Optimizing Sparse Matrix-Vector Multiplication Using Index and Value Compression [*]

Kornilios Kourtis
kkourt@cslab.ece.ntua.gr

Georgios Goumas
goumas@cslab.ece.ntua.gr

Nectarios Koziris
nkoziris@cslab.ece.ntua.gr

National Technical University of Athens
School of Electrical and Computer Engineering
Computing Systems Laboratory
Zografou Campus, Zografou 15780, Greece

ABSTRACT

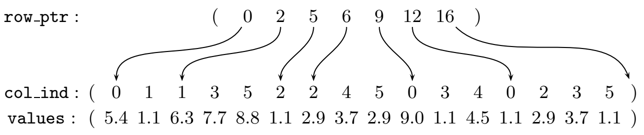
Previous research work has identified memory bandwidth as the main bottleneck of the ubiquitous Sparse Matrix-Vector Multiplication kernel. To attack this problem, we aim at reducing the overall data volume of the algorithm. Typical sparse matrix representation schemes store only the non-zero elements of the matrix and employ additional indexing information to properly iterate over these elements. In this paper we propose two distinct compression methods targeting index and numerical values respectively. We perform a set of experiments on a large real-world matrix set and demonstrate that the index compression method can be applied successfully to a wide range of matrices. Moreover, the value compression method is able to achieve impressive speedups in a more limited yet important class of sparse matrices that contain a small number of distinct values.

1. INTRODUCTION

Large sparse matrices are encountered in a wide range of scientific and engineering problems and most commonly in the numerical solutions of Partial Differential Equations (PDE), which frequently involve large sparse systems. Methods like Conjugate Gradient (CG) and Generalized Minimum Residual (GMRES) [17] that are employed to solve such problems use the Sparse Matrix-Vector multiplication (SpMxV) as their basic operation. Sparse matrices, by definition, are populated primarily with zeros and thus special representation schemes are used to enable efficient storage and computational operations. These representations usually store the non-zero values of the matrix with additional indexing information about the position of these values. In the rest of the paper we will make a distinction between data that are used for the representation of the matrix structure and data that represent the numerical values of the matrix elements. We will refer to the former as *index data* and to the latter as *value data*.

[*]This research is supported by the PENED 2003 Project (EPAN), co-funded by the European Social Fund (75%) and National Resources (25%).

$$A = \begin{pmatrix} 5.4 & 1.1 & 0 & 0 & 0 & 0 \\ 0 & 6.3 & 0 & 7.7 & 0 & 8.8 \\ 0 & 0 & 1.1 & 0 & 0 & 0 \\ 0 & 0 & 2.9 & 0 & 3.7 & 2.9 \\ 9.0 & 0 & 0 & 1.1 & 4.5 & 0 \\ 1.1 & 0 & 2.9 & 3.7 & 0 & 1.1 \end{pmatrix}$$



row_ptr : (0 2 5 6 9 12 16)

col_ind : (0 1 1 3 5 2 2 4 5 0 3 4 0 2 3 5)

values : (5.4 1.1 6.3 7.7 8.8 1.1 2.9 3.7 2.9 9.0 1.1 4.5 1.1 2.9 3.7 1.1)

Figure 1: Example of CSR Storage Format

One of the most commonly applied storage formats for sparse matrices is the Compressed Sparse Row (CSR) format [2, 17], which stores all the non-zero values in contiguous memory locations (`values` array) and uses two additional arrays for indexing information: `row_ptr` contains the start of each row within the non-zero elements array and `col_ind` contains the column number associated with each non-zero element. The size of the `values` and `col_ind` arrays are equal to the number of non-zero elements (`nnz`), while the size of the `row_ptr` array is equal to the number of rows (`nrows`) plus one. An example of the CSR format for a sparse 6×6 matrix is presented in Figure 1. The matrix-vector multiplication operation is easily implemented for matrices stored in CSR form (Figure 2).

```
for (i=0; i<N; i++)
  for (j=row_ptr[i]; j<row_ptr[i+1]; j++)
    y[i] += values[j]*x[col_ind[j]];
```

Figure 2: Sparse Matrix-Vector Multiplication for CSR

The working set (`ws`) of the SpMxV operation consists of the matrix data described above and the value data of the two vectors: `x` and `y`. Thus, the working set size for the CSR storage format can be computed by the following formula:

$$\begin{aligned} ws &= csr\_size + vectors\_size = \\ &\quad (nnz \times (idx\_s + val\_s) + (nrows + 1) \times idx\_s) \\ &\quad + (nrows + ncols) \times val\_s \end{aligned}$$

where *idx_size* and *val_size* is the memory size required for the storage of an index and a value respectively. Since for real-life sparse matrices it holds $nnz \gg nrows, ncols$[1], it is clear that the most dominant terms of the working set are the sizes of the `col_ind` and `values` arrays, which both have `nnz` elements. In most cases the vectors `x` and `y` have less than 2^{32} elements due to memory size restrictions and thus a 4-byte integer is used for index storage. Floating point values, on the other hand, require double precision most of the times, so the common value for *val_size* is 8 bytes. Under these conditions, the values constitute the larger portion of the working set by a factor of 2/3, if we consider only the `col_ind` and `values` arrays.

In our recent previous work [5], as well as in related literature [1], the memory subsystem and more specifically the memory bandwidth is identified as the main performance bottleneck of the SpMxV kernel. The above statement can be supported by the fact that SpMxV performs $O(\texttt{nnz})$ operations on $O(\texttt{nnz})$ amount of data, which means that most of the data are accessed in a streaming manner and there is little temporal locality. Additionally, it should be noted that this performance problem remains relevant with regard to modern and emerging microprocessors for two main reasons: The increasing gap in the performance between memory and CPU (memory wall problem) and the trend towards multicore design, where different processor units share a part of the memory hierarchy.

The goal of the work presented in this paper is to explore the design space for methods that reduce the working set and improve the performance of the SpMxV kernel by alleviating the pressure on the memory subsystem. We use the CSR storage format as our basis and consider two different approaches: reducing the index data and reducing the value data by compressing the `col_ind` and `values` arrays respectively. One challenge in this attempt is that the overhead imposed by the decompression mechanisms should not outweigh the benefits of the reduction of the working set. A secondary, yet important, requirement we set for these methods is performance predictability. To achieve the above goal, on one hand we propose a method that compresses the indexing structure of the sparse matrix taking into consideration the distribution of the non-zero elements in the columns of the matrix and exploiting the distances between elements rather than using absolute indexes. On the other hand, we introduce the idea of compression in the data structure that stores the values of the sparse matrix. To our knowledge there is no previous research work that aims at alleviating the memory pressure imposed by SpMxV, by compressing the floating-point values of the sparse matrix. Both approaches lead to the proposal of two new storage formats called CSR Delta Unit (*CSR-DU*) for index compression and CSR Value Indexed (*CSR-VI*) for value compression and to the corresponding multiplication algorithms that encapsulate the decompression strategies. Our experimental results show that there exist opportunities for accelerating the performance of SpMxV using compression in both cases. It should be noted that the objective of this work is not limited to the proposal of compression methods, but additionally aims to explore future directions, by understanding the issues at hand and investigating the various tradeoffs. The rest of the paper is organized as follows: Sections 2 and 3 present techniques for reducing the size of the index and value data respectively. Section 4 contains the results of a performance evaluation of the SpMxV kernel for the aforementioned methods. Section 5 discusses previous work and Section 6 discusses overall conclusions and directions for future work.

[1]In our 100 matrix set the `nnz` elements are on average larger than the number of `nrows` and `ncols` by a factor of 16.

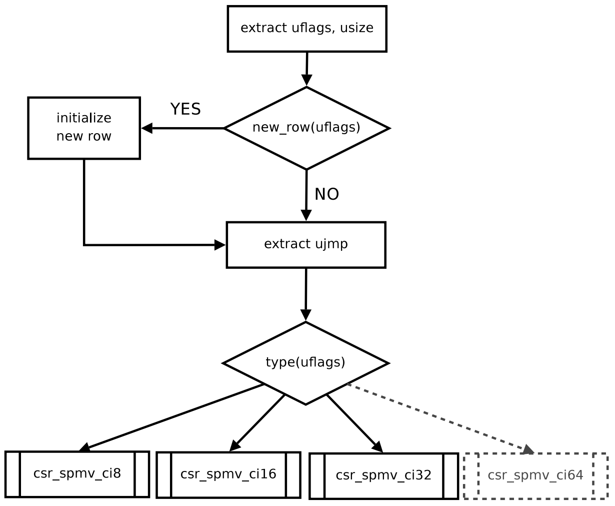
2. INDEX COMPRESSION

2.1 Motivation

As discussed above, the standard CSR format stores the column index of each non-zero element in the `col_ind` array using a 4-byte integer. A lot of research papers propose methods that result in the reduction of this array by exploiting contiguous non-zero elements within the matrix. For example the Block Compressed Sparse Row (BCSR) format [15, 8] reduces the index data by keeping only one index for each $k \times l$ dense subblock of the matrix. Our approach for the compression of index data is based on the general premise that there exist parts within sparse matrices exhibiting some level of density without necessarily containing contiguous non-zero elements. These parts can contribute significantly to the storage volume reduction of the sparse matrix indexing data. As presented in [23], delta encoding can be used to reveal the highly redundant nature of the `col_ind` array. Each delta, defined as the difference of the current index with the previous one, is positive and less or equal than its corresponding column index. Consequently, parts of a sparse matrix that exhibit density allow for their corresponding delta values to be stored in less than 4-byte integers, leading to the reduction of the index data size.

Instead of encoding each delta value to use only the necessary number of bytes, we propose a more coarse grain approach in which the matrix is divided into *units* with a variable number of elements. For each of these units the maximum delta value is calculated and a size that can represent this value is selected for all the delta values of the unit. This technique enables for innermost loops with minimum overheads by sacrificing some space. Normally, if each delta value was encoded separately, the innermost loop of the SpMxV kernel would contain branches that might lead to branch mispredictions on execution time and thus significantly degrade the performance of the kernel. It should be noted that an important factor for the performance of this approach lies in the choice of the unit size. If it is too small, the overhead introduced by the method will dominate the performance gain from the compression. On the contrary, if the unit size is large, there will statistically be less opportunities for compression, because a single large delta value will enforce big storage requirements for the whole unit.

Although this is a simple approach, it demonstrates a more abstract strategy for exploiting patterns in the structure of sparse matrices. Ideally, the index data, which constitute the structure of the matrix, could be mapped directly to optimized code for the efficient execution of the required operations in the matrix. The concept of *units* could be extended to support more types of regularities, thus providing a number of advantages towards this direction: (a) It can be used to exploit local regularities in specific areas of the matrix, (b) It operates on a coarse grain level and thus it can effectively minimize the introduced overhead by selecting sufficiently large sizes and (c) it can bound the search space for regularities or patterns and assure that the com-



```
flags = ctl_get_u8(ctl);
size = ctl_get_u8(ctl);
if ( flags_new_row(flags) ){
  y_indx++; x_indx=0;
}
x_indx += ctl_get_jmp(ctl);
switch ( flags_type(flags)  ){
  case CSR_DS_U8:
  for (;;) {
    y[y_indx] += (*values++) * x[x_indx];
    if (--size == 0)
      break;
    x_indx += ctl_get_u8(ctl);
  }
  break;

  case CSR_DS_U16:
  ...
}
```

Figure 3: Flowchart and code snippet for the SpMxV kernel for CSR-DU

pression procedure will not exceed the available resources (e.g. time or storage). Simple examples of types of regularities that can be exploited for performance improvements are matrix areas with sequential or diagonal elements.

2.2 Delta unit storage format

We will refer to our proposed format as CSR-DU for CSR Delta Unit. In CSR-DU the `col_ind` and `row_ptr` arrays of CSR are replaced by a single byte-array called `ctl`. This array contains all the necessary indexing information for each unit, which consists of four sections: `uflags`, `usize`, `ujmp` and `ucis`. Both `uflags` and `usize` have size of 1 byte and they identify the type of the unit and its size respectively. `ujmp` is a variable length integer that denotes the distance of this unit's column index from the previous one, while `ucis` is an array of `usize` $-\ 1$ elements, which contains the delta values of the column indices of this unit. In an analogy to network packets, one could say that `uflags` and `usize` constitute the header, while `ujmp` and `ucis` the payload. In the current version of CSR-DU there are four types of units supported, one for each integer type size: 8, 16, 32 and 64 bits. Additionally there is an indicator in the `uflags` variable that marks the beginning of a new row.

sections / unit	**uflags**	**usize**	**ujmp**	**ucis**
0	u8, NR	2	0	1
1	u8, NR	3	1	2,2
2	u8, NR	1	2	-
3	u8, NR	3	2	2,1
4	u8, NR	3	0	2,1
5	u8, NR	4	0	2,1,2

Table 1: Example of the information included in the `ctl` structure for the matrix presented in Fig. 1

An example of the information included in the `ctl` structure is given in Table 1, which shows the units that represent the indexing information of the sparse matrix of Figure 1. There are totally six units each of which has delta values that are stored in 1 byte (`u8`) and include a marker for the existence of a new row (`NR`). A flowchart and a simplified code snippet for the SpMxV operation for the CSR-DU format is presented in Figure 3. First the `uflags` and `usize` variables are extracted from the `ctl` array and if this unit belongs in a new row, the appropriate initializations are performed. Next, the `ujmp` distance is extracted and the proper multiplication is executed based on the type of the unit, which is encoded in `uflags`. Since, for the great majority of the matrices, there is no need for 64-bit column indices and thus the unit type for 64-bit sized deltas can be effectively omitted, it is marked with dashed lines in the flowchart.

In general the compression of a matrix can be performed either "off-line" or at run-time. In the first scenario the cost of the compression procedure is negligible because the compression needs to be done only once and a specialized disk storage format can be used to allow for fast loading of the matrix in memory. However, if the compression needs to be performed at run-time the cost should not be high enough to outweigh the benefits. The decision of whether a compression scheme can be applied at run-time depends on the application (e.g. the number of the SpMxV iterations that need to be performed). Nevertheless, we argue that CSR-DU is suitable for run-time deployment since the compression procedure has the same complexity with CSR parsing ($O(nnz)$) and can be performed in one pass with no backtracking using only local information for each unit.

2.3 Implementation Details

In this section we discuss some implementation details for the CSR-DU format. The information about the type of each unit is encoded in `uflags` in the form of bitflags. The basic bits used are presented in Table 2. It should be noted that `uflags` size has been deliberately chosen to be larger than strictly needed for this method to be able to support various future extensions and enhancements. The `usize` field encodes the size of this unit and thus the maximum size that can be represented is 256 elements. The `ujmp` field is a variable byte length integer that is always positive and encodes the distance between the column index of the first element of the unit and the column index of the previous element. In this way the delta values of two dense areas with large distance between them can be encoded efficiently. For the implementation of these variable length positive integers we used a simple scheme where the integer's bitstring is divided in parts of 7 bits. These parts are stored in consecutive bytes, in which the MSB is used to mark the last byte of the integer.

Bits	Description
0-1	These bits encode the size of the delta column index values: 00 → 1 *byte*, 01 → 2 *bytes*, 10 → 4 *bytes*, 11 → 8 *bytes*
2	new row bit, which designates that the current unit is on a new row

Table 2: Basic bits and their corresponding description for `uflags`

The compression of the matrices is a straightforward procedure: the elements are scanned one by one and put into buffers along with their indexing information until it is decided that a unit ends by an external algorithm. When this happens, the `ctl` and `values` arrays are filled with the appropriate values. A minor issue that arose from this strategy was that since the size of the `ctl` array is not known beforehand, we needed to implement a dynamic growing array structure. Moreover, we used a simple algorithm for determining the end of the units, in which a unit is finalized only when its size is full, or a new row is detected. This algorithm can be easily extended to support more elaborate schemes. One such scheme, for example, would dictate to finalize a unit before the delta value of a new element increases the delta storage size, if the unit has more than a predefined number of elements.

Finally, we applied a number of optimizations to the multiplication kernel. First we had to explicitly direct the compiler to write the $y[i]$ value into memory only at the end of each row processing and not in every iteration. Additionally, since the access of unaligned variables may cause performance problems, we aligned the `ucis` sections in the `ctl` array to their corresponding size by applying padding, so that the accesses of delta indices are performed in an aligned manner.

3. VALUE COMPRESSION

3.1 Motivation

Although a lot of discussion has been made concerning the distribution of the non-zero elements within the sparse matrix (e.g. symmetric, non-symmetric, structured, unstructured [17] etc.), to our knowledge, no attention has been paid on the content of the matrix in terms of floating-point values. As it was pointed out previously, in the typical case, the values constitute the larger part of the working set of a CSR matrix, because they require 64-bit storage. Hence, there is much more to gain from the compression of the values than the indices in terms of working set reduction. Nevertheless, the compression of floating point values is not as straightforward as integers, because floating point arithmetic operations produce rounded results. However, we have observed that there is a significant number of matrices in our experimental set in which only a small portion of their total values are unique. This redundancy can be exploited by storing only the common values and pointers to them instead of the `nnz` values, which will lead to the reduction of the working set, if the total-to-unique values ratio is high. Consequently, a considerable reduction will lead to performance improvement, despite the overhead induced by the indirect access of each value.

3.2 Value indexed storage format

In this section we describe a simple scheme for compressing the size of the values of a sparse matrix, which includes a small number of unique values relative to the total non-zero elements (`nnz`). In our proposed format, called CSR-VI for CSR Value Indexed, the `values` array of CSR is replaced with two arrays: `vals_unique` and `val_ind`. The first contains the unique values of the matrix and the second the index of the value in the `vals_unique` array for each of the `nnz` matrix elements. An example of this value structure is presented in Figure 4, which contains the values of the matrix presented in Figure 1.

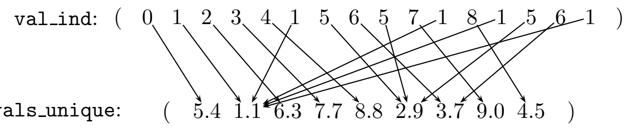

val_ind: (0 1 2 3 4 1 5 6 5 7 1 8 1 5 6 1)
vals_unique: (5.4 1.1 6.3 7.7 8.8 2.9 3.7 9.0 4.5)

Figure 4: Example of the value indexing structure for the CSR-VI format for the matrix presented in Figure 1

The SpMxV kernel implementation for CSR-VI is presented in Figure 5 and can be easily derived from the CSR case by replacing the direct access of `values` with an indirect access of `vals_unique` based on the value of `val_ind`. While the resulting code includes an additional memory reference for each of the `nnz` elements of the matrix, it will lead to fewer cacheline transfers from the main memory when the number of unique values is relatively small. It should be noted that the indirect access to the values array induces additional overhead in terms of execution instructions and thus a reduction of the working set will not necessarily lead to an improvement in performance.

```
for(i=0; i<N; i++)
  for(j=row_ptr[i]; j<row_ptr[i+1]; j++){
    val = vals_unique[val_ind[j]];
    y[i] += val*x[col_ind[j]];
  }
```

Figure 5: SpMxV kernel for the CSR-VI storage format

The practice of this method is to replace a large number of numeric values, with the same number of indices and a much smaller number of values. The working set reduction is achieved because only a small number of values needs to be addressed and thus the storage of an individual index is significantly less than the storage of an individual numerical value. On a second level, this may lead to more optimization opportunities by compressing these indices. Nevertheless, a scheme like delta encoding is not as suitable as in the compression of `col_ind`, because the access in the `vals_unique` is random and the delta values can be negative. Hence, we chose a more conservative approach, in which the size of the indices is determined by the size of the `vals_unique` array, which effectively is the maximum value of the `val_ind`'s elements. Although this approach is not optimal with regard to the compression of the value indices, it has minimum overhead, because it does not need additional branches, which can be the source of performance degradation. The compression procedure of CSR-VI has similar properties with that of CSR-DU: its complexity is $O(\texttt{nnz})$ and can be performed in a single pass and without backtracking if the maximum value of the `val_ind`'s element is known beforehand.

3.3 Implementation Details

The parsing for the creation of CSR-VI matrices is similar to the one used for CSR. The difference is that CSR-VI requires a method for detecting the common values, and finding their corresponding index in the `vals_unique` array. Our implementation is based on a hash table, in which the corresponding index is stored using the 64-bit floating point number as a key. So, if a numerical value exists in the hash table it means that this value has been encountered before and its index in the `vals_unique` is the corresponding value of the hash table. If a numeric value does not exist in the hash table, it is inserted in both the `vals_unique` array and the hash table. Its index and the hash table value is the number of elements of the `vals_unique` array prior to its insertion. Again, we have optimized the SpMxV implementation to write the resulting `y[i]` to memory only at the end of each row processing.

4. EXPERIMENTAL EVALUATION

4.1 Experimental Setup

Our experiments were conducted on an Intel Core 2 Xeon (Woodcrest) processor with 2.6 GHz clock speed, two 32 KB 8-way caches for instructions and data and a unified 4 MB 16-way L2 cache. The system was running a 64-bit version of linux and the compiler used was version 4.2 of gcc with the optimization flag -O3. All the results are expressed in terms of speedup with regard to the CSR SpMxV kernel with 32-bit indices and double precision (64-bit) values. The CSR version of the SpMxV kernel was also optimized to write the `y[i]` value at the end of each innermost loop, which led to an average performance improvement of 5% for the whole matrix set. The experiments were conducted by measuring the execution time of 128 consecutive SpMxV operations with randomly created `x` vertices. It should be noted that we made no attempt to artificially pollute the cache after each iteration, in order to better simulate iterative scientific application behavior, where the data of the matrices are present in the cache because either they have just been produced or they were recently accessed.

4.2 Matrix set

As a starting point for our experimental evaluation we use a set of 100 matrices (see Table 4). The majority of them was selected from Tim Davis' collection [4]. The first matrix is a dense 1000×1000 matrix, matrices 2-45 are also used in SPARSITY[6], matrix #46 is a 100000×100000 random sparse matrix with roughly 150 non-zero elements per row, matrix #87 is a 5-pt stencil finite-difference matrix for a $202 \times 202 \times 102$ regular grid created by SPARSKIT [16], while the rest are the largest rectangular matrices of the collection both in terms of non-zero elements and number of rows. Note that we made no attempt to select matrices with a particular structure of non-zero elements. The reader is referred to [4] for additional information on the specific characteristics of each matrix.

As shown in [5], two classes of matrices can be distinguished based on SpMxV performance. Matrices with a working set that fits into the L2 cache, experience only compulsory misses and exhibit a thoroughly different performance behavior compared to matrices whose working set is larger than the L2 cache size and may experience capacity misses. Since in this work we are mainly concerned with matrices that perform poorly due to memory bandwidth limitation, we are only considering matrices from the second class. More specifically, in order to also cover border-line cases (e.g. memory accesses due to conflict misses) we reject matrices whose working set is less than 3/4 of the L2 cache size, which in our case means $ws > 3\ MB$. Additionally, we also remove the dense matrix (#1) from our set to obtain more accurate average speedup results. The resulting set consists of the following 77 matrices: 2-13, 15, 17, 21, 25, 26, 36, 40-42, 44-53, 55-100 and we will refer to it as the *initial matrix set*.

4.3 CSR-DU

As noted in Section 2.1, small unit sizes are bound to lead to degraded performance in CSR-DU due to the loop overhead they induce. The class of matrices with a large percentage of small rows will suffer from this problem, since units cannot span multiple rows. In our experiments we used a simple qualitative criterion to filter those matrices out from our initial matrix set: we rejected matrices in which 85% percent or more of their elements are members of rows with size smaller or equal than 6 elements. The resulting rejected set consisted of 13 matrices which exhibited significant slowdown or similar performance to the CSR benchmark. The resulting speedup values for the remaining 64 matrices are presented in Figure 6. The number in the top of the bars is the percentage of the matrix size reduction relative to the original CSR size $((size_{CSR} - size_{CSR-DU})/size_{CSR})$.

The mean speedup value for our set is 8.1%. There exist 53 matrices which exhibit speedup and 5 that exhibit slowdown. Additionally, the larger speedup and slowdown are 18.9% for matrix #11 and 8.1% for matrix #5 respectively. Moreover, the dense matrix which constitutes the upper limit on the performance gain for this method, achieved a speedup of 1.35. An initial observation is that there does not seem to be a absolute correlation between the size reduction and the speedup, which leads to the conclusion that the size reduction is not always capable of assuring significant performance improvement, due to other factors affecting the overall performance (e.g. branch mispredictions).

Although the speedups of the CSR-DU method are relatively small, we argue that a number of factors make this approach relevant. First, it seems to generally satisfy the requirement for stability and performance predictability, since few matrices exhibit significant slowdowns in our considerably rich set. Additionally, as memory size increases with a very large rate, there will soon be matrices that can fit into memory and require more than 32-bit indices for the storage of their column indices. In that case the performance gain from index compression would be significantly higher, because the index data would double in size. In order to quantify the performance gain we repeated the experiments

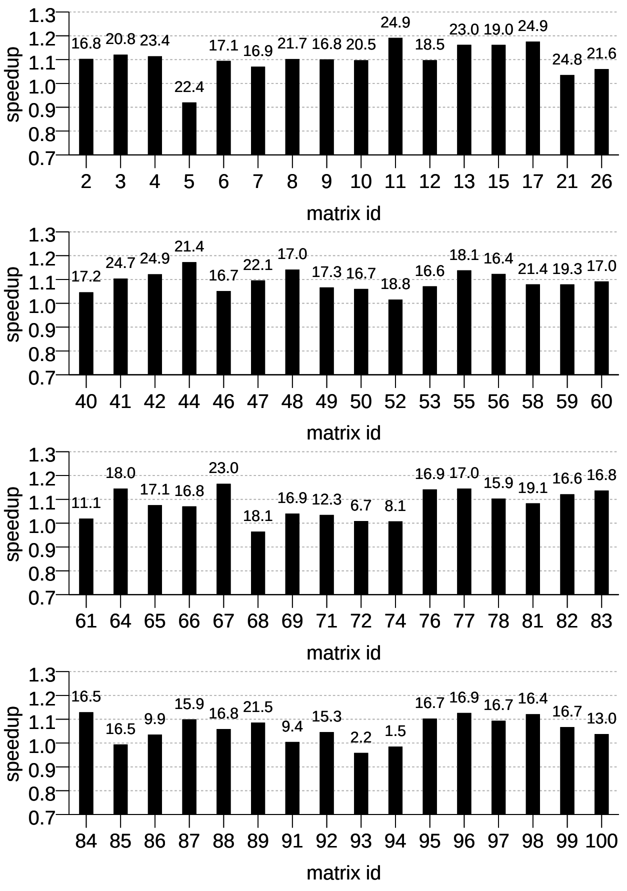


Figure 6: Speedups for the CSR-DU method

for our matrix set using 64-bit indices. As expected, the increase in the working set led to a performance drop in CSR by 36%, hence CSR-DU compared to the 64-bit indexed version of CSR achieves an average 44.1% performance gain. Finally, we argue that in the case of multi-core architectures, where a number of processing elements share a part of the memory subsystem, this approach would greatly benefit a multithreaded implementation of the SpMxV kernel, because of the impact of the memory contention imposed by multiple SpMxV threads.

4.4 CSR-VI

In order to elaborate on the applicability of the CSR-VI method to a given matrix, we consider the *total-to-unique* (*ttu*) values ratio, which is defined as the division of the total (nnz) values by the number of the values that are unique in the matrix. A high total-to-unique values ratio means that this matrix is fitting for the CSR-VI method, while a small one means that this method will most likely result in slowdown. We present experimental results for those matrices of our initial set that satisfy the $ttu > 5$ requirement. The resulting matrix set comprises of 30 matrices. The speedups obtained for each matrix are presented in Figure 7, along with the percentage of the matrix size reduction relative to the original CSR size, which is marked on top of each bar. Additionally, the speedup for each matrix relative to its *ttu* value is presented in Figure 8.

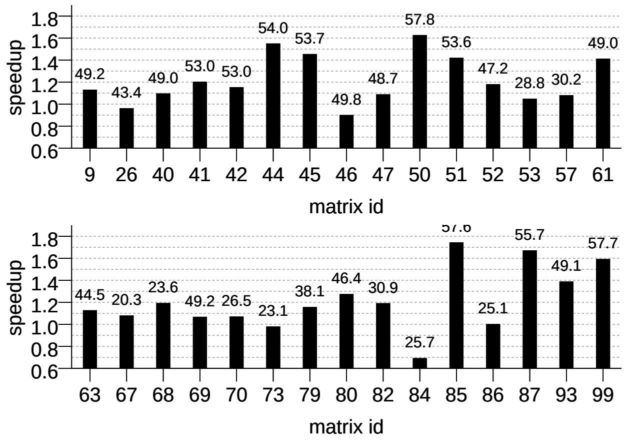


Figure 7: Speedups for the CSR-VI method

The average speedup value for our set is 21.5%. Out of the 30 matrices 4 result in slowdown, one of them being the artificial random matrix #42. The largest speedup (74.1%) appears on matrix #85, while the largest slowdown is 31.1% and appears on matrix 84. Moreover, there is a strong correlation between the *ttu* ratio and the performance of the CSR-VI, as can be seen in Figure 8. Specifically, for *ttu* values larger than 10^3, the speedup is increased almost linearly in a logarithmic scale, until it reaches 10^6. Although the CSR-VI method is not a universal method and cannot be applied to all matrices indiscriminately, there is a large class of matrices which can benefit significantly from it. Additionally, the method is quite simple and can be easily implemented for scientific problems that are known to produce sparse matrices with large *ttu* ratios.

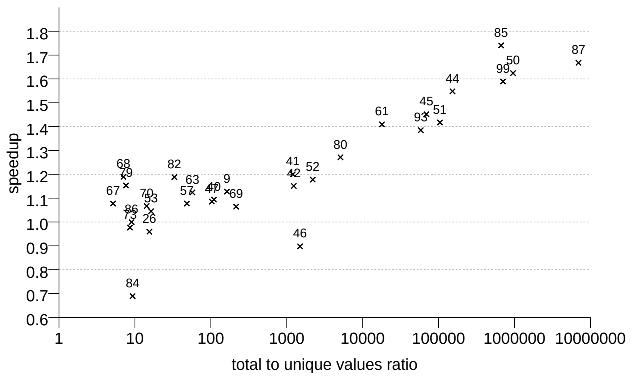


Figure 8: Speedups of the CSR-VI method associated with the total-to-unique values ratio

5. RELATED WORK

5.1 Sparse matrix formats and SpMxV kernel optimization

There is a great deal of work regarding the efficient representation of sparse matrices, not only in terms of required storage but also in terms of computational performance for common operations, such as the SpMxV kernel. Besides CSR, which has been extensively discussed in Section 1, there are a number of other commonly used formats like: BCSR (Blocked-CSR), JD (Jagged Diagonal), CDS (Compressed Diagonal Storage) and Ellpack-Itpack [2, 17]. Moreover, due to the importance of the SpMxV kernel there are numerous works that propose methods for its efficient execution. Toledo [19] evaluates a number of optimization techniques that mainly target RISC processors: (a) precomputing addresses for indirect addressing (b) reordering the matrix to reduce its bandwidth[2] (c) representing nonzeros in small dense blocks, and (d) prefetching to allow cache-hits-under-miss processing. White and Sadayappan [22] state that besides data locality another crucial issue for the performance of SpMxV is small line lengths, which are frequently encountered in sparse matrices and may drastically degrade performance due to ILP reduction. For this reason, the authors propose alternative storage schemes that enable unrolling. Pinar and Heath [15] refer to irregular and indirect accesses on `x` as the main factors responsible for performance degradation. Focusing on indirect accesses, the application of one-dimensional blocking with the BCSR storage format is proposed in order to drastically reduce the number of indirect memory references. In addition, a column reordering technique which enables the construction of larger dense sub-blocks is also proposed. With a primary goal to exploit reuse on vector `x`, Im and Yelick propose the application of register blocking, cache blocking, and reordering [7, 6, 8]. Moreover, their blocked versions of the algorithm are capable of reducing loop overheads and indirect referencing while increasing the degree of ILP. Additionally, the authors also propose a heuristic to determine an efficient block size. Vuduc et al. [20] estimate the performance bounds of the algorithm and evaluate the register blocked code with respect to these bounds. Furthermore, they propose a new approach to select near-optimal register block sizes. Mellor-Crummey and Garvin [12] accentuate the problem of short row lengths and propose the application of the unroll-and-jam compiler optimization in order to overcome this problem. Pichel et al. [13] model the inherent locality of a specific matrix with the use of distance functions and improve this locality by applying reordering to the original matrix. The same group proposes also the use of register blocking to further increase performance in [14]. Vuduc et al. [21] extend the notion of blocking in order to exploit variable block shapes by decomposing the original matrix to a proper sum of submatrices storing each submatrix in a variation of the BCSR format. Finally, Silva and Wait [18] propose an alternative storage format where both the indices and the numerical values are kept in a single data structure to reduce cache misses.

5.2 Index Compression

A large number of research works on SpMxV optimization achieve considerable performance improvement due to reduction of the index data of the sparse matrix. Typical examples are blocking methods such as BSCR and VBR [16] that store only per-block index information, which leads to index data reduction and alleviates the pressure from the memory subsystem. Nevertheless, only a small number of works in the literature target explicitly the compression of the index data. Lee et al. in [11] exploit matrix symmetry by storing only half the matrix (reducing significantly both value and index data) and Williams et al. [24] apply a simple index reduction technique, in which 16-byte indices are used when it is possible. A notable work regarding index compression in the SpMxV context is [23] which, to our knowledge, is the first to apply a delta encoding scheme. In this paper, Willcock and Lumsdaine propose two methods: DCSR which compresses the column indices using a byte-oriented delta scheme and RPCSR, which generates matrix-specific dynamic code by applying aggressive compression on column indices patterns. We will focus our comparison on the DCSR method which operates on the same level as CSR-DU. DCSR encodes the matrix using a set of six command codes for operations such as changing rows, performing multiplication and increasing the current value of the column index, in order to represent indices larger than one byte. A problem with this approach is that, due to the fact that each delta value must be decompressed separately, the resulting SpMxV code will more likely suffer from frequent mispredicted branches that lead to performance degradation. This problem is dealt by encoding patterns of frequent instances of six of these commands into groups that are executed sequentially without branches. However, a number of problems are created by this approach: (a) These frequently encountered patterns are matrix specific and thus a technique such as dynamic code generation needs to be employed to avoid the danger of branch misprediction costs for the generic case. (b) Matrices that exhibit large variation with regard to the patterns encountered cannot be optimized with the use of common patterns. (c) This method is quite complex and needs significant implementation effort. For example, as the authors state, they needed architecture specific implementations, some of which were written in assembly language.

The proposed CSR-DU method tackles the problem of branch misprediction performance penalties in the design level and not in the implementation, by using a more coarse grain approach than encoding each delta value separately. This allows for a much simpler and general implementation, while sustaining a small performance gain gap with regard to the DCSR method. A direct comparison between the two methods is difficult since we are not aware of all the implementation details of the DCSR method. We present some rough comparison data between CSR-DU and DSCR in Table 3. We use the speedups reported in [23] for the Opteron architecture for the DSCR and results for the CSR-DU method from experiments conducted in a similar system[3]. The matrix set we used is the one presented in [23], but without the symmetric matrices, which were specifically optimized and the `stomach` matrix which has many short rows.

Although there can be no direct comparison between these results, a rough conclusion is that the performance gap between the two methods is small. Thus, we argue that CSR-DU provides an antagonistic alternative method for index compression and especially when a simple or portable implementation is required, which is also supported by the fact that there exist matrices, for which CSR-DU outperforms DCSR. Moreover, we argue that CSR-DU can provide a more fitting starting basis for implementing a more

[2]the bandwidth of a sparse matrix is the maximum distance, in diagonals, between two non-zero elements of the matrix.

[3]2× Dual Core AMD Opteron(tm) 265 Processors, 64-bit 2.6.23 linux operating system and version 4.2 of gcc

matrix	DSCR (%)	CSR-DU (%)
cage12	1.68	11.36
cage13	-1.03	8.31
cage14	-3.13	2.34
e40r5000	26.35	9.55
lhr71	18.85	8.43
li	14.87	9.38
pre2	3.57	2.51
rim	17.28	10.57
stomach	19.99	14.45
twotone	7.21	8.07
xenon2	16.79	16.25
average	11.13	9.20

Table 3: Speedups for an Opteron system for the DSCR and for the CSR-DU methods

advanced system which will support different types of units as modules. Each of these modules would provide functions for detection and dynamic code generation for different types of possible regularities met in the matrix and thus allow for a wide variety of optimizations. An indication to the above is the fact that the RPCSR method uses groups of intervals, which are similar to the concept of units used in CSR-DU.

5.3 Value Compression

To our knowledge there is little previous work that targets the compression of the matrix numerical values in the context of SpMxV optimization, despite the fact that in the common scenario these value constitute the largest part of the working set data. A number of works discuss the usage of single precision numerical values as an optimization method. For example Keyes in [9], elaborating on performance improvements on PDE solvers, proposes the use of lower precision representation for poorly known data like preconditioner matrix coefficients, which do note pose problems in the convergence procedure. Moreover, there are works such as [10] that propose mixed precision algorithms, which deliver double precision arithmetic, while performing the bulk of the work in single precision. Although these works focus more on the exploitation of computational characteristics of modern architectures (e.g. vectorization), they also contribute significantly to the reduction of the required memory bandwidth. Other related scientific work include general compression methods for double precision floating point values. For example Burtscher in [3] proposes such a method which is based on value predictors and it is mostly aimed at scientific data transfers in the context of distributed memory architectures, such as clusters. Although methods as this impose great overhead and cannot be directly applied in the context of SpMxV kernel optimization, they can serve as guidelines for the development of more advanced schemes for value compression.

6. CONCLUSIONS – FUTURE WORK

Guided by previous research that identifies memory bandwidth as the main performance bottleneck of the SpMxV kernel [1, 5] we propose two methods for the compression of sparse matrices. The first, called CSR-DU, performs compression on the matrix index data, while the second, called CSR-VI, targets the numerical values of the matrix. Both achieve significant speedups over a substantial number of matrices. More specifically, the CSR-DU method exploits the redundancy of `col_ind` values using a delta encoding scheme. It can be applied successfully to most matrices that do not contain a large number of short rows and thus it can be characterized as a general method. The performance gain achieved by the use of this method depends on the percentage of the index data size over the total matrix size. In the case of 32-bit indexed matrices, which contain 64-bit numerical values this percentage is close to 1/3 and thus the performance gain is bound by this factor. Nevertheless, we argue that matrices which require 64-bit addressing for their indices will be able to fit into the main memory of near-future machines and thus index compression methods such as CSR-DU will achieve significantly higher speedups and become more important. Numerical values of sparse matrices are more difficult to compress than indices for two main reasons: (a) The constrained regularity they exhibit and (b) The limits imposed by the nature of the representation of floating point numbers. The CSR-VI method proposed in this paper exploits redundancy in the values of matrices that contain a large number of common elements, using indirect access. Although the requirement for a large number of common values limits the generality of this method, we have found that it can be applied to a large number of practical matrices achieving substantial speedups. We have deployed and evaluated our methods using the CSR storage format but they are generic enough to be applied to other formats as well. However, the employment of the CSR-DU method to a storage scheme with reduced index data (e.g. BSCR) would not normally lead to a performance improvement. As a general conclusion, we argue that a method of compression can be beneficial for the SpMxV kernel as long as the decompression method does not burden the processor with additional branches that are irregular and thus hard to predict. This is due to the fact that, as shown in [5], the kernel is very sensitive to extra operations, loop overheads and branch mispredictions that can easily harm its performance.

As future work we intend to explore various aspects of the compression schemes presented in this paper. With regard to the CSR-DU method we plan to evaluate various extensions towards more aggressive optimizations. Examples include the exploitation of more complex regularities in the structure of the matrix (e.g sequential or diagonal elements), the usage of advanced techniques such as automatic code generation to mitigate branch prediction costs and methods for performing matrix-wide compression by exploiting regularities in the structure of multiple rows. Furthermore, we plan to explore optimization opportunities by extending the CSR-VI method. A scheme similar to delta encoding can be developed for compressing the CSR-VI indices using the XOR operation. Additionally, the XOR operation can be used in a similar way to implement compression for matrices whose values are not equal but their differences are very small, since a large part of the leading bits used will be common for all values. Finally, we plan to evaluate the usage of a combined method where both index and data compression is applied. Another future work direction is the usage of compression methods for the optimization of SpMxV kernel in shared memory parallel architectures such as SMP or multicore systems. In these systems a large part of the memory hierarchy is shared and thus the memory bandwidth bottleneck is more eminent, due to simultaneous accesses to the main memory by a number of different processors.

7. REFERENCES

[1] W. K. Anderson, W. D. Gropp, D. K. Kaushik, D. E. Keyes, and B. F. Smith. Achieving high sustained performance in an unstructured mesh cfd application. In *SC '99: Proceedings of the 1999 ACM/IEEE conference on Supercomputing*, page 69, New York, NY, USA, 1999. ACM.

[2] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. M. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. V. der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods.* SIAM, Philadelphia, 1994.

[3] M. Burtscher and P. Ratanaworabhan. High throughput compression of double-precision floating-point data. In *DCC '07: Proceedings of the 2007 Data Compression Conference*, pages 293–302, Washington, DC, USA, 2007. IEEE Computer Society.

[4] T. Davis. University of Florida sparse matrix collection. *NA Digest*, 97(23):7, 1997.

[5] G. Goumas, K. Kourtis, N. Anastopoulos, V. Karakasis, and N. Koziris. Understanding the performance of sparse matrix-vector multiplication. In *PDP '08: Proceedings of the 16th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, 2008.

[6] E. Im. *Optimizing the performance of sparse matrix-vector multiplication.* PhD thesis, University of California, Berkeley, May 2000.

[7] E. Im and K. Yelick. Optimizing sparse matrix-vector multiplication on SMPs. In *9th SIAM Conference on Parallel Processing for Scientific Computing.* SIAM, Mar. 1999.

[8] E. Im and K. Yelick. Optimizing sparse matrix computations for register reuse in SPARSITY. *Lecture Notes in Computer Science*, 2073:127–136, 2001.

[9] D. Keyes. *Four Horizons for Enhancing the Performance of Parallel Simulations Based on Partial Differential Equations.* Springer, 2000.

[10] J. Langou, J. Langou, P. Luszczek, J. Kurzak, A. Buttari, and J. Dongarra. Exploiting the performance of 32 bit floating point arithmetic in obtaining 64 bit accuracy (revisiting iterative refinement for linear systems). In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 113, New York, NY, USA, 2006. ACM.

[11] B. Lee, R. Vuduc, J. Demmel, and K. Yelick. Performance models for evaluation and automatic tuning of symmetric sparse matrix-vector multiply. In *ICPP '04: Proceedings of the International Conference on Parallel Processing*, pages 169–176 vol.1, 15-18 Aug. 2004.

[12] J. Mellor-Crummey and J. Garvin. Optimizing sparse matrix-vector product computations using unroll and jam. *International Journal of High Performance Computing Applications*, 18(2):225, 2004.

[13] J. C. Pichel, D. B. Heras, J. C. Cabaleiro, and F. F. Rivera. Improving the locality of the sparse matrix-vector product on shared memory multiprocessors. In *PDP*, pages 66–71. IEEE Computer Society, 2004.

[14] J. C. Pichel, D. B. Heras, J. C. Cabaleiro, and F. F. Rivera. Performance optimization of irregular codes based on the combination of reordering and blocking techniques. *Parallel Computing*, 31(8-9):858–876, 2005.

[15] A. Pinar and M. T. Heath. Improving performance of sparse matrix-vector multiplication. In *Supercomputing'99*, Portland, OR, Nov. 1999. ACM SIGARCH and IEEE.

[16] Y. Saad. SPARSKIT: A basic tool kit for sparse matrix computations. Technical report, Computer Science Department, University of Minnesota, Minneapolis, MN 55455, June 1994. Version 2.

[17] Y. Saad. *Iterative Methods for Sparse Linear Systems.* SIAM, Philadelphia, PA, USA, 2003.

[18] M. Silva. Sparse matrix storage revisited. In *CF '05: Proceedings of the 2nd conference on Computing frontiers*, pages 230–235, New York, NY, USA, 2005. ACM.

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

[20] R. Vuduc, J. Demmel, K. Yelick, S. Kamil, R. Nishtala, and B. Lee. Performance optimizations and bounds for sparse matrix-vector multiply. In *Supercomputing*, Baltimore, MD, Nov. 2002.

[21] R. W. Vuduc and H. Moon. Fast sparse matrix-vector multiplication by exploiting variable block structure. In *High Performance Computing and Communications*, volume 3726 of *Lecture Notes in Computer Science*, pages 807–816. Springer, 2005.

[22] J. White and P. Sadayappan. On improving the performance of sparse matrix-vector multiplication. In *HiPC '97: 4th International Conference on High Performance Computing*, 1997.

[23] J. Willcock and A. Lumsdaine. Accelerating sparse matrix computations via data compression. In *ICS '06: Proceedings of the 20th annual international conference on Supercomputing*, pages 307–316, New York, NY, USA, 2006. ACM Press.

[24] S. Williams, L. Oilker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, Reno, NV, Nov. 2007.

Matrix	`nrows`	`nnz`	ws(MB)	uvals	Matrix	`nrows`	`nnz`	ws(MB)	uvals
1.dense	1000	1000000	11.463	30306	51.Hamrle3	1447360	5514242	90.712	53
2.raefsky3	21200	1488768	17.442	671474	52.ASIC_320ks	321671	1827807	27.053	832
3.olafu	16146	515651	6.209	506318	53.Si87H76	240369	5451000	66.966	334180
4.bcsstk35	30237	740200	9.048	735126	54.SiNa	5743	102265	1.280	24317
5.venkat01	62424	1717792	20.849	1626036	55.ship_001	34920	2339575	27.440	1209604
6.crystk02	13965	491274	5.889	472350	56.af_5_k101	503625	9027150	112.913	9027150
7.crystk03	24696	887937	10.633	870155	57.ASIC_680k	682862	3871773	57.333	80211
8.nasasrb	54870	1366097	16.680	589225	58.bcsstk37	25503	583240	7.161	547869
9.3dtube	45330	1629474	19.512	10000	59.bmw3_2	227362	5757996	70.232	4126650
10.ct20stif	52329	1375396	16.738	1017666	60.bundle1	10581	390741	4.673	374610
11.af23560	23560	484256	5.991	310481	61.cage13	445315	7479343	94.088	417
12.raefsky4	19779	674195	8.093	652375	62.turon_m	189924	912345	14.063	284875
13.ex11	16614	1096948	12.870	285032	63.ASIC_680ks	682712	2329176	39.677	40708
14.rdist1	4134	94408	1.159	94104	64.thread	29736	2249892	26.315	2085970
15.av41092	41092	1683902	20.054	1199095	65.e40r2000	17281	553956	6.669	374723
16.orani678	2529	90158	1.080	49455	66.sme3Da	12504	874887	10.251	650304
17.rim	22560	1014951	12.045	991013	67.fidap011	16614	1091362	12.807	211502
18.memplus	17758	126150	1.782	51595	68.fidapm11	22294	623554	7.561	88276
19.gemat11	4929	33185	0.474	32524	69.gupta2	62064	2155175	25.848	10000
20.lhr10	10672	232633	2.866	204020	70.helm2d03	392257	1567096	25.416	109526
21.goodwin	7320	324784	3.856	74020	71.hood	220542	5494489	67.086	5048077
22.bayer02	13935	63679	0.995	25541	72.inline_1	503712	18660027	223.155	18016122
23.bayer10	13436	94926	1.343	35815	73.language	399130	1216334	21.533	141504
24.coater2	9540	207308	2.554	168772	74.ldoor	952203	23737339	289.814	21675099
25.finan512	74752	335872	5.270	335867	75.mario002	389874	1167685	20.799	547809
26.onetone2	36057	227628	3.293	14672	76.nd12k	36000	7128473	82.266	4857071
27.pwt	36519	181313	2.772	10000	77.nd6k	18000	3457658	39.913	2367789
28.vibrobox	12328	177578	2.267	23247	78.pwtk	217918	5926171	71.976	5592868
29.wang4	26064	177168	2.525	176	79.rail_79841	79841	316881	5.149	41551
30.lnsp3937	3937	25407	0.366	4176	80.rajat31	4690002	20316253	321.956	3985
31.lns_3937	3937	25407	0.366	4176	81.rma10	46835	2374001	28.062	1223223
32.sherman5	3312	20793	0.301	15096	82.s3dkq4m2	90449	2455670	29.828	74283
33.sherman3	5005	20033	0.325	11027	83.nd24k	72000	14393817	166.097	9731838
34.orsreg_1	2205	14133	0.204	111	84.af_shell9	504855	9046865	113.162	968711
35.saylr4	3564	12940	0.216	11	85.kim2	456976	11330020	138.378	17
36.shyy161	76480	329762	5.233	196333	86.rajat30	643994	6175377	82.955	683418
37.wang3	26064	177168	2.525	176	87.fdif	4000000	27840000	394.897	4
38.mcfe	765	24382	0.294	24381	88.sme3Db	29067	2081063	24.370	1552542
39.jpwh_991	991	6027	0.088	14	89.stomach	213360	3021648	38.650	2257584
40.gupta1	31802	1098006	13.172	10000	90.thermal2	1228045	4904179	79.547	4819424
41.lp_cre_b	9647	260785	3.426	217	91.F1	343791	13590452	162.088	13038962
42.lp_cre_d	8894	246614	3.240	199	92.torso3	259156	4429042	55.629	3121632
43.lp_fit2p	3000	50284	0.673	560	93.cage14	1505785	27130349	339.203	465
44.lp_nug20	15240	304800	3.998	2	94.audikw_1	943695	39297771	467.727	37023578
45.apache2	715176	2766523	45.301	40	95.Si41Ge41H72	185639	7598452	90.498	4665454
46.random	100000	14977726	173.314	10000	96.crankseg_2	63838	7106348	82.543	4397887
47.bcsstk32	44609	1029655	12.634	10000	97.Ga41As41H72	268096	9378286	112.439	3597854
48.msc10848	10848	620313	7.306	617922	98.af_shell10	1508065	27090195	338.787	10889891
49.msc23052	23052	588933	7.179	575636	99.boneS10	914898	28191660	340.078	40
50.bone010	986703	36326514	434.544	38	100.msdoor	415863	10328399	126.131	9777773

Table 4: Matrix suite including the CSR SpMxV working set (ws) assuming 32-bit indices and 64-bit values and the number of unique values (uvals) for each matrix