Improving the Performance of Multithreaded Sparse Matrix-Vector Multiplication using Index and Value Compression

Kornilios Kourtis, Georgios Goumas and Nectarios Koziris
National Technical University of Athens
School of Electrical and Computer Engineering
Computing Systems Laboratory
{kkourt, goumas, nkoziris}@cslab.ece.ntua.gr

***Abstract*—The Sparse Matrix-Vector Multiplication kernel exhibits limited potential for taking advantage of modern shared memory architectures due to its large memory bandwidth requirements. To decrease memory contention and improve the performance of the kernel we propose two compression schemes. The first, called CSR-DU, targets the reduction of the matrix structural data by applying coarse grain delta encoding for the column indices. The second scheme, called CSR-VI, targets the reduction of the numerical values using indirect indexing and can only be applied to matrices which contain a small number of unique values. Evaluation of both methods on a rich matrix set showed that they can significantly improve the performance of the multithreaded version of the kernel and achieve good scalability for large matrices.**

I. Introduction

In recent years the processor industry has performed a technology shift towards chip multiprocessor (CMP – multicore) designs due to the difficulties in trying to achieve higher performance using conventional techniques such as frequency scaling [1], [2]. As a result the research community has a revitalized interest in shared memory architectures and the problem of application scalability up to a large number of processing cores is considered of vital importance. Different classes of applications have different scalability properties with regard to shared memory architectures. Applications characterized by good temporal locality scale well, since each core can work independently using local data residing in its cache, without interfering with the operation of other cores. On the other hand, applications with streaming access patterns are characterized by poor temporal locality and tend to exhibit poor scaling due to contention on the memory subsystem.

An important and ubiquitous computational kernel with streaming memory accesses is the Sparse Matrix-Vector multiplication (SpMxV). SpMxV is used in a large variety of applications in scientific computing and engineering. For example, it is the basic operation of iterative solvers, such as Conjugate Gradient (CG) and Generalized Minimum Residual (GMRES), extensively used to solve sparse linear systems resulting from the simulation of physical processes that are described by partial differential equations [3]. Furthermore, SpMxV is a member of one of the ”seven dwarfs”, which are classes of applications that are believed to be important for at least the next decade [4].

The distinguishing characteristic of sparse matrices is that they are populated by a large number of zero elements. Thus, it is highly inefficient to perform operations on these matrices using typical (dense) array structures. Instead, special storage schemes are used, which target not only the efficient storage of the matrix in terms of space, but also the efficient execution of various operations by performing only the necessary actions. Thus, the common approach is to store only the non-zero values of the matrix, and employ additional indexing information about the position of these values. In this paper a distinction will be made between data that are used for the representation of the matrix structure (*index data*), and data that represent the numerical values of the matrix elements (*value data*).

In our recent work [5], as well as in related literature [6], the memory subsystem, and more specifically the memory bandwidth, has been identified as the main performance bottleneck of the SpMxV kernel when executed in a uniprocessor environment. Obviously, if more processing elements access the main memory through a common bus, this performance bottleneck will become more severe. Consequently, it is expected that a multithreaded version of the kernel, targeted for shared memory architectures, will have poor performance scaling as the number of processing elements increases. An approach for alleviating this problem is the reduction of the data that need to be accessed during the execution of the kernel (*working set*). In this direction and using the standard *CSR* [3], [7] sparse format as a starting point, we propose two storage formats: *CSR-DU* and *CSR-VI* [8]. CSR-DU is a general format that reduces the index data using coarse grain delta encoding for the compression of column indices, while CSR-VI is a specialized format that exploits the redundancy of matrices with a large number of common values using indirect value accesses. The intrinsic basis of compression is to trade data storage volume for computation. We argue that as the number of processing elements that share the memory subsystem increases, this tradeoff will become more beneficial for the performance of memory bound applications such as SpMxV, even if it results in degraded performance in the uniprocessor case.

We perform an experimental evaluation of the benefits of the aforementioned formats in a multithreaded environment. Our experimental results confirm that the multithreaded version of the SpMxV kernel exhibits poor scalability in a typical modern shared memory architecture and that the proposed compression schemes can alleviate the pressure on the memory subsystem leading to significant performance improvement. The rest of the paper is organized as follows: Section II provides an introduction to various issues that are related to this work and sets the context for the rest of the paper, while Section III discusses the related literature. Sections IV and V briefly present the two compression methods and Section VI contains the results of the experimental performance evaluation of the methods in question in a shared memory architecture. The paper is concluded in Section VII.

II. Preliminaries

A. Shared Memory Architectures

While shared memory architectures have been studied extensively in the past [9], the current trend of multicore processors, along with indications for many-core next-generation processors [10] has motivated the research community to revisit the performance issues of shared memory architectures and to investigate methods for allowing applications to scale up to a large number of processing units. A difference between multicore processors and classic SMP systems is that in the former different cores may share a part of the cache hierarchy (e.g. the L2 or the L3 cache). Cache sharing is an important factor of the system's performance and can be either *constructive* or *destructive* for a given application, depending on whether threads scheduled on the cores that share a cache operate on common data or not.

B. Sparse Matrix Formats and SpMxV

The most commonly used storage format for sparse matrices is the Compressed Sparse Row (CSR) format [3], [7]. In CSR the matrix is stored in three arrays: `values`, `row_ptr` and `col_ind`. The `values` array stores the non-zero elements of the matrix in row-major order, while the other two arrays store indexing information: `row_ptr` contains the location of the first (non-zero) element of each row within the `values` array and `col_ind` contains the column number for 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 `row_ptr` array size is equal to the number of rows (`nrows`) plus one. An example of the CSR format for a 6×6 sparse matrix is presented in Fig. 1. Other generic formats are the Compressed Sparse Column (CSC), which is similar to *CSR* storing columns instead of rows, and the Coordinate format (COO), where each non-zero is stored as a triplet along with the coordinates of its location in the matrix.

The SpMxV operation (`y` $= A$`x`), is the multiplication of a sparse matrix A with a (dense) vector `x` with the result stored in another (dense) vector `y`. The operation is easily implemented for matrices stored in CSR form. The SpMxV code for a matrix with N rows in CSR format is:

```
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]];
```

The working set (`ws`) of the SpMxV operation consists of the matrix and vector data and its size is expressed by the following formula:

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

where idx_s and val_s is the memory size required for the storage of an index and a value respectively. Since for real-life sparse matrices it holds $nnz \gg nrows, ncols$, the most dominant terms of the working set is the size of the `col_ind` and `values` arrays, which have `nnz` elements. Commonly, 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, normally require double precision, so the typical 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. Thus, in the scenario of 4-byte indices and 8-byte values, it is evident that the value data size reduction can be more beneficial in terms of the total `ws` reduction. Another observation is that the vast majority of the data are accessed in a streaming fashion, hence the characterization of the SpMxV kernel as a streaming application in previous paragraphs.

C. Parallelizing SpMxV

There are a number of partitioning schemes for parallelizing the SpMxV kernel on a shared memory architecture. In the case of the CSR format the coarse grain *row partitioning* scheme is usually applied [11], where different blocks of rows are assigned to different threads (see Fig. 2). Each thread operates on different parts of the `row_ptr`, `col_ind`, `values`, and `y` arrays, while all threads access elements on the `x` array. Since access on `x` is read only, the data can reside in each processor's cache without causing invalidation traffic due to the cache coherency protocol. In theory, the common use of `x` offers potential for constructive cache sharing, but in practice this potential is not realized, since there is limited space for the rest of the data, which are disjoint for each thread.

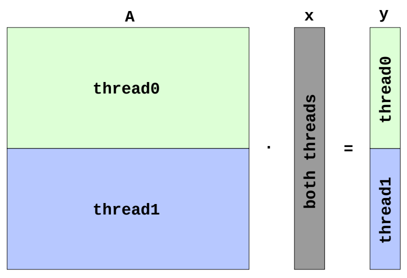
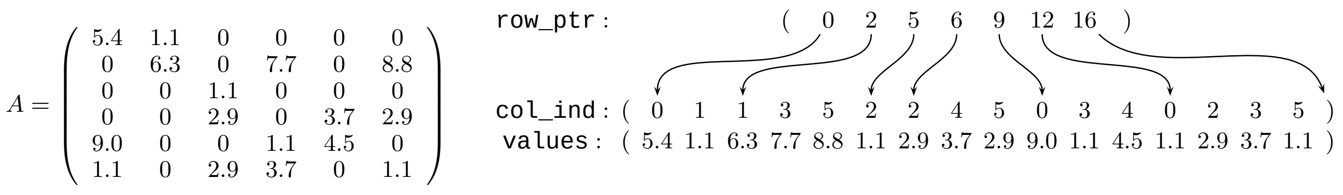


Fig. 2: Row partitioning on SpMxV

The complementary approach to row partitioning is *column partitioning*, where each thread is assigned a block of columns.



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

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)

Fig. 1: Example of the CSR Storage Format

Although this approach is more naturally applied to the CSC format, it can also be applied to the CSR format. An advantage of column partitioning is that each thread operates on a different part of the `x` array, which allows for better temporal locality on the array's elements in case of distinct caches. A disadvantage, on the other hand, is that all threads must perform writes on all the elements of the `y` array. To avoid cache-line ping-pongs the best practice is to have each thread use its own `y` array and perform a reducing addition at the end of the multiplication. Additionally, in the case of the CSR format column partitioning may lead to empty or small rows, and to a degradation of performance due to loop overheads. The combined method of *block partitioning*, where each thread is assigned an arbitrary two-dimensional block, has the additional benefit of allowing configurable data sizes for each thread. This property is important for machines with processing elements that have limited memory space (e.g. the Cell processor). The interaction of each of the above parallelization approaches with specific sparse matrices and their effect on performance are beyond the scope of this paper and will be a matter of future research.

For our experimental evaluation we used row partitioning. It can be easily implemented for CSR and the effect of the proposed compression methods on the performance of the SpMxV kernel are, for the most part, orthogonal to the partition method used. Another issue with regard to the parallelization of the kernel is the balancing of the workload of each thread. We applied a static balancing scheme based on the non-zero elements, where each thread is assigned approximately the same number of elements and thus the same number of floating-point operations.

III. Related Work

A. Serial SpMxV

Due to the importance of SpMxV there is an abundance of scientific work targeting the optimization of the serial version of the kernel. A number of alternatives to CSR have been proposed such as BCSR (Blocked-CSR), JD (Jagged Diagonal), CDS (Compressed Diagonal Storage) and Ellpack-Itpack [3], [7]. These formats try to exploit regularities in the structure of the sparse matrix in order to reduce the storage requirements and the execution time of SpMxV. Moreover, there is a large number of works that propose optimization techniques for the efficient execution of the kernel. Several of these works [12]–[15] aim at the optimization of the irregular and indirect accesses on the `x` vector using methods such as matrix reordering, register blocking and cache blocking. Other works [16], [17] are concerned with the performance problems that arise in matrices with a large number of rows with small length.

B. Index Compression

A significant part of the SpMxV optimization techniques reported in the related literature result in index data reduction. Typical examples are blocking methods such as BSCR and VBR [18] that store only per-block index information. To our knowledge, the only work that explicitly targets the compression of the index data is [19]. In this paper, Willcock and Lumsdaine propose two methods: *DCSR*, which compresses column indices using a byte-oriented delta encoding scheme to exploit the highly redundant nature of the `col_ind` array and *RPCSR*, which generates matrix-specific dynamic code by applying aggressive compression on column indices patterns for the whole matrix. 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 primitive sub-operations that can be used to implement the SpMxV kernel. Examples of such sub-operations are the increment of the current row and column index, and the multiplication of a number of the matrix values with the appropriate vector elements. A significant performance problem of this approach is that the decoding of these sub-operations must be performed very often, which results in frequent mispredicted branches. This problem is dealt by a form of unrolling, where patterns of frequent instances of six of these sub-operations are grouped together allowing them to be executed sequentially without branches. Contrarily, our approach, which is also based on delta encoding, tackles the problem of branch misprediction performance penalties in a more basic level by being more coarse grained. This allows for a much simpler and general implementation, while sustaining a small performance gain gap with regard to the DCSR method. Moreover, it can handle worst-case scenarios of the DCSR method such as matrices that exhibit large variation with regard to the patterns encountered. A more detailed comparison of the two methods can be found in [8].

C. Value Compression

Despite that, in the common case, the value data constitute the larger part of the working set of SpMxV, there has been little research effort targeting its reduction. Lee et al. [20] exploit matrix symmetry by storing only half the matrix (reducing significantly both value and index data). Additionally, there exist a number of works in the general area of scientific computation that are related to the value compression for the SpMxV kernel. Keyes [21], proposes the use of lower precision representation for data that do not pose problems in the convergence procedure, while Langou et al. [22] propose mixed precision algorithms, which deliver double precision arithmetic, while performing the bulk of the work in single precision. Even though these works target more on the exploitation of characteristics of modern architectures (e.g. vectorization), they also contribute significantly to the required memory bandwidth reduction. In a different context Burtscher [23] proposes a method for the efficient compression of double precision floating point values targeting network data transfers.

D. *Multithreaded SpMxV*

As far as the multithreaded version of the code is concerned, past work focuses mainly on SMP clusters, where researchers either apply and evaluate known uniprocessor optimization techniques (e.g. register and cache blocking) on SMPs, or examine reordering techniques to improve locality of references and minimize communication cost [24]–[27]. Recently, Williams et al. [11] presented an evaluation of SpMxV on a set of emerging multicore architectures. Their study covers a wide and diverse range of high-end chip multiprocessors, including recent multicores from AMD (Opteron X2) and Intel (Clovertown), Sun's Niagara2 and platforms comprised of one or two Cell processors. Their work includes a rich collection of optimizations, including some that are targeted specifically at multithreading architectures on a set of 14 matrices. In their conclusions they state that memory bandwidth could be a significant bottleneck and advocate working set reduction techniques. It should also be noted that one of the optimizations they apply is a simple index reduction technique, in which 16-byte indices are used when this is applicable.

IV. Index Compression

Our general approach for the compression of the index data is to search for regularities in the sparse matrix and exploit them by using specially tailored run-time methods. Hence, in our scheme the matrix is logically divided into areas, called *units*, each of which is characterized by its regularity type. More specifically, we target the exploitation of matrix areas that exhibit some level of density, without necessarily containing contiguous non-zero elements. This is achieved using a delta-encoding scheme. In our storage format, called *CSR-DU* for CSR Delta Unit, each unit type is characterized by the required storage size for the delta values that express the column distance between consecutive non-zeros. For example, areas for which the distance between all consecutive column indices is less than $2^8 = 256$, require only one byte for the storage of each delta value. This approach compared to the separate encoding of each delta value using variable length integers achieves less data size reduction, but allows for innermost loops with minimum overheads, if the unit size is large enough. It should be noted that a limitation of CSR-DU is that units can not span multiple rows, which results in small units for rows with a small number of elements.

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	3,1
5	u8, NR	4	0	2,1,2

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

The CSR-DU method uses a byte-array called `ctl` to store all the indexing information required for each unit, which consists of four sections: `uflags`, `usize`, `ujmp` and `ucis`. `uflags` and `usize` have 1 byte size each and contain the unit type and size respectively. `ujmp` is a variable length integer that denotes the distance of the unit's column index from the previous one, while `ucis` is an array of `usize` $-\ 1$ elements, which contains the delta values for the remaining column indices. The storage size of the `ucis` elements (1, 2, 4 or 8 bytes) is stored in `uflags`, along with a flag that marks the beginning of a new row. An example of the information included in the `ctl` structure for the matrix of Fig. 1 is given in Table I. There exist six units in total, 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 simplified code snippet of the SpMxV operation for the CSR-DU format is presented in Fig. 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 code is executed based on the type of the unit. The compression procedure of CSR-DU is straightforward and can be performed in $O(\texttt{nnz})$ steps by scanning the matrix elements once, and keeping appropriate information in buffers until a unit is finalized. This means that the construction process of CSR-DU involves no overhead in terms of time complexity compared to that of CSR. We parallelize using the row partitioning scheme. Both the compression method and the SpMxV kernel can be easily extended to support multiple threads. The information that each thread needs is an offset in the `ctl`, `values` and `y` arrays, to mark the beginning of its data, and the total number of rows that have been assigned to it.

V. Value Compression

Conversely with index data, value data do not inherently contain redundancy in the general case. Nevertheless, we have noticed that a significant number of matrices from our experimental set contain a small number of unique values relative to the total non-zero values (`nnz`). To exploit this redundancy we propose a simple storage format, called CSR-VI for CSR Value Index, where only the unique values are kept, along with indices to them. More specifically, the `values` array of CSR is replaced with two arrays: `vals_unique` and `val_ind`. The first contains the unique matrix values and the second the index in the `vals_unique` array for each of the `nnz` matrix elements. For this scheme to be beneficial in terms of working set reduction, the value indexing data size must be significantly smaller than that of the initial numerical values. A simple approach towards this goal is to enforce smaller

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

   case CSR_DS_U16:
   ...
}
```

Fig. 3: code snippet for the SpMxV kernel for CSR-DU

storage requirements for the individual value indices compared to the original values. Hence, the indices size in our method is determined by the number of the unique values that need to be addressed. For example, if there exist uv unique values and it holds $2^8 < un \leq 2^{16}$, then a 2-byte integer will be used for each value index.

An example of this value structure is presented in Fig. 4, which contains the Fig. 1 matrix values. The SpMxV kernel implementation for CSR-VI is presented in Fig. 5 and can be easily derived from the CSR case by replacing the direct accesses of `values` with an indirect access of `vals_unique` based on the value of `val_ind`. Even though the resulting code includes an additional memory reference for each of the `nnz` elements, it will lead to fewer memory accesses when the number of unique values is relatively small. The compression method for CSR-VI is implemented using a hash table and as in CSR-DU its complexity is $O(\texttt{nnz})$. The multithreaded version is trivially derived from the serial by providing to each thread the first and the last row that it needs to process.

```
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]];
  }
```

Fig. 5: SpMxV kernel for the CSR-VI storage format

VI. Experimental Evaluation

A. Experimental Setup

Our experiments where conducted on an 8-core system, comprising of two Intel Clovertown processors (Fig. 6). The Clovertown processor is a quad core processor which is built by combining two Woodcrest chips. Each of the Woodcrest chips contains two cores with two private 32 KB 8-way caches for instructions and data and a shared unified 4 MB 16-way L2 cache. The processors interface with the main memory with the Intel 5000p Memory Controller Hub, which provides 4 channels of fully buffered DDR2 DIMM memory. All the cores operate on $2\ GHz$.

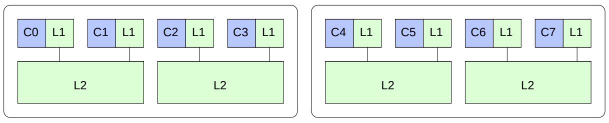


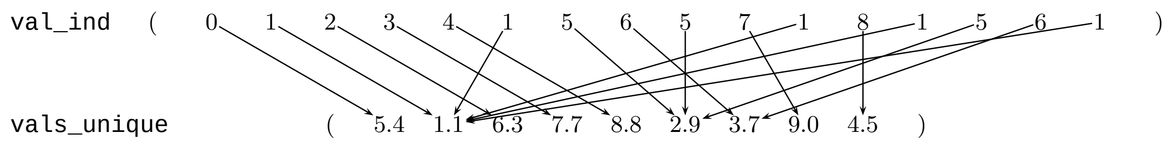
Fig. 6: A 8-core system comprising of two Clovertown processors

The system was running a 64-bit version of Linux (2.6.23) and the compiler used was version $4.2.3$ of gcc with the optimization flag -O3. The storage size for the indices and values were 32 and 64 bits respectively. The parallelization of the various versions of the SpMxV kernel was done explicitly, using the pthread interface of the GNU libc library (NPTL 2.7). Moreover, the `sched_setaffinity()` system call was used to bind the various threads to predefined processors. The results presented in the following sections include experiments for 1, 2, 4 and 8 threads. The threads are always scheduled to run to as “close” as possible processors. For example 2 threads are scheduled on cores which share the L2 cache, unless otherwise stated, while 4 threads are scheduled on the same physical package.

The code for the SpMxV kernel was optimized to write the `y[i]` value at the end of each innermost loop by keeping the intermediate result in a register. 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.

B. Matrix Set

One of our initial requirements was to perform experiments on a rich and diverse set of matrices. In [5] we have presented a performance evaluation of the SpMxV kernel for 100 matrices, the majority of which have been selected from Tim Davis’s collection [28]. These matrices will be used as our basis, and can be identified by their name and id number (see [5]). Two basic classes of matrices can be distinguished, depending on whether the working set of the kernel fits completely into the L2 cache or not. In an iterative SpMxV computation, matrices which have working set larger than the L2 cache may experience capacity misses, while matrices with a smaller working set experience only compulsory misses and generally perform better. Since in this work we are mainly concerned with matrices that perform poorly due to memory bandwidth limitations, we only consider matrices from the second class. Hence, we reject matrices with working set less than $3/4$ of the L2 size, in order to also cover border-line cases (e.g. memory accesses due to conflict misses), which for our systems translates to $ws \geq 3\ MB$. We also reject the dense matrix. The resulting set consists of 77 matrices (2-13, 15 17,



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)

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

21, 25, 26, 36, 40-42, 44-53, 55-100) and we will refer to it as $\boldsymbol{\mathcal{M}_0}$.

The utilization of multiple cores with separate caches increases the total cache size available to the kernel. Thus, it is possible for the working set of a matrix in $\mathcal{M}_0$, to fit completely in the system's cache as more cores are used. In this case the kernel is expected to exhibit significant speedup, which may even be superlinear. Conversely, the performance of matrices that are too large to fit in the total L2 cache will differ significantly from the rest. Thus, we divide $\mathcal{M}_0$ into two subsets: $\boldsymbol{\mathcal{M}_L}$, which contains matrices with a working set that is larger or equal to $4 \times L2 + 1MB = 17MB$ (2, 5, 8-10, 15, 40, 45, 46, 50-53, 55-57, 59, 61-64, 69-78, 80-100) and $\boldsymbol{\mathcal{M}_S}$, which contains the rest of the matrices.

C. CSR Performance

Table II presents overall results (average, maximum and minimum) for the performance of the CSR SpMxV kernel for the $\mathcal{M}_S$, $\mathcal{M}_L$ and $\mathcal{M}_0$ sets. The results of the serial version are expressed in floating operations per second (FLOPS), while the results for the multithreaded versions in terms of speedup relative to the corresponding serial version. There are two results presented for 2 threads, one for cores with a shared cache and one for cores on the same die but with separate caches, which confirm that cache sharing is destructive for SpMxV. As expected, CSR scales rather poorly (3.44) for 8 threads, and especially for matrices that belong in the $\mathcal{M}_L$ set (2.12), due to large memory bandwidth requirements. On the other hand, CSR performance for matrices of the $\mathcal{M}_S$ set scale relatively well for 8 cores (6.19) due to reduced memory contention on the bus.

	$\mathcal{M}_S$ (25 matrices)			$\mathcal{M}_L$ (52 matrices)			$\mathcal{M}_0$
core(s)	avg	max	min	avg	max	min	avg
1	**619.4**	886.6	465.2	**477.8**	594.4	202.4	**523.6**
2 $(1{\times}L2)$	**1.17**	1.62	0.90	**1.15**	1.40	1.07	**1.16**
2 $(2{\times}L2)$	**1.93**	2.59	1.24	**1.24**	1.47	1.09	**1.46**
4	**2.63**	4.32	1.54	**1.28**	1.73	1.12	**1.72**
8	**6.19**	8.71	2.12	**2.12**	6.30	1.58	**3.44**

TABLE II: overall CSR SpMxV performance (serial and multithreaded)

D. CSR-DU

Fig. 7 provides a comparison of the CSR and CSR-DU methods by showing CSR-DU speedups relative to the CSR serial version for each matrix in $\mathcal{M}_0$ (bars), along with the corresponding speedups of the CSR multithreaded version (black squares). In addition the matrix size reduction relative to the original CSR size is also presented (text). It should be noted, that the matrices are sorted by their speedup and each sub-graph has a different scale. Table III presents overall performance improvements of CSR-DU against the CSR version that utilizes the same number of threads, in terms of speedups. The last column for the $\mathcal{M}_S$ and $\mathcal{M}_L$ sets contains the number of matrices for which the usage of CSR-DU results in a non-negligible slowdown ($speedup < 0.98$).

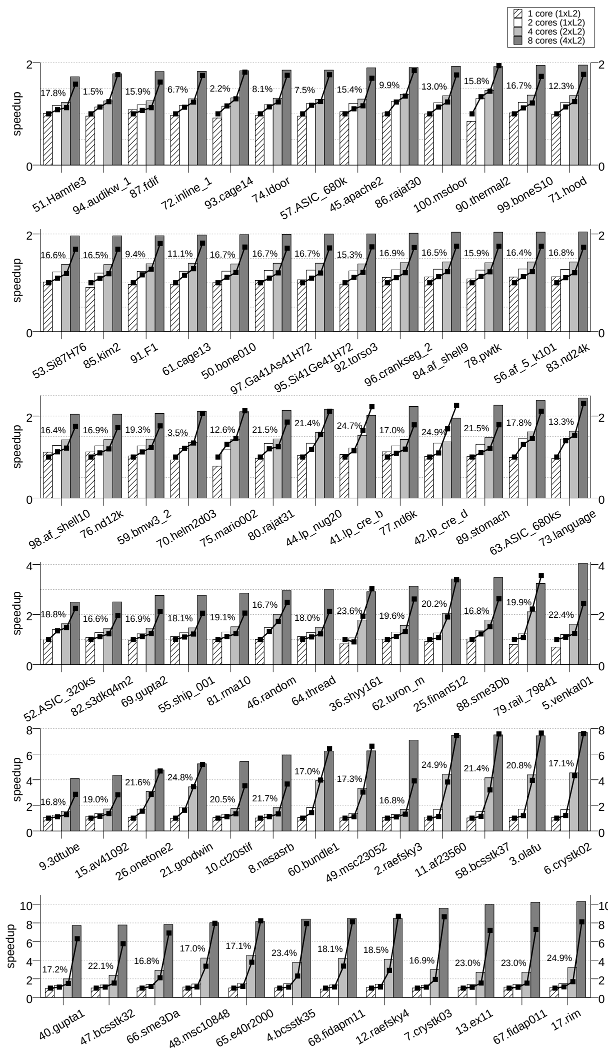


Fig. 7: Detailed performance evaluation of the CSR-DU multithreaded SpMxV kernel

A first observation is that CSR-DU performs better than CSR on average. In the uniprocessor case our method performs similarly with the CSR (2% improvement). This indicates a discrepancy with previous results in [8], which in general showed better performance improvement on a Woodcrest pro-

	$\mathcal{M}_S$ (25 matrices)				$\mathcal{M}_L$ (52 matrices)				$\mathcal{M}_0$
core(s)	avg	max	min	<0.98	avg	max	min	<0.98	avg
1	**1.02**	1.12	0.80	5	**1.01**	1.14	0.69	17	**1.01**
2	**1.24**	1.49	1.06	0	**1.10**	1.19	0.90	2	**1.15**
4	**1.24**	1.89	0.81	4	**1.15**	1.36	0.99	0	**1.18**
8	**1.05**	1.40	0.86	8	**1.20**	1.82	0.99	0	**1.15**

TABLE III: Overall comparison of CSR-DU and CSR multi-threaded versions

cessor. We attribute this change in the lower clock frequency of the Clovertown processor, which makes the computation/memory access tradeoff less effective. Experimental results for a number of matrices in the Woodcrest processor with its frequency reduced to $2\ GHz$ support this claim. Conversely with the serial case, the multithreaded version of CSR-DU achieves more noticeable speedups on the $\mathcal{M}_0$ set: 15%, 18% and 15% for 2, 4 and 8 threads respectively. This suggests that CSR-DU and index compression schemes in general can be beneficial for shared memory architectures, even if the serial performance is similar or worse than that of CSR. For the $\mathcal{M}_S$ set, the performance improvement of CSR-DU drops significantly (from 24% to 5%) for 8 cores since a large part of the working set resides in the cache leaving little space for optimization by its reduction. Moreover, as the number of cores increases so does the number of matrices that exhibit a significant slowdown when the CSR-DU method is applied in the $\mathcal{M}_S$ set (0, 4, 8 for 2, 4 and 8 threads respectively). On the other hand, for matrices of the $\mathcal{M}_L$ set CSR-DU exhibits significant performance scalability as it reaches 20% for 8 cores and there are no matrices for which the CSR-DU results in significant slowdown for 4 or 8 threads.

E. CSR-VI

Contrary to the CSR-DU method, CSR-VI can be applied meaningfully only to matrices with a large number of common values. Thus, to elaborate on the applicability of the method to a given matrix, we consider the *total-to-unique* (ttu) values ratio, which is defined as the fraction of the total values (nnz) to the number of values that are unique in the matrix. A high total-to-unique values ratio indicates that the matrix is fitting for the CSR-VI method, while a small one shows that it will most likely result in slowdown. We use the empirical criterion $ttu \geq 5$ to select the appropriate matrices from $\mathcal{M}_0$. The resulting set consists of 30 matrices and we will refer to it as $\boldsymbol{\mathcal{M}_0^{vi}}$. It should be noted that the $\mathcal{M}_0^{vi}$ matrices constitute approximately the 39% of $\mathcal{M}_0$, which indicates that CSR-VI can be applied to an important number of real-world matrices. By applying an analogous rationale as before, we split the $\mathcal{M}_0^{vi}$ set into two subsets ($\boldsymbol{\mathcal{M}_L^{vi}}$ and $\boldsymbol{\mathcal{M}_S^{vi}}$), separating matrices which are memory-bound even when all the cores are used, from those which are not. $\mathcal{M}_L^{vi}$ contains 22 matrices (9, 40, 45, 46, 50-53, 57, 61, 63, 69, 70, 73, 80, 82, 84-87, 93, 99) and $\mathcal{M}_S^{vi}$ 8 (26, 41, 42, 44, 47, 67, 68, 79). The results of the experimental evaluation for the CSR-VI method are presented in an identical way as the results of the CSR-DU method. Fig. 8 contains detailed results for the CSR-VI and the CSR methods, expressed as speedups relative to the serial CSR performance, and Table IV contains overall comparison results between CSR and CSR-VI.

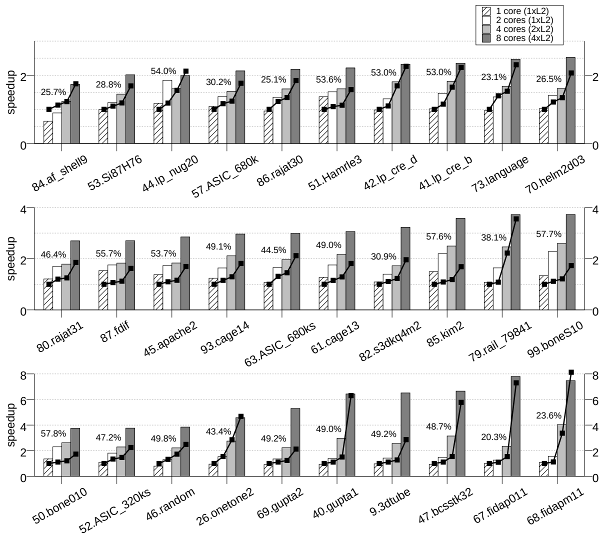


Fig. 8: Detailed performance evaluation of the CSR-VI multithreaded SpMxV kernel

	$\mathcal{M}_S^{vi}$ (8 matrices)				$\mathcal{M}_L^{vi}$ (22 matrices)				$\mathcal{M}_0^{vi}$
core(s)	avg	max	min	<0.98	avg	max	min	<0.98	avg
1	**1.03**	1.17	0.94	2	**1.12**	1.54	0.65	7	**1.10**
2	**1.30**	1.56	0.99	0	**1.36**	2.07	0.80	3	**1.35**
4	**1.25**	2.04	0.96	1	**1.55**	2.16	1.00	0	**1.47**
8	**1.02**	1.15	0.92	3	**1.59**	2.50	0.99	0	**1.44**

TABLE IV: Overall comparison of CSR-VI and CSR multi-threaded versions

The performance behavior of the CSR-VI method is analogous to the one of the CSR-DU. For the serial case CSR-VI achieves a 10% over CSR, which again is significantly less than the results presented in [8] and can be attributed to the lower frequency of the processor in our system. For the multithreaded case the average speedups against CSR over the $\mathcal{M}_0^{vi}$ set are 35%, 47% and 44% for 2, 4 and 8 threads respectively. This fact shows that the CSR-VI method can be very beneficial for matrices with a small number of unique values, when multiple threads are employed. The average performance improvement of $\mathcal{M}_0^{vi}$ is marginally reduced by 3% from 4 to 8 cores, due to the behavior of matrices in the $\mathcal{M}_S^{vi}$ set, for which the CSR-VI speedups drop drastically (from 25% to 2%). On the other hand the matrices in the $\mathcal{M}_L^{vi}$ set remain memory bound and their speedup improves by a small factor (from 55% to 59%) when all 8 cores are used.

VII. Conclusions

We have presented two sparse-matrix storage formats, named CSR-DU and CSR-VI, that target the performance improvement of the multithreaded SpMxV operation by alleviating the contention on the memory subsystem via index and value matrix data compression respectively. More specifically, CSR-DU applies a coarse grain delta encoding for the column indices, while CSR-VI uses indirect indexing for the numerical value data and can be applied only to matrices that exhibit a large number of common values. Both formats exhibit significant performance improvement when compared to the CSR version over a rich set of matrices and especially for those which are large enough to preserve the memory bound nature of the kernel. In addition, the proposed methods are stable, as there was no memory bound matrix for which they resulted in a significant slowdown for 4 or 8 threads when compared to CSR. Since we compare our methods against a CSR version with 32-bit indices and 64-bit values the CSR-VI method achieved substantial better improvement than CSR-DU. It should be noted though, that this imbalance is subject to change, as the available physical memory of machines increases and it becomes possible to support matrices which require 64-bit index addressing. Finally, we argue that our approach designates a general optimization methodology for memory intensive problems (e.g. graph or database algorithms), where compression sacrifices CPU cycles to alleviate the memory pressure and can potentially lead to substantial performance improvements in multithreaded execution, even if it leads to slowdowns in the uniprocessor case.

REFERENCES

[1] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2003.

[2] D. Geer, "Chip makers turn to multicore processors," *IEEE Computer*, vol. 38, no. 5, pp. 11–13, 2005.

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

[4] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The landscape of parallel computing research: A view from berkeley," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2006-183, December 18 2006. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html

[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] 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*. New York, NY, USA: ACM, 1999, p. 69.

[7] 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.* Philadelphia: SIAM, 1994.

[8] K. Kourtis, G. Goumas, and N. Koziris, "Optimizing sparse matrix-vector multiplication using index and value compression," in *CF '08: Proceedings of the 2008 conference on Computing frontiers*. New York, NY, USA: ACM, 2008, pp. 87–96.

[9] D. Culler and J. Singh, *Parallel computer architecture*. Morgan Kaufmann Publishers San Francisco, 1999.

[10] L. Hsu, R. Iyer, S. Makineni, S. Reinhardt, and D. Newell, "Exploring the cache design space for large scale CMPs," *ACM SIGARCH Computer Architecture News*, vol. 33, no. 4, pp. 24–33, 2005.

[11] 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.

[12] S. Toledo, "Improving the memory-system performance of sparse-matrix vector multiplication," *IBM Journal of Research and Development*, vol. 41, no. 6, pp. 711–725, 1997.

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

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

[15] 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.

[16] 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.

[17] J. Mellor-Crummey and J. Garvin, "Optimizing sparse matrix-vector product computations using unroll and jam," *International Journal of High Performance Computing Applications*, vol. 18, no. 2, p. 225, 2004.

[18] Y. Saad, "SPARSKIT: A basic tool kit for sparse matrix computations," Computer Science Department, University of Minnesota, Minneapolis, MN 55455, Tech. Rep., Jun. 1994, version 2.

[19] J. Willcock and A. Lumsdaine, "Accelerating sparse matrix computations via data compression," in *ICS '06: Proceedings of the 20th annual International Conference on Supercomputing*. New York, NY, USA: ACM Press, 2006, pp. 307–316.

[20] 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*, 15-18 Aug. 2004, pp. 169–176 vol.1.

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

[22] 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*. New York, NY, USA: ACM, 2006, p. 113.

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

[24] 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.

[25] R. Geus and S. Röllin, "Towards a fast parallel sparse matrix-vector multiplication," in *Parallel Computing: Fundamentals and Applications, International Conference ParCo*. Imperial College Press, 1999, pp. 308–315.

[26] 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*. IEEE Computer Society, 2004, pp. 66–71. [Online]. Available: http://csdl.computer.org/comp/proceedings/pdp/2004/2083/00/20830066abs.htm

[27] U. V. Catalyuerek and C. Aykanat, "Decomposing irregularly sparse matrices for parallel matrix-vector multiplication," *Lecture Notes In Computer Science*, vol. 1117, pp. 75–86, 1996.

[28] T. Davis, "University of Florida sparse matrix collection," *NA Digest*, vol. 97, no. 23, p. 7, 1997.