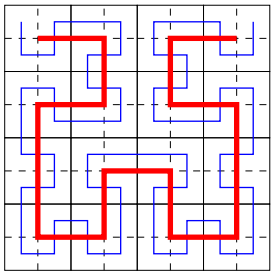


Fig. 1: Hilbert curve projected on a four-by-four and eight-by-eight matrix.

there is also the possibility to split up a dense block into multiple tasks. To perform this splitting efficiently, the nonzero elements in each subblock are stored in Z-Morton order [20]. This ordering first stores the elements of the top-left quadrant, then the top-right, bottom-left, and finally the bottom-right quadrant. This ordering is applied recursively in each quadrant. By using the Z-Morton order, dense blocks can be efficiently split using binary search. Furthermore, this ordering can also induce extra cache reuse inside blocks because the Z-Morton order provides additional locality.

B. Row-Distributed Block CO-H

The Row-Distributed Block CO-H [27] (BCOH) algorithm is also based on sparse subblocks. The nonzero elements in each subblock are stored using compressed ICRS. The indices are compressed separately and two arrays with compressed size are kept. On the sparse block level, the matrix is stored using the BICRS storage format. The index arrays used to store the block matrix are also compressed. Thus, two compressed sparse storage formats are used: compressed BICRS on the block level and compressed ICRS inside each block. For efficient compression, the block size selection method of CSB is also used for the BCOH storage format. However, the upper bound on β needs to be tweaked because the ICRS format used inside the blocks expects that overflow is possible. Thus, an upper bound of 2^{15} is set on the block size.

As for CSB, the motivation to use subblocks is better cache reuse. The authors believe that using a space-filling curve inside a subblock is unnecessary because the relevant part of the input and output vector will fit into the L2 cache [27]. To improve the locality nonetheless, they induce an ordering on the subblocks themselves, which is possible because BICRS is used on the block level. The ordering which is chosen follows the Hilbert curve [12], whose recursive definition is depicted in Figure 1.

The parallel SpMV multiplication algorithm for the BCOH storage format uses static load balancing. This is achieved by first dividing the rows of the matrix among the available threads such that each thread processes approximately the same number of nonzero elements. Then, each thread applies the blocking storage format described above on its part of the matrix. The SpMV multiplication proceeds by each thread performing the multiplication on its own part of the matrix.

C. Merge-Based

The Merge-Based parallel SpMV multiplication [19] is an algorithm that works directly on the CRS storage format. It uses an analogy to the parallel merging of a sorted list to try to obtain perfect load balancing. More specifically, the algorithm ensures that each thread performs the same amount of operations. These are multiply-add operations and storing a value in the output vector.

The workload is split up statically, where each thread processes a continuous chunk of nonzero elements. This is done to avoid false sharing as much as possible. Furthermore, each thread does not store its final value in the output vector to avoid write conflicts. The final values are stored in a separate array, and once all threads have finished, these values are added to the output vector sequentially.

IV. Hybrid algorithms

The algorithms discussed in the previous section use several techniques to obtain faster execution time for SpMV multiplication. While each algorithm uses its own combination of these techniques, it is also interesting to combine aspects of different algorithms because each state-of-the-art algorithm achieves high performance for SpMV multiplication. Therefore, three new *hybrid* algorithms will be explored in this section.

A. Compressed Sparse Blocks

A new algorithm was created by altering the CSB storage format. The ordering inside the blocks was changed from the Z-Morton curve to the Hilbert curve. The new SpMV multiplication algorithm will be called the Compressed Sparse Blocks Hilbert (CSBH) algorithm.

This *hybrid* algorithm is inspired by the observation that the locality provided by the Z-Morton ordering is suboptimal compared to the Hilbert curve. The Z-Morton curve induces large jumps across diagonals. This never happens for the Hilbert curve, where only one index changes at a time and the index value always increases or decreases by one. However, if the nonzero elements of a matrix are sorted using these curves, the nonzero structure also has to be considered. Therefore, both orderings will induce sudden jumps in row and column index. Nonetheless, we will show that the Hilbert curve provides better locality. The only algorithmic change needed to support the new storage format is the dense block splitting. A binary search method can still be used to find the start and end point of each subblock, but the different ordering needs to be taken into account.

B. Row-Distributed Block CO-H

Subsequent alterations to the BCOH algorithm lead to two new *hybrid* algorithms. The first alteration consists of changing the storage format inside the blocks. The ICRS storage format is changed to compressed COO storage. The compressed indices are, as for CSB, stored by packing them into a 32 bit integer. The ordering of nonzero elements remains row-wise. The described alteration to the BCOH algorithm is called the BCOH Compression (BCOHC) algorithm.

The original BCOH algorithm uses compressed ICRS inside each block. The reason for this is that the storage cost of a CRS-based format is lower than for the COO format [5]. But, this scheme also has its downsides. First, the ICRS storage format does not allow arbitrary nonzero ordering. Second, hardware simulations show that the average DRAM bandwidth usage of the BCOH algorithm is significantly lower than other SpMV multiplication algorithms [4]. Because most time is spent in the block multiplication step, the memory usage pattern can be attributed to the storage format used inside the blocks. Both of these concerns are addressed by changing to the block storage format to compressed COO. This makes an arbitrary nonzero ordering possible, and it does not suffer from suboptimal DRAM usage. The downside of changing from ICRS to the COO format is that the storage cost increases.

The second *hybrid* algorithm based on BCOH is the BCOH Compression Hilbert (BCOHCH) algorithm. This algorithm uses the storage format of BCOHC but sorts all nonzero elements per thread using the Hilbert curve. So, the rows of the sparse matrix are first divided amongst the threads, as is done for the original BCOH algorithm. Then, all nonzero elements per thread are sorted in Hilbert order. Finally, the elements are stored inside their respective block using the compressed COO format. Because of the recursive definition of the Hilbert curve, the sparse block matrix will still be sorted in Hilbert order per thread. The advantages of this change are twofold. First, the elements inside each block are ordered following the Hilbert curve. For most matrices, this will increase locality and lead to better cache reuse. Second, because for each thread all elements are sorted using the Hilbert curve, the transitions between blocks are also as local as possible. This can be seen by interpreting the blue curve of Figure 1 as a four-by-four block matrix with blocks of size two-by-two. We can then easily see that during the transition from one block to the next, the row or column index always increases by one.

V. Experimental design

This section describes the evaluation method of the six algorithms described above. Two metrics are benchmarked: the SpMV multiplication speed, and the time needed to convert from the COO format to the specific storage format each algorithm requires. To enable a comprehensive analysis of the runtime, we introduce a high-performance implementation, a set of test matrices, and the test machines.

A. Implementation

An efficient implementation of the algorithms described in Section III and Section IV is provided as open source software[1]. The software is written in `C++` and all tested methods are parallelized using *OpenMP* [22]. The code is compiled using the `icpx` compiler, which is part of the *Intel oneAPI* programming model [15]. Compilation is performed using the `-DNDEBUG`, `-O3`, and `-march=native` flags. Before execution, the *OpenMP* environment variable `OMP_PROC_BIND` was set to true to bind threads to cores.

[1]https://gitlab.kuleuven.be/numa/public/spmat_public/tree/v1.1

Some changes were made to have an efficient execution on NUMA architectures. On machines with multiple NUMA domains, pieces of data are always stored closest to one domain. This makes that the cores of that domain will have a significantly faster access time compared to the other cores. Because of this, the operating system will try to obtain an optimal data allocation by moving data between sockets during execution [11]. This hurts performance because these inter-socket data movements are slow [18]. However, even though intersocket data movement is inevitable, it can be minimized using interleaved allocations [16]. These are allocations where chunks of data are interleaved between the different memory banks. On NUMA architectures, interleaved allocation is always used for the input vector because it will be accessed from all CPUs. For the output vector and matrix storage, the interleaved allocation is only used if the parallelization is performed dynamically (CSB and CSBH).

A parallel implementation of the conversion between the COO format and the storage formats required by each of the SpMV multiplication algorithms is also provided. The conversion always consists of two steps. First, the nonzero elements are sorted in the specific ordering needed for the storage format. Second, the different arrays for the storage format are populated by copying, and potentially compressing the index information. The first step is usually the most costly because it has been proven that the best average complexity of a sorting algorithm is $\mathcal{O}(n \log n)$ [7]. For sorting the In-place Parallel Super Scalar Samplesort algorithm [2] is used. This parallel sorting algorithm performs the sorting in-place and also provides a high cache-efficiency.

B. Test setup

All algorithms are tested on fifteen unstructured square sparse matrices, presented in Table I. All matrices are obtained from the Florida sparse matrix collection [8], except for `HHH` and `LHH` which are randomly generated. The test matrices were selected to cover a wide range of matrix types. The selection was primarily based on matrix density and size. The density is calculated as the amount of nonzero entries to the total amount of entries. The test set is split in two classes: low density matrices with a density lower than 10^{-6}, and matrices with a density higher than this threshold.

The algorithms were tested on four different machines, listed in Table II. The Sapphire Rapids machine has two Intel Xeon Platinum 8468 CPUs. This machine is configured with 4 NUMA domains per CPU. Ice Lake NUMA consists of two Intel Xeon Platinum 8360Y CPUs. Each CPU is configured with one NUMA domain. Ice Lake UMA is the same machine, but only one CPU is used to avoid NUMA effects. Finally, the Cascade Lake machine has one Intel Core i9–10980XE CPU with hyperthreading enabled, and is configured with one NUMA domain.

The SpMV multiplication algorithms and the conversion step were tested on all matrices with different amounts of threads. Each SpMV multiplication test consisted of 550 timed executions, and the result is the minimum recorded time. These

TABLE I: Test matrices[2]. Max and variance are with respect to the amount of nonzero elements per row. The matrices are split in a class with low (top), and higher density (bottom).

Matrix	size	nnz	density	max	variance
kmer_P1a	139M	298M	1.53×10^{-8}	40	4.06×10^{-1}
europe_osm	51M	108M	4.17×10^{-8}	13	2.30×10^{-1}
LHH	50M	171M	6.84×10^{-8}	144	6.25×10^{1}
road_usa	24M	58M	1.01×10^{-7}	9	8.63×10^{-1}
hugebubbles	21M	64M	1.41×10^{-7}	3	6.60×10^{-4}
hugetrace	16M	48M	1.87×10^{-7}	3	6.00×10^{-4}
wb_edu	10M	57M	5.89×10^{-7}	4K	4.12×10^{2}
uk-2002	19M	298M	8.69×10^{-7}	2K	7.58×10^{2}
ljournal	5M	79M	2.77×10^{-6}	2K	1.37×10^{3}
LiveJournal	5M	69M	2.94×10^{-6}	20K	1.30×10^{3}
indochina	7M	194M	3.53×10^{-6}	7K	4.66×10^{4}
cage15	5M	99M	3.73×10^{-6}	47	3.29×10^{1}
HHH	3M	38M	3.93×10^{-6}	537	1.14×10^{3}
com-Orkut	3M	234M	2.48×10^{-5}	33K	2.40×10^{4}
kron	2M	182M	4.14×10^{-5}	214K	5.71×10^{5}

TABLE II: Test machines. Each memory stick is placed such that it is the only stick in the memory channel.

Name	CPU cores	RAM
Sapphire Rapids	2×48 @ 2.1 GHz	16×16 GB, 4800 MHz, DDR5
Ice Lake NUMA	2×36 @ 2.4 GHz	16×16 GB, 3200 MHz, DDR4
Ice Lake UMA	36 @ 2.4 GHz	16×16 GB, 3200 MHz, DDR4
Cascade Lake	18 @ 3 GHz	2×16 GB, 3200 MHz, DDR4

timings are compared to the Intel oneMKL [14] CRS SpMV multiplication implementation, and the speedup compared to this algorithm is calculated. The test of the conversion speed consisted of 25 timed executions. The results are presented as minimum conversion time divided by the fastest execution of the oneMKL SpMV multiplication.

VI. Results

We describe the results of the tests introduced in the previous section. First, we discuss the SpMV multiplication speed. Second, the conversion time between the COO format and the specific storage formats needed for the SpMV multiplication algorithms is presented.

A. SpMV multiplication

The results of the SpMV multiplication tests are summarized in Table III and Table IV. The first table displays the mean speedup compared to the oneMKL algorithm for the test matrices with a density smaller than 10^{-6}. The second table aggregates the results for the more dense test matrices.

In all results we see that the new hybrid algorithms always manage to outperform the current state-of-the-art algorithms. The changes made to these algorithms always have a positive or negligible effect. Ordering the nonzero elements inside the blocks using the Hilbert curve has a small, but positive effect. By far the best change is changing the storage format inside the blocks of BCOH, this leads to speedups of up to 30%.

[2]For conciseness the names of the following matrices are shortened: hugebubbles-00020, hugetrace-00020, ljournal-2008, soc-LiveJournal1, indochina-2004, kron_g500-logn21.

TABLE III: Average best speedup for low density test matrices. Highest speedup for each machine is indicated in bold. The number of threads for which the result is obtained is in parentheses. Newly developed methods are in bold.

Methods	SR	IL NUMA	IL UMA	CL
Merge	0.904 (96)	**0.947 (72)**	0.945 (36)	0.976 (12)
CSB	0.608 (96)	0.671 (72)	1.015 (36)	1.137 (12)
CSBH	0.627 (96)	0.689 (72)	**1.026 (36)**	**1.151 (12)**
BCOH	0.962 (96)	0.776 (72)	0.735 (36)	0.731 (6)
BCOHC	1.000 (96)	0.795 (72)	0.755 (36)	0.709 (6)
BCOHCH	**1.007 (96)**	0.785 (72)	0.719 (36)	0.705 (6)

TABLE IV: As Table III, for test matrices with higher density.

Methods	SR	IL NUMA	IL UMA	CL
Merge	1.186 (96)	1.081 (72)	0.998 (36)	**1.107 (14)**
CSB	0.535 (72)	0.561 (68)	0.874 (36)	1.056 (12)
CSBH	0.574 (88)	0.601 (72)	0.897 (36)	1.052 (12)
BCOH	1.048 (96)	0.962 (72)	0.861 (36)	1.057 (18)
BCOHC	1.371 (96)	1.194 (72)	1.033 (36)	0.975 (18)
BCOHCH	**1.407 (96)**	**1.223 (72)**	**1.060 (36)**	0.971 (18)

The importance of RAM speed is also immediately clear. The Cascade Lake machine has a significantly lower RAM speed than the other machines. This is noticeable in all results, where the low RAM speed hinders extra speedup when using additional threads. For matrices with high sparsity, each algorithm achieves peak performance when using twelve threads or fewer. This shows that memory is a more significant bottleneck than computing power for all algorithms. Furthermore, we see that hyperthreading has a negative effect on algorithm performance, as no algorithm achieves the best speedup when using more threads than there are physical cores.

For the matrices with low density of nonzero elements, the results show that for NUMA machines it is hard to outperform oneMKL. However, for the UMA machines it is possible. The CSB variants consistently outperform the oneMKL method, particularly on the Cascade Lake machine with low RAM speed, where CSBH achieves up to 15% better performance.

For higher-density matrices some tested algorithms do outperform the oneMKL algorithm significantly. Especially the new hybrid algorithms based on the BCOH algorithm show great performance on the NUMA machines. On the Sapphire Rapids machine BCOHCH outperforms oneMKL by 40%.

Figure 2 shows the performance each algorithm achieves per matrix on the Sapphire Rapids machine. This data clearly shows that as the density of matrices increases, the performance of the algorithms does too. This is because denser matrices lead to better cache reuse. We also observe a significant limitation of the CSB and CSBH algorithms: for some matrices these algorithms perform really poorly. This could be due to these matrices having many dense blocks which increases the parallel scheduling overhead. Furthermore, the BCOHC and BCOHCH algorithms consistently outperform the oneMKL baseline for all matrices with a density higher than 2×10^{-7}. This shows that on machines with high core counts and multiple NUMA domains, these algorithms should

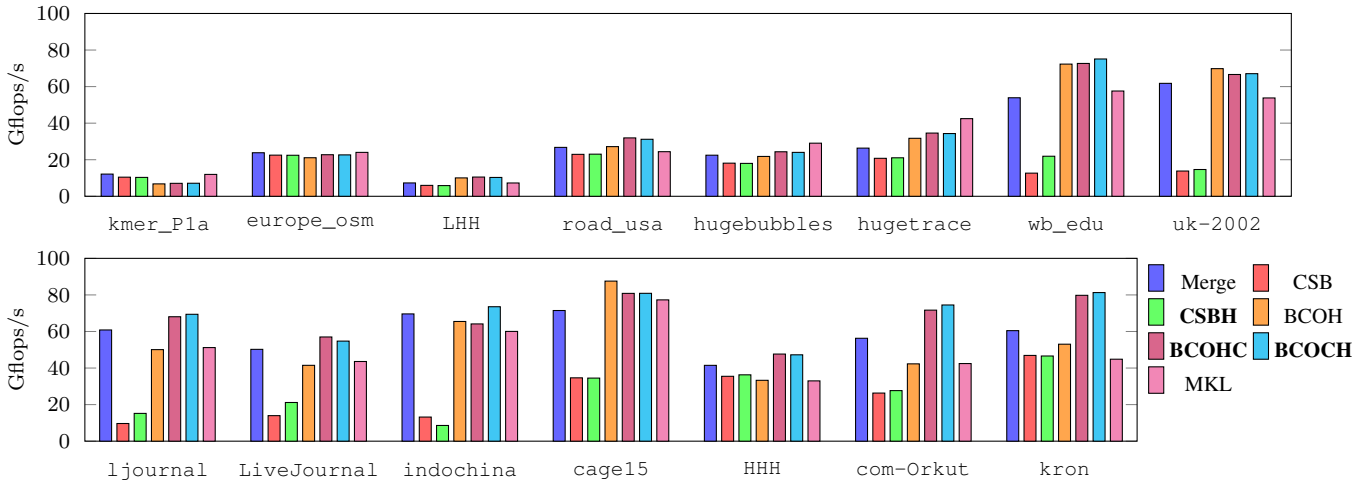


Fig. 2: Performance of the SpMV multiplication on the Sapphire Rapids machine. Matrices are sorted with increasing density.

TABLE V: Average conversion time for the low density test matrices. Conversion time is displayed as the amount of oneMKL SpMV multiplications. The number of threads that resulted in the fastest SpMV multiplication is used. Newly developed methods are in bold.

Methods	SR	IL NUMA	IL UMA	CL
Merge	91.8	32.4	20.3	7.6
CSB	83.7	34.9	27.7	10.9
CSBH	347.7	240.9	247.8	69.6
BCOH	194.7	143.5	134.9	74.5
BCOHC	186.2	133.6	124.2	69.0
BCOHCH	464.5	395.3	399.5	204.1

TABLE VI: As Table V, for test matrices with higher density.

Methods	SR	IL NUMA	IL UMA	CL
Merge	143.3	47.6	30.6	13.4
CSB	140.5	66.2	57.3	21.6
CSBH	643.2	523.2	526.1	150.0
BCOH	256.9	217.9	215.9	49.2
BCOHC	258.9	217.5	214.9	49.6
BCOHCH	667.7	659.0	688.8	131.1

be considered instead of oneMKL to achieve better performance when processing higher-density matrices. Finally, we can also see that the highest achieved performance is around $87\,\mathrm{Gflop/s}$. This is significantly lower than the system's peak performance of $5312\,\mathrm{Gflop/s}$, as measured by the oneMKL DGEMM benchmark [13]. This result is as expected, because the SpMV multiplication is severely memory bound.

B. Storage format conversion

Table V and Table VI display the time needed to convert a matrix, with respectively low and high density, to a specific storage format. The time needed is displayed as the number of SpMV multiplications of the oneMKL algorithm.

If we compare the tables in general, it is clear that the relative conversion cost is always higher for denser matrices. This can be explained by noting that the matrix density does not have a big influence on the conversion time. Therefore, the difference between the relative conversion time for really sparse matrices and more dense matrices is explained by the fact that the SpMV multiplication is faster for more dense matrices. Furthermore, we see that on the more powerful NUMA machines, the preprocessing induces far more overhead than on the UMA machines with fewer cores.

These results highlight the trade-offs of more complex algorithms, which require more expensive storage formats. For the current state-of-the-art we see that the CSB storage formats requires comparable preprocessing time to the CRS storage format, which is the standard. The BCOH storage format, however, requires up to four times as much preprocessing time. Conversion to the hybrid BCOHC storage format shows that changing the storage format inside the blocks does not have a significant effect. But, changing the ordering of nonzero elements can increase preprocessing time significantly. For BCOHCH changing to the Hilbert ordering inside the blocks increases the overhead by up to three times, and CSBH can require up to nine times more preprocessing time than CSB.

VII. Conclusion

The thorough analysis of multiple high-performance algorithms shows that choosing an algorithm for SpMV multiplication is not straightforward. Different machine and matrix properties call for different algorithms to be used. Furthermore, the number of times a specific matrix will be used in an SpMV multiplication is also of interest because storage format conversion is expensive. If many multiplications are required, we recommend the BCOHC or BCOHCH algorithms on NUMA machines for higher-density matrices, where performance can be up to 40% better than oneMKL. On UMA machines, for matrices with low nonzero density, both the CSB algorithm and its Hilbert-sorted variant clearly outperform oneMKL. However, CSBH should only be used when many multiplications are needed because of conversion overhead.

Acknowledgments

The research of Raf Vandebril was partially supported by the Research Council KU Leuven (Belgium), project C16/21/002, and by the Fund for Scientific Research — Flanders (Belgium), projects G0A9923N and G0B0123N. The work of Karl Meerbergen is supported by the KU Leuven Research Council (Belgium), project C24E/19, and by the Fund for Scientific Research — Flanders (Belgium), projects G088622N and G004124N. Some resources and services used in this work were provided by the VSC (Flemish Supercomputer Center), funded by the Research Foundation — Flanders (Belgium) and the Flemish Government.

References

[1] E. Almaas and A.-L. Barabási, *Power Laws in Biological Networks*, Springer US, Boston, MA, 2006, pp. 1–11.

[2] M. Axtmann, S. Witt, D. Ferizovic, and P. Sanders, *Engineering in-place (shared-memory) sorting algorithms*, ACM Trans. Parallel Comput., 9 (2022).

[3] A. Azad, M. M. Aznaveh, S. Beamer, M. Blanco, J. Chen, L. D'Alessandro, R. Dathathri, T. Davis, K. Deweese, J. Firoz, H. A. Gabb, G. Gill, B. Hegyi, S. Kolodziej, T. M. Low, A. Lumsdaine, T. Manlaibaatar, T. G. Mattson, S. McMillan, R. Peri, K. Pingali, U. Sridhar, G. Szarnyas, Y. Zhang, and Y. Zhang, *Evaluation of graph analytics frameworks using the gap benchmark suite*, in 2020 IEEE International Symposium on Workload Characterization (IISWC), 2020, pp. 216–227.

[4] K. Bergmans, K. Meerbergen, and R. Vandebril, *Performance of the parallel sparse matrix-vector multiplication for unstructured matrices*, master's thesis, KU Leuven. Faculteit Ingenieurswetenschappen, Leuven, 2023.

[5] A. Buluc, J. Gilbert, and V. B. Shah, *Implementing sparse matrices for graph algorithms*, in Graph Algorithms in the Language of Linear Algebra, J. Kepner and J. Gilbert, eds., SIAM, 2011, pp. 287–313.

[6] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, *Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks*, in Proceedings of the Twenty-First Annual Symposium on Parallelism in Algorithms and Architectures, SPAA '09, New York, USA, 2009, ACM, pp. 233–244.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, Cambridge MA: The MIT Press, thrid ed., 2009, ch. 8, p. 167.

[8] T. A. Davis and Y. Hu, *The university of Florida sparse matrix collection*, ACM Trans. Math. Software, 38 (2011).

[9] J. Dongarra, *Sparse matrix storage formats (section 10.1)*, in Templates for the Solution of Algebraic Eigenvalue Problems, Z. Bai, J. Demmel, J. Dongarra, A. Ruhe, and H. van der Vorst, eds., SIAM, 2000, pp. 315–319.

[10] M. Faloutsos, P. Faloutsos, and C. Faloutsos, *On power-law relationships of the internet topology*, in Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, SIGCOMM '99, New York, NY, USA, 1999, ACM, p. 251–262.

[11] G. Gurbinder, *Measure the impact of NUMA migrations on performance*, https://www.intel.com/content/www/us/en/developer/articles/technical/measuring-impact-of-numa-migrations-on-performance.html (accessed 2025-06-16).

[12] D. Hilbert, *Ueber die stetige Abbildung einer Line auf ein Flächenstück*, Mathematische Annalen, 38 (1891), pp. 459–460.

[13] Intel, *Benchmarking GEMM on Intel Architecture Processors*, https://www.intel.com/content/www/us/en/developer/articles/technical/benchmarking-gemm-with-intel-mkl-and-blis-on-intel-processors.html (accessed 2025-07-03). Version from 07-05-2022.

[14] Intel, *Intel® oneAPI Math Kernel Library (oneMKL)*, https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html (accessed 2025-06-27). Version 2023.1.0.

[15] Intel, *oneAPI*, https://www.intel.com/content/www/us/en/developer/tools/oneapi/overview.html (accessed 2025-06-18). Version 2021.4.0.

[16] A. Kleen, *libnuma*, https://github.com/numactl/numactl/tree/master (accessed 2025-06-26).

[17] J. Koster, *Parallel templates for numerical linear algebra, a high-performance computation library*, master's thesis, Department of Mathematics, Utrecht University, 09 2002.

[18] Y. Li, I. Pandis, R. Müller, V. Raman, and G. M. Lohman, *NUMA-aware algorithms: the case of data shuffling*, in CIDR 2013, Sixth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings, January 2013, https://www.microsoft.com/en-us/research/publication/numa-aware-algorithms-the-case-of-data-shuffling/.

[19] D. Merrill and M. Garland, *Merge-based parallel sparse matrix-vector multiplication*, in SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2016, pp. 678–689.

[20] G. M. Morton, *A computer oriented geodetic data base and a new technique in file sequencing*, tech. report, IBM Ltd., Ottawa, Canada, March 1966.

[21] L. Muchnik, S. Pei, L. C. Parra, S. D. S. Reis, J. S. Andrade Jr, S. Havlin, and H. A. Makse, *Origins of power-law degree distribution in the heterogeneity of human activity in social networks*, Scientific Reports, 3 (2013), p. 1783.

[22] OpenMP Architecture Review Board, *OpenMP*, https://www.openmp.org/ (accessed 2025-06-24). Version 5.2.

[23] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st ed., 2017, p. 510.

[24] Y. Saad, *Iterative Methods for Sparse Linear Systems*, SIAM, second ed., 2003.

[25] A. Yzelman and R. H. Bisseling, *Two-dimensional cache-oblivious sparse matrix-vector multiplication*, Parallel Computing, 37 (2011), pp. 806–819. 6th International Workshop on Parallel Matrix Algorithms and Applications (PMAA'10).

[26] A.-J. N. Yzelman and R. H. Bisseling, *A cache-oblivious sparse matrix–vector multiplication scheme based on the Hilbert curve*, in Progress in Industrial Mathematics at ECMI 2010, M. Günther, A. Bartel, M. Brunk, S. Schöps, and M. Striebel, eds., Berlin, Heidelberg, 2012, Springer Berlin Heidelberg, pp. 627–633.

[27] A.-J. N. Yzelman and D. Roose, *High-level strategies for parallel shared-memory sparse matrix-vector multiplication*, IEEE Trans. Parallel Distrib. Syst., 25 (2014), pp. 116–125.

[28] A. N. Yzelman and R. H. Bisseling, *Cache-oblivious sparse matrix–vector multiplication by using sparse matrix partitioning methods*, SIAM J. Sci. Comput., 31 (2009), pp. 3128–3154.