Exploiting Compression Opportunities to Improve SpMxV Performance on Shared Memory Systems

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

The Sparse Matrix-Vector Multiplication (SpMxV) kernel exhibits poor scaling on shared memory systems, due to the streaming nature of its data access pattern. To decrease memory contention and improve kernel performance we propose two compression schemes: CSR-DU, that targets the reduction of the matrix structural data by applying coarse-grained delta-encoding, and CSR-VI, that targets the reduction of the values using indirect indexing, applicable to matrices with a small number of unique values. Thorough experimental evaluation of the proposed methods and their combination, on two modern shared memory systems, demonstrated that they can significantly improve multithreaded SpMxV performance upon standard and state-of-the-art approaches.

Categories and Subject Descriptors: E.4 [**Data**]: Coding and Information Theory—*Data compaction and compression*; C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors); G.4 [**Mathematics of Computing**]: Mathematical Software

General Terms: Experimentation, Performance, Measurement

Additional Key Words and Phrases: sparse matrix, data compression, memory bandwidth, shared memory systems

1. INTRODUCTION

The ineffectiveness of traditional techniques, like frequency scaling and ILP exploitation, for enhancing processor performance led to a technology shift towards chip multiprocessor (CMP or multicore) designs, both in terms of commodity products and future research directions [Hennessy and Patterson 2007; Geer 2005]. As a result, a large number of research efforts target the scalability challenges that arise

Extension of Conference Paper: This paper extends the papers "Optimizing Sparse Matrix-Vector Multiplication Using Index and Value Compression" published in *Proceedings of the 2008 Conference on Computing Frontiers* (CF'08), and "Improving the Performance of Multithreaded Sparse Matrix-Vector Multiplication Using Index and Value Compression" published in *Proceedings of the 37th International Conference on Parallel Processing, 2008* (ICPP'08). The main additions with regard to prior work presented in this paper are: (a) Extension of the CSR-DU method to support units with contiguous non-zero elements. (b) Development and evaluation of a new method that combines CSR-DU and CSR-VI. (c) Evaluation of all proposed methods in a new-generation system, that implements NUMA. (d) Comparison with state-of-the-art BCSR storage format.

in applications when a large number of processing units operate on a shared memory hierarchy.

The scalability behavior of an application in a shared memory environment depends on its data access pattern. Applications with no data dependencies and good temporal locality scale well: each core can work independently using local data residing in its cache, without interfering with the operation of other cores. On the contrary, applications with streaming access patterns tend to exhibit poor scaling, due to contention on the memory subsystem.

An important and ubiquitous computational kernel with streaming memory access pattern 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). CG and GMRES are extensively used to solve sparse linear systems resulting from the simulation of physical processes described by partial differential equations [Saad 2003]. 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 [Asanovic et al. 2006].

The distinguishing characteristic of sparse matrices is that they are populated by a large number of zero elements, making it highly inefficient to perform operations using typical (dense) array structures. Special storage schemes are used instead, which target both the sparing storage of the matrix in terms of space, and the efficient execution of various operations by performing only the necessary actions. 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 representing matrix structure (*index data*), and data representing numerical values (*value data*).

Our recent work [Goumas et al. 2008], as well as related literature [Anderson et al. 1999], has identified the memory subsystem as the main performance bottleneck of the SpMxV kernel when executed in a uniprocessor environment. Obviously, if more processing units access the main memory, this 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 utilized cores increases. An approach for alleviating this problem to reduce the data accessed during the execution of the kernel (*working set*).

To this direction and using the standard CSR [Barrett et al. 1994; Saad 2003] sparse matrix storage format as a starting point, we propose two new formats: CSR-DU and CSR-VI [Kourtis et al. 2008a]. CSR-DU is a general format that reduces index data volume using coarse-grained delta-encoding for the compression of column indices. CSR-VI is a specialized format that exploits the data redundancy of matrices with a large number of common values using indirect value accesses. We also consider the combination of these two formats (CSR-DUVI), where both indices and values are compressed. The intrinsic basis of compression is to trade data storage volume for computation overhead. We argue that as the number of processing cores 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.

To evaluate our proposed methods we perform experimental tests over a large number of matrices. We focus our attention on the scalability issues that arise as core count increases. Our experimental platform consists of two systems that correspond to the ends of the commodity hardware spectrum regarding memory performance: an SMP system with centralized memory, and a new generation NUMA system, with a strong architectural focus on memory throughput performance. Our experimental results confirm that the multithreaded version of the SpMxV kernel exhibits poor scalability. We compare CSR-DU against CSR and BCSR [Im and Yelick 2001], which constitutes the most popular optimized format. CSR-DU significantly outperforms CSR and BCSR when all available cores participate in the computation. CSR-VI is able to drastically reduce the working set in a large subset of our experimental matrix suite and considerably improve performance. Finally, experimental results for CSR-DUVI demonstrate the effectiveness of the combination of index and value compression.

The rest of the paper is organized as follows: Section 2 sets the context, by providing an introduction to issues related to this work. Sections 3 and 4 present the proposed compression methods, and Section 5 discusses the experimental evaluation results. Section 6 discusses the related literature, and Section 7 concludes the paper.

2. PRELIMINARIES

2.1 Shared memory architectures

Shared memory architectures with multiple processors (SMPs) have been studied extensively in the past [Culler and Singh 1999]. The current trend of multicore processors, along with indications for many-core next-generation processors [Hsu et al. 2005], has motivated the research community to revisit the performance issues of shared memory architectures and investigate methods for scaling applications up to a large number of processing units. The major performance problem of these architectures is the main memory, which is typically centralized and shared among all processors.

It is worth mentioning that 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), instead of just the main memory. Cache sharing is an important factor of the system's performance and can be either constructive or destructive, depending on the application and on whether threads scheduled on the cores that share a cache operate on common data or not.

As core count increases, CPU designers turn to more scalable designs like Non-Uniform Memory Access (NUMA) architectures where the memory is distributed on different nodes, connected via a scalable interconnect. For a given CPU, NUMA memory nodes are characterized as either *local*, or *remote*. Accessing a local node is faster than accessing a remote node. This architecture mitigates the memory bandwidth bottleneck, since it allows different CPUs to operate on different NUMA nodes. In general, these systems provide a coherent unified view of memory, and is up to the operating system or the programmer to distribute data in different nodes to maximize performance.

2.2 Sparse matrix formats and the SpMxV operation

The most commonly used storage format for sparse matrices is the Compressed Sparse Row (CSR) format [Barrett et al. 1994; Saad 2003]. 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. An example of the CSR format for a 6×6 sparse matrix is presented in Figure 1. 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. Other generic formats for sparse matrices 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 matrix location coordinates.

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

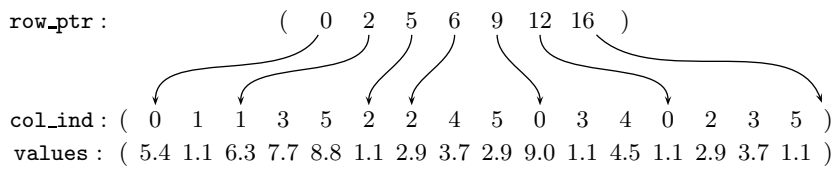


Fig. 1: Example of the CSR storage format.

The SpMxV operation ($y = Ax$), is the multiplication of a sparse matrix A and a (dense) vector x, with the result stored in another (dense) vector y. The CSR implementation for a matrix with N rows 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) consists of the matrix and vector data. Its size is expressed by the following formula where idx_s and val_s represent the storage size required for an index and a value respectively.

$$ws = \overbrace{(nnz \cdot (idx\_s + val\_s) + (nrows + 1) \cdot idx\_s)}^{\text{sparse matrix}} + \overbrace{(nrows + ncols) \cdot val\_s}^{\text{vectors}}$$

Because the majority of real-life sparse matrices satisfy: $nnz \gg nrows, ncols$, we can approximate the above expression by $ws = nnz \cdot (idx\_s + val\_s)$. Commonly, a 4-byte integer is used for index storage, due to memory size restrictions that limit

x and y vectors to a maximum of 2^{32} elements. Numerical data, however, normally require double precision. Under these assumptions, i.e., $idx\_s = 4$ and $val\_s = 8$, value data constitute the larger portion of the working set by a factor of 2/3. For this reason, value data compression is expected to have a greater impact to overall working set reduction.

The 2/3 factor is a result of memory size limitations (we assume that the sparse matrix resides in memory) and can change in the future. Specifically, matrices with dimensions larger than 2^{32}, require indices larger than 32 bits. We consider a square sparse matrix with $n = nrows = ncols = 2^{32}$ and $nnz = 100 \cdot n = 100 \cdot 2^{32}$[†]. The required CSR storage would be $100 \cdot 2^{32} \cdot (4 + 8)$ bytes ≈ 4.7 TiB. Currently, only some high-end enterprise servers contain this much memory. However, given the current rate of advancement, it is probable that near-future commodity hardware will support these capabilities.

2.3 Multithreaded SpMxV

There are several data partitioning schemes for parallelizing the SpMxV kernel on a shared memory architecture. For CSR, coarse-grained *row partitioning* is usually applied [Williams et al. 2007] where different blocks of rows are assigned to different threads (see Figure 2). Threads operate on disjoint subsets of `row_ptr`, `col_ind`, `values`, and `y` arrays. The only sharing occurs in `x` array data, but it does not constitute a performance problem. Access to `x` is read only, allowing efficient data caching over all processors. One could argue that the common use of `x` offers potential for constructive cache sharing. In practice, however, this potential is not realized, because shared data constitute a small part of the working set and cache space is limited.

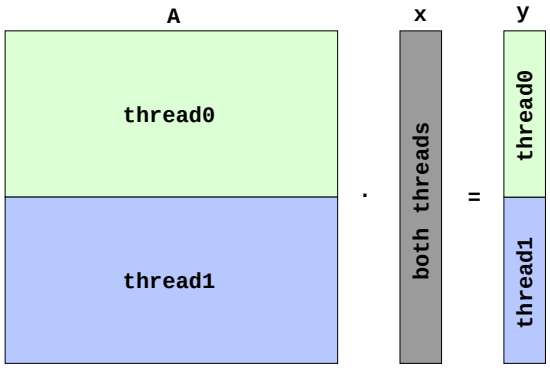


Fig. 2: Row partitioning on SpMxVfor two threads.

The complementary approach to row partitioning is *column partitioning* where each thread is assigned a block of columns. Although this approach is more naturally applied to the CSC format, it can also be applied to CSR. An advantage

[†]The number 100 has been chosen because it is close to the average value of nnz/n for our matrix suite (see Table II)

of column partitioning is that each thread operates on a different part of the `x` vector, which leads to better temporal locality for its elements in case of distinct caches. A disadvantage, however, is the possibility of cache-line ping-pongs, since each thread performs updates over all `y` elements. Having each thread use its own `y` array eliminates this problem. The final result can be obtained by adding the partial `y` arrays. Additionally, applying column partitioning to CSR may result in performance degradation, due to loop overheads caused by empty or small rows.

Block partitioning is the combination of the two aforementioned schemes where each thread is assigned an arbitrary two-dimensional block. It has the benefit of allowing configurable data sizes for each thread. For this reason, it is applied when the available memory space is limited (e.g., in the Cell processor [Gschwind et al. 2006]). Although we consider only row partitioning in our experimental evaluation, we argue that the proposed formats will result in similar performance improvements for other partitioning schemes as well.

Another issue of SpMxV parallelization is balancing each thread's workload. We apply a static balancing scheme based on the non-zero elements. Each thread is assigned approximately the same number of elements and thus the same number of floating-point operations.

3. INDEX COMPRESSION

3.1 Motivation and general approach

Sparse storage formats traditionally try to exploit contiguous elements, either in one (Figure 3a) or two dimensions (Figure 3b). Examples include the BCSR format [Saad 1994; Im and Yelick 2001], and the variable length one-dimensional block format described in [Pinar and Heath 1999]. BCSR can be viewed as a generalization of CSR where the granularity unit is an $r \times c$ dense block. The effect to overall matrix size when converting from CSR to BCSR depends on the aptitude of the selected block shape to capture the matrix structure. If resulting blocks contain a small number of zeroes, significant index reduction is achieved. For example, perfect blocking — i.e., none of the BCSR blocks contain zeroes — leads to an index reduction by a factor of $r \cdot c$. On the contrary, zeroes included in BCSR blocks must be explicitly added to value data, because all BCSR blocks are stored in a dense form. This, depending on the matrix structure and selected block shape, may lead to overall matrix size increase.

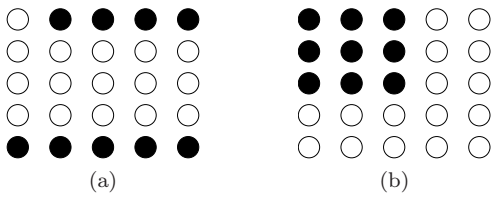


Fig. 3: Sparse matrix patterns: (a) sequential elements, (b) two-dimensional blocks.

Our approach is based on the general premise that sparse matrices have dense areas that do not necessarily contain contiguous non-zero elements (i.e., areas where

elements are close but not sequential). These areas can contribute significantly to index data size reduction when delta encoding is used to reveal the highly redundant nature of the `col_ind` array [Willcock and Lumsdaine 2006]. In a delta encoding scheme the column indices are replaced with *deltas*, each of which is defined as the difference of the current index with the previous one. Since delta values are positive and less or equal than their corresponding column indices, they can be stored in smaller size integers, leading to index data size reduction.

Instead of encoding each delta value to use only the necessary number of bytes, we propose a coarse-grained approach where 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 to implement decoding. Misprediction of these branches in execution time leads to significant performance degradation.

An important factor for the performance of this method lies in the choice of the unit size. If the size is too small, the decompression overhead introduced will dominate the performance gain from the compression. On the contrary, if the unit size is large, there will be less opportunities for compression, because a single large delta value will enforce big storage requirements for the whole unit.

This approach demonstrates a more abstract optimization strategy for the SpMxV kernel, used to exploit matrix-specific structure information. To this direction the concept of *units* could be extended to support more types of regularities, thus providing a number of advantages: (a) It can be used to exploit local regularities in specific areas of the matrix, (b) It operates on a coarse-grained level and thus 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 compression procedure will not exceed the available resources (e.g., time or storage). In Section 3.4 we discuss a method for exploiting sequential elements.

3.2 The CSR-DU storage format

The CSR-DU (CSR with Delta Units) storage format divides index data into units which are stored in a single byte-array called `ctl`. Each unit is limited to elements of a single row and consists of two parts. First, the *header* where the unit's properties are stored. Second, the *main body* where the delta-encoded column indices are stored. The header, in its simplest form, consists of two one-byte fields: (a) `usize`, the number of elements the unit contains and (b) `uflags`, a bit-vector encoding the unit's characteristics. Since `usize` is stored in a single byte, the maximum possible number of elements per unit is $2^8 = 256$. The size (1, 2, 4 or 8 bytes)[‡] of the delta values stored in the main body can be extracted from the `uflags` field, along with a marker that designates the beginning of a new row. Figure 4 presents an example of the CSR-DU format. In this example a row with 8 elements is split into two units.

[‡]8 bytes delta values are unnecessary due to hardware limitations, but supported for completeness.

The first unit has 5 elements, 1-byte delta size, and a designator for a new row (`nr`). The second unit has 3 elements, and 2-byte delta size.

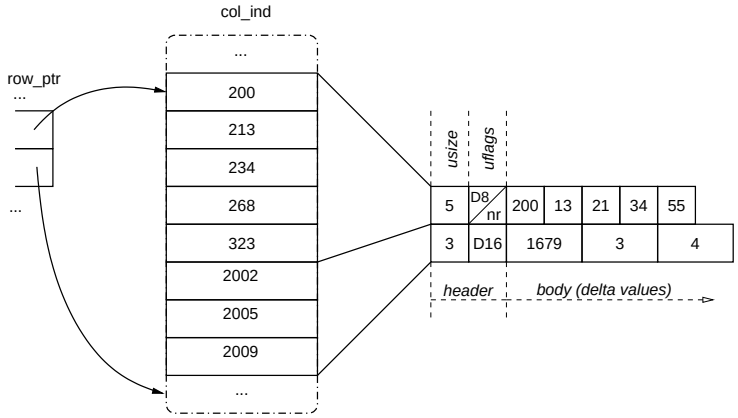


Fig. 4: Example of the CSR-DU storage format where a row is split into two units.

The compression procedure of CSR-DU is straightforward. It can be performed in $O(nnz)$ steps by scanning the matrix elements once, while 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 CSR. An important decision during this procedure is when to finalize a unit. We implemented a simple approach where a unit is finalized if (a) a new row starts in the next element, or (b) the maximum unit size is reached. A more elaborate scheme would be to finalize a unit if a new element increases the delta storage size, as long as the unit already contains more than a predetermined number of elements.

The SpMxV implementation for the CSR-DU storage format is presented in Figure 5. Access to the `ctl` array is performed via macros (e.g., `ctl_get_u16()`) that return the appropriate value and advance the array pointer as necessary. Initially, the `uflags` and `usize` header fields are extracted from the `ctl` byte-array. If the unit belongs to a new row, appropriate initializations are performed: the `y` index is increased and the `x` index is zeroed. Finally, the appropriate multiplication code is executed based on the unit type. The innermost loops of the multiplication code for the various cases do not contain any branches, which allows for fast execution by the processor.

Parallelization is similar to CSR. For example, we consider the row partitioning scheme described in Section 2.3. Each thread needs 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. The next paragraphs discuss extensions to CSR-DU format for performance improvement.

3.3 Unit offsets

A problem with CSR-DU, as described in the previous paragraphs, is that the unit's first delta value can be significantly larger than the rest, imposing an unnecessarily

```
do {
    usize = ctl_get_u8(ctl);
    uflags = ctl_get_u8(ctl);
    if ( flags_new_row(uflags) ){
       y_indx++;
       x_indx = 0;
    }
    switch ( flags_type(uflags)  ){
       case CSR_DU_U8:
       for (i=0; i<usize; i++) {
          x_indx += ctl_get_u8(ctl);
          y[y_indx] += *(values++) * x[x_indx];
       }
       break;

       case CSR_DU_U16:
       for (i=0; i<usize; i++) {
          x_indx += ctl_get_u16(ctl);
          y[y_indx] += *(values++) * x[x_indx];
       }
       break;

       case CSR_DU_U32:
       ...
    }
} while (values < values_end);
```

Fig. 5: SpMxV CSR-DU implementation.

large storage size for the rest of the deltas. In the example of Figure 4 the density of the second unit's elements allows for one-byte delta values. However, the large distance from the first unit dictates two-byte storage. To counter this problem, we modify the original CSR-DU format to include a column index offset from the previous unit in the header. The offset is called `ujmp` and is stored as a (positive) variable-length integer at the end of the header. This technique improves compression of the column indices at no cost for performance since the change does not effect innermost loops. For the implementation of the variable-length integers, we used a simple scheme where the integer's bits in its normal form are divided in 7-bits parts. These parts are stored in consecutive bytes in which the MSB is used to mark the last byte of the integer.

3.4 Sequential elements

Although delta encoding can significantly reduce index data volume, it does not handle the occurrence of sequential elements efficiently. If all unit elements are sequential column indexing information can be completely omitted. This, not only reduces the working set size, but also eliminates indirect accesses on `x` allowing for better optimization from both the compiler and the CPU. We extend CSR-DU, in a way similar to the format presented in [Pinar and Heath 1999], to support units

containing sequential elements. An example of this unit type (*sequential units*) is illustrated in Figure 6. Besides the `usize` and `uflags` fields, the unit data also contain the column index offset from the previous unit as a variable length integer. Note that if unit offsets are used, then the last field of the unit coincides with `ujmp`.

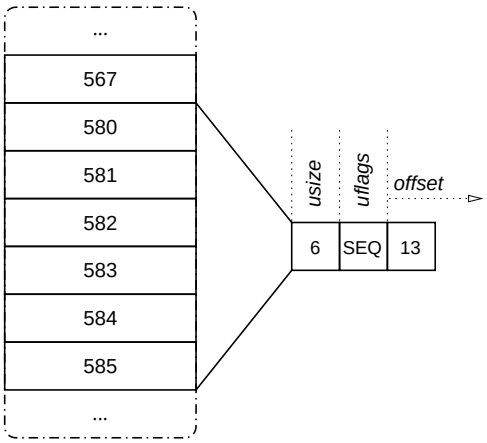


Fig. 6: Example of sequential elements unit.

An important parameter that needs to be considered during the compression phase is the minimum possible size for the sequential units. We will refer to this parameter as *seq*. Consecutive elements of size less than *seq* will be encoded using delta encoding as described in previous sections. Tuning of this parameter prevents performance degradation from sequential units with small size. For example, if *seq*=1 then all units of the matrix will be encoded as sequential. This will result in poor performance if the matrix does not contain enough sequential elements. In general, the effect of *seq* on SpMxV performance depends on: (a) the architecture of the execution platform and (b) the structure of the matrix (e.g., frequency of sequential units).

3.5 Alignment of `ctl` array values

Another issue with the CSR-DU format is that packing of delta values larger than 1 byte in the `ctl` array may lead to unaligned storage. For example in the case of Figure 4 if the first field of the `ctl` array is aligned then the three 16-bit deltas in the second unit are stored in an unaligned manner. Some ISAs disallow unaligned access. Others (e.g., the `x86` and `x86_64` ISAs) include instructions that allow unaligned access, but may result in performance degradation. In our implementation, we pad the `ucis` sections in the `ctl` array, so that the accesses of delta indices are always performed in an aligned manner.

4. VALUE COMPRESSION

4.1 Motivation and general approach

As mentioned in Section 2.2, values typically constitute the larger part of the working set of a CSR matrix because they require 64-bit storage. Hence, value compression is potentially more beneficial than index compression in terms of working set reduction. Conversely with index data, value data do not inherently contain redundancy in the general case. Moreover, the compression of floating point values is not as straightforward as integers, because floating point arithmetic operations produce rounded results.

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). From an information theory perspective this results in low entropy for the value data, making them a good target for compression. Since we aim for a computationally light decompression scheme we follow a simple approach: instead of storing all the nnz values, we store only the common values and pointers to them. If the total-to-unique values ratio is high enough, the working set data volume will be reduced. Adequate size reduction can lead to SpMxV execution time decrease, despite the overhead induced by indirectly accessing each value.

4.2 The CSR-VI storage format

The CSR-VI (CSR with Values Indirect) format replaces the CSR `values` array with two arrays: `vals_unique` and `val_ind`. The first contains only the unique matrix values. The second contains indices in the `vals_unique` array for each of the nnz matrix elements. To achieve working set size reduction, `val_ind` size must be significantly smaller than `values` size. A simple approach towards this goal is to reduce the storage requirements of individual value indices compared to the storage requirements of original values. Hence, in CSR-VI the value index size is determined by the number of the unique values that need to be addressed. For example, if there exist uv unique values and $2^8 < uv \leq 2^{16}$, then a 2-byte integer will be used for each value index. Although this approach does not optimally compresses value indices, it induces only a small overhead because it does not add any branches. An example of this value structure is presented in Figure 7, which contains the matrix values from Figure 1.

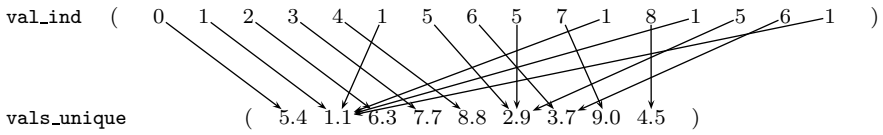


Fig. 7: Example of the value indexing structure for the CSR-VI format for the matrix presented in Figure 1.

The SpMxV kernel implementation for CSR-VI is presented in Figure 8. It can be easily derived from the CSR case by replacing the access of `values` with an indirect access of `vals_unique` based on the value of `val_ind`. Even though the resulting code contains an additional memory reference for each of the nnz elements,

it will lead to fewer data being transferred from memory when the number of unique values is relatively small. Nevertheless, working set reduction alone is not sufficient to achieve performance improvement; the additional overhead of indirect accesses must be amortized. The compression for CSR-VI is implemented using a hash table and, as in CSR-DU, its complexity is $O(nnz)$.

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

Fig. 8: SpMxV kernel for the CSR-VI storage format.

4.3 Combining CSR-VI and CSR-DU

CSR-DU and CSR-VI can be applied independently, because they operate on different data sets: CSR-DU is concerned with index data, while CSR-VI with matrix numerical values. We will refer to their combination as CSR-DUVI, a storage format that applies compression to both index and value data. Obviously, CSR-DUVI is not a general format, but can only be applied to matrices with a small number of unique values. A part of the CSR-DUVI SpMxV kernel implementation is shown in Figure 9.

```
usize = ctl_get_u8(ctl);
uflags = ctl_get_u8(ctl);
if ( flags_new_row(uflags) ){
   y_indx++;
   x_indx = 0;
}
switch ( flags_type(uflags)  ){
   case CSR_DU_U8:
   for (i=0; i<usize; i++) {
      x_indx += ctl_get_u8(ctl);
      y[y_indx] +=  vals_unique[*(val_ind++)] * x[x_indx];
   }
   break;

   case CSR_DU_U16:
   ...
}
```

Fig. 9: Code snippet for the SpMxV kernel for CSR-DUVI.

5. EXPERIMENTAL EVALUATION

5.1 Experimental setup

We conduct experiments on two systems. The first system is equipped with two quad-core Intel Harpertown processors (Figure 10). Cores operate at 2 GHz, include two private L1 32 KiB caches (instructions and data), and are grouped in pairs that share a unified 6 MiB L2 cache. The processors interface with main memory via the Intel 5000p Memory Controller Hub (MCH) which provides four channels of fully buffered DDR2 DIMM (FB-DDR2) memory.

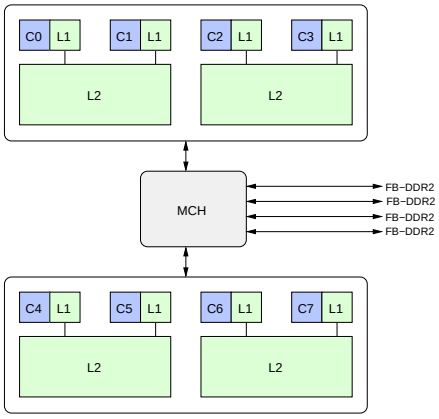


Fig. 10: An 8-core system comprising of two Harpertown processors.

In contrast with Harpertown that uses a unique interface with main memory, the second system consists of two Intel Nehalem[§] processors that implement NUMA. Each processor has four cores that operate on 2.8 GHz. Each core has private L1 (32 KiB instructions and data) and L2 (256 KiB unified) caches. Cores of the same processor share an L3 (8 MiB unified) cache. Moreover, Nehalem is equipped with an on-chip memory controller that supports three DDR3 memory channels. Communication with other memory nodes and I/O devices is implemented via QuickPath (QP) interconnect point-to-point links (Figure 11). Additionally, Nehalem cores implement Simultaneous MultiThreading (SMT), providing two different thread contexts per core.

As depicted in Figures 11 and 10, real-world systems usually employ a hierarchical topology where different core sets share different parts of the memory hierarchy. To distinguish between different scheduling configurations we will use a notation that explicitly describes the number of threads used in each level of the hierarchy. The levels are represented as:

- `t` : SMT threads on the same core (Nehalem).
- `c0` : cores that share L2 (Harpertown)
- `c1` : cores that do not share L2 (Harpertown)

[§]An initial performance evaluation of a Nehalem system can be found in [Barker et al. 2008].

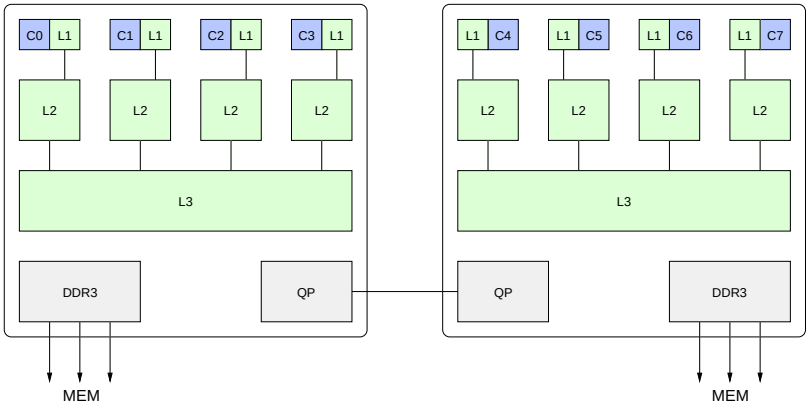


Fig. 11: An 8-core system comprising of two Nehalem processors.

`c` : cores that do not share L3 (Nehalem)

`d` : different dies (Harpertown and Nehalem)

Table I provides a concise overview of the two processors used for our experimental evaluation.

System	**Harpertown**	**Nehalem**
Model	E5405	X5560
Frequency (Ghz)	2.0	2.8
L1 (data/instruction)	32k/32k	32k/32k
L2 (unified)	6M (1/2 cores)	256k (1/core)
L3 (unified)	-	8M (1/chip)
Multithreading configuration	2c0 x 2c1 x 2d	2t x 4c x 2d

Table I: Overview of the systems used in the experimental evaluation.

For our evaluation we compiled our code with gcc 4.3.2, and performed our experiments in a 64-bit version of the Linux operating system (2.6.30). We explicitly parallelized all versions of the SpMxV kernel using the pthreads interface of the GNU libc library (NPTL 2.7). Moreover, we bound threads to specific cores using the `sched_setaffinity()` system call, and we allocated memory from specific NUMA nodes using the libnuma library (version 2.0.2).

We set the default storage size for indices and values to 32 and 64 bits respectively. Furthermore, we optimized the SpMxV code 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. We made no attempt to artificially pollute the cache after each iteration, to better simulate iterative scientific application behavior where matrix data are present in the cache hierarchy, because either they have just been produced, or they were recently accessed. Additionally, we set x to be the y vector of the previous iteration, so that our benchmark has similar behaviour with scientific methods based on SpMxV (e.g., GMRES). Setting y as x, however, restricts our matrix suite

to contain only square matrices. Finally, we applied the row partitioning scheme to all multithreaded implementations.

5.2 Memory throughput benchmark

To quantify system limits and the role of the various micro-architectural characteristics, we developed a benchmark to measure maximum throughput when different threads read data from the main memory. These measurements can be used to reveal system performance trends for applications with intensive streaming reads, such as the SpMxV kernel. The benchmark allocates and initializes large memory areas and subsequently performs read operations using streaming instructions (x86 Streaming SIMD Extensions – SSE).

Results for the Harpertown system are shown in Figure 12 that illustrates the achieved memory throughput for different scheduling configurations. As expected, scalability is poor. For example, when all available cores are used the memory throughput is increased by a factor of 1.62 relevant to the single thread scenario. This scalability problem is more intense for threads that operate on the same die: two threads in the same core achieve only a 1.12 throughput increase when compared to the serial case, while the same number for threads in different dies achieve about 1.54. Another observation from the diagram is that memory accesses from different threads may lead to performance degradation due to contention. For example, the throughput of 8 threads is less than the throughput of the `2c0×2d` configuration.

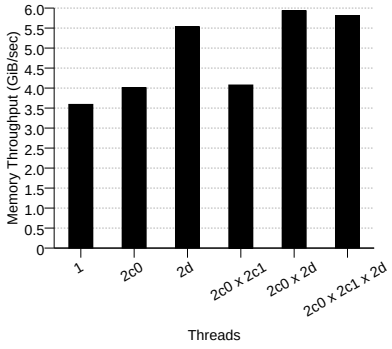
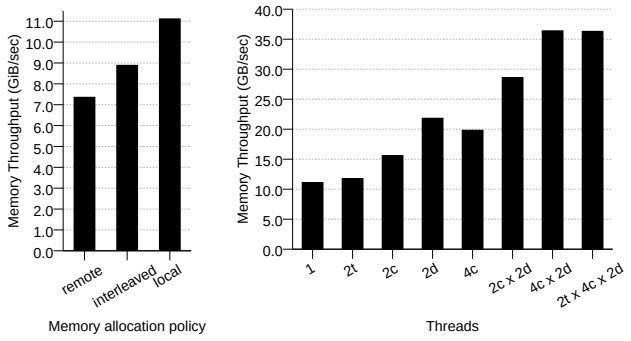


Fig. 12: Harpertown read memory throughput for different thread configurations.

The results for the Nehalem system are presented in Figure 13. Figure 13a shows the achieved memory throughput of one thread for different NUMA memory allocation policies. Local node allocation outperforms remote and interleaved policy by a significant factor (1.51 and 1.25 respectively). Figure 13b presents results for various thread configurations when using memory allocated on the local NUMA node for each thread. A single thread achieves 11.1 GiB/sec when reading from a local node, which constitutes a 3.1 improvement over Harpertown single-thread performance. Moreover, NUMA allows for good scalability when different processors are used. The speedup achieved for two threads running on different dies is — as expected — almost linear (1.96), and when all cores are utilized the speedup is 3.27. It also is

worth noting that when all cores are utilized the Nehalem processor outperforms the Harpertown processor by a factor of 6.25 (36.4 vs 5.9 GiB/sec) in this benchmark.



(a) (b)

Fig. 13: Read memory throughput for the Nehalem architecture: (a): Nehalem read memory throughput for one thread and different NUMA allocation policies. (b): Nehalem read memory throughput for different thread configurations. Local node allocation policy is used.

A comparison between these two systems shows a technology shift towards designs that focus on memory throughput performance, and indicates the importance of the memory subsystem for future multicore systems. Regarding SpMxV, we expect that the kernel will scale better in Nehalem, especially if it is assured that data are distributed among NUMA nodes so that each thread accesses local memory.

5.3 Matrix suite

Iterative use of SpMxV induces temporal locality to the application. Hence, the streaming behavior of the kernel is maintained only if the working set, and more specifically the matrix data, is significantly larger than the system's aggregate cache. For this reason, we build our matrix suite using matrices with a CSR working set larger than $4{\cdot}6 = 24$ MiB, which is the greater aggregate cache for the systems used in our experimental evaluation. The resulting matrix set consists of 50 matrices which are listed in Table II. The table also includes their characteristics, and the achieved size reduction from the proposed methods. Size reduction for the CSR-DUVI method is not included, but can be derived from the sum of the corresponding size reductions for CSR-DU and CSR-VI.

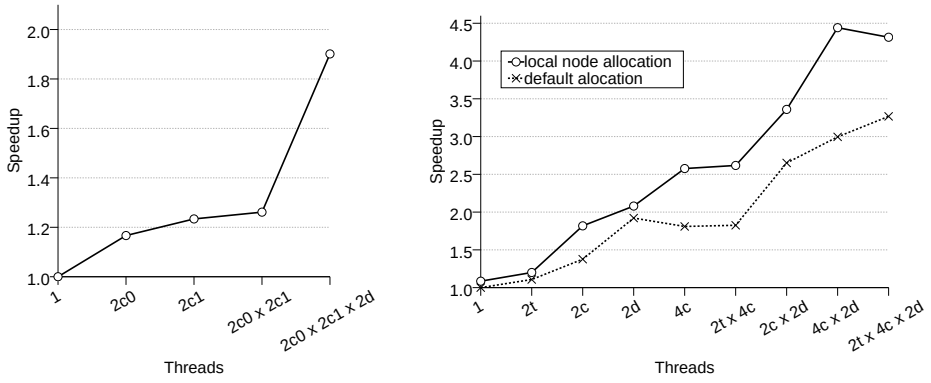
The majority of the matrices represent real-world problems and were selected from the University of Florida Sparse Matrix Collection [Davis 1997]. Our suite includes the `fdif202x202x102` matrix, which is a matrix obtained by a 5-pt finite difference problem for a $202 \times 202 \times 102$ regular grid created by SPARSKIT [Saad 1994], and two artificial matrices that represent the two ends of the sparsity spectrum: (a) a dense 2000×2000 matrix (`large-dense`) and (b) a random 100000×100000 sparse matrix (`random100000`).

matrix characteristics					size reduction (%)				
name	dim /10^3	nnz /10^6	size /1MiB	ttu	BCSR (max)	DU			VI
						noseq	seq=8	seq=4	
boneS10	914.9	55.5	638.3	1,386,710.6	22.5	16.6	27.1	30.0	58.0
ldoor	952.2	46.5	536.0	2.1	9.5	6.9	19.9	30.1	-
inline_1	503.7	36.8	423.3	2.0	22.4	4.7	15.8	22.8	-
fdif202x202x102	4,000.0	27.8	333.9	6,960,000.0	-38.5	15.9	15.9	15.9	55.7
F1	343.8	26.8	308.4	2.1	22.3	5.8	16.2	21.0	-
rajat31	4,690.0	20.3	250.4	5,098.2	-39.3	21.5	21.5	21.5	46.4
msdoor	415.9	20.2	233.2	2.1	10.2	11.9	22.3	29.1	-
Freescale1	3,428.8	18.9	229.6	2.0	-31.7	0.5	0.5	0.6	-
Ga41As41H72	268.1	18.5	212.6	5.1	-16.9	16.6	21.7	25.1	20.3
af_shell9	504.9	17.6	203.2	18.2	12.0	16.6	28.8	30.9	29.4
af_5_k101	503.6	17.6	202.8	1.9	12.1	16.5	28.8	30.9	-
TSOPF_RS_b2383	38.1	16.2	185.2	21.2	26.3	21.0	33.1	33.1	30.2
kkt_power	2,063.5	14.6	175.1	173.5	-61.2	5.2	5.2	5.2	31.5
Si41Ge41H72	185.6	15.0	172.5	3.2	-13.5	16.6	21.8	25.7	-
random100000	100.0	15.0	171.8	1,497.8	-66.3	16.7	16.7	16.7	-
nd12k	36.0	14.2	162.9	2.9	16.4	16.7	29.3	30.0	-
crankseg_2	63.8	14.1	162.2	3.2	5.8	16.8	25.8	28.7	-
pwtk	217.9	11.6	134.0	2.1	14.1	15.7	31.3	31.6	-
bmw3_2	227.4	11.3	130.1	2.7	7.4	17.5	25.9	30.0	-
ohne2	181.3	11.1	127.3	2.1	-24.8	16.7	17.8	20.0	-
hood	220.5	10.8	124.1	2.1	10.4	11.3	22.5	29.3	-
Si87H76	240.4	10.7	122.9	31.9	-29.6	16.6	20.4	22.7	31.0
bmwcra_1	148.8	10.6	122.4	3.4	22.4	16.7	25.7	28.4	-
atmosmodj	1,270.4	8.8	105.7	2,203,720.0	-38.4	16.0	16.0	16.0	55.7
thermal2	1,228.0	8.6	102.9	1.8	-42.5	12.5	12.5	13.3	-
G3_circuit	1,585.5	7.7	93.7	31,787.7	-27.1	9.3	9.3	9.3	54.6
cage13	445.3	7.5	87.3	17,936.1	-56.2	11.1	11.1	11.1	49.0
rajat30	644.0	6.2	73.1	9.0	-31.6	9.9	10.7	14.0	25.1
pre2	659.0	6.0	70.7	7.6	-38.5	14.1	14.6	14.8	23.7
Hamrle3	1,447.4	5.5	68.6	104,042.3	-6.2	17.8	17.8	19.6	53.6
largebasis	440.0	5.6	65.3	17,539.7	15.7	0.7	19.2	21.8	48.7
Chebyshev4	68.1	5.4	61.8	3.5	-28.3	20.8	26.0	26.0	-
apache2	715.2	4.8	57.9	120,446.8	-37.7	15.9	15.9	15.9	55.6
s3dkq4m2	90.4	4.8	55.5	64.9	16.7	16.6	31.8	31.8	32.1
ship_001	34.9	4.6	53.3	3.8	6.2	16.8	28.7	31.2	-
torso3	259.2	4.4	51.7	1.4	-25.4	15.3	15.3	15.3	-
thread	29.7	4.5	51.3	2.1	22.3	17.1	26.5	28.2	-
ASIC_680k	682.9	3.9	46.9	48.3	-32.6	7.5	9.1	10.9	30.2
large-dense	2.0	4.0	45.8	122.1	29.2	24.9	33.2	33.2	-
barrier2-9	115.6	3.9	45.0	3.6	-37.6	16.8	16.8	17.3	-
xenon2	157.5	3.9	44.9	41.4	19.4	21.0	21.0	21.8	31.3
parabolic_fem	525.8	3.7	44.1	14.2	-46.9	1.0	1.0	1.0	27.3
FEM_3D_thermal2	147.9	3.5	40.5	1.9	-13.7	19.3	19.3	19.3	-
sme3Dc	42.9	3.1	36.2	1.3	-58.0	16.8	16.8	16.8	-
stomach	213.4	3.0	35.4	1.3	-29.7	21.5	21.5	21.5	-
thermomech_dK	204.3	2.8	33.4	1.4	25.6	12.8	12.8	13.1	-
helm2d03	392.3	2.7	32.9	25.0	-52.1	1.4	1.4	3.8	29.3
ASIC_680ks	682.7	2.3	29.3	57.2	-31.7	17.8	20.3	22.3	44.5
poisson3Db	85.6	2.4	27.5	1.0	-60.0	17.1	17.1	17.1	-
rma10	46.8	2.4	27.3	1.9	5.3	19.1	26.1	29.4	-

Table II: Matrix suite used for the experimental evaluation. The first columns contain information about each matrix: *dim* contains the number of rows and columns of the matrix in thousands (*nrows* = *ncols*, since we consider only square matrices), *nnz* contains the number of non-zero elements in millions, *size* contains the matrix size in MiB when stored in CSR format and *ttu* contains the total-to-unique ratio for the values of each matrix. The last five columns show the size reduction achieved by various methods compared to CSR. For BCSR we selected the block shape that achieved maximum size reduction.

5.4 CSR performance evaluation

Figure 14a illustrates the average speedup of multithreaded CSR over all matrices, for different thread scheduling configurations on the Harpertown system. The speedup for 8 threads is 1.9, demonstrating the poor scalability of SpMxV. The speedup increase observed between the `2c0` and `2c1` cases — 1.17 and 1.23 respectively — can be accredited to matrix data caching during consecutive SpMxV executions. Cases `2c1` and `2c0`×`2c1` achieve roughly the same performance, even though available processors are doubled. We attribute this fact to the limited memory bandwidth since, as is shown in Figure 12, the available memory throughput for 2 and 4 cores in a single die is essentially the same.



(a) Harpertown (b) Nehalem

Fig. 14: Average speedups for the multithreaded CSR SpMxV kernel across different thread configurations. The "local node allocation" line in the Nehalem figure corresponds to a NUMA-aware version of the kernel, that binds matrix data in local NUMA node memory. The speedup for Nehalem is obtained using the single-threaded performance with default allocation, as the base performance.

A NUMA-oblivious multithreaded program can run unmodified in a NUMA system. However, there is no guarantee that data placement will be efficient. Thus, to maximize performance, we developed NUMA-aware versions of our methods, where memory allocation ensures that data accessed from a single thread are placed into the local NUMA node of the corresponding processor. Data shared between threads, e.g., the unique values array in the CSR-VI and CSR-DUVI methods, are allocated using standard mechanisms (i.e., `malloc()`).

Figure 14b presents results for the Nehalem system for two versions of the CSR SpMxV kernel: default allocation (NUMA-oblivious) and local allocation (NUMA-aware). The large memory throughput capabilities of Nehalem result in noticeably better performance than Harpertown, even for the NUMA-oblivious version. The NUMA-aware version further improves performance, achieving a 4.44 speedup for the `4c`×`2d` case. SMT threads utilization in this case, however, degrades performance (4.31).

Even though the Nehalem memory subsystem architecture drastically increases CSR SpMxV performance, it is still far from the theoretical maximum, leaving room for performance improvement by applying compression schemes. In the following sections we present only NUMA-aware versions for all methods on the Nehalem system to focus on cases that maximize performance.

5.5 CSR-DU performance evaluation

To evaluate CSR-DU, we performed experimental runs using several different combinations for the method's parameters. Versions with aligned deltas and unit offsets performed consistently better or similar than the rest, and so we present only them in the following results. Regarding sequential units, we consider three cases: absence of sequential units (*noseq*) and sequential units with a minimum of 8 (*seq*=8) and 4 (*seq*=4) elements. It should be noted that *seq*=4 performs more aggressive compression than *seq*=8. We compare CSR-DU performance against both CSR and BCSR. For the BCSR method we performed experiments with a number of different block shapes configurations*, using specialized SpMxV versions. In the following results, unless otherwise stated, we use the best performing BCSR case over all available block shape configurations.

The compression ratios achieved for each matrix are listed in Table II. The size reduction achieved for the `large-dense` matrix is the maximum possible for CSR-DU: 24.9% for noseq, and 33.2% for sequential units. The compression ratios of other matrices in our suite show a large variation, ranging from zero to close to maximum. On average, CSR-DU reduces matrix data by 14.2% for *noseq*, 19.3% for *seq*=8 and 21.1% for *seq*=4. BCSR is not able to efficiently capture the structure of matrices in our suite: it results in size increase for 28 matrices. Moreover, only for 2 matrices (`F1`, `thermomech_dK`) the best BCSR reduction is greater than reduction achieved by CSR-DU *seq*=4.

Figure 15 illustrates Harpertown CSR-DU performance results. Figure 15a shows the average speedup of CSR, BCSR, and CSR-DU over single-threaded CSR, for different thread affinity configurations. BCSR performs worse than CSR on average for all cases. The reason for that is the large number of matrices for which BCSR results in size increase. When all cores are utilized, CSR-DU methods perform better on average than CSR and BCSR. The *seq*=4 case achieves the best average speedup for 8 threads (2.45), improving performance by 28.7% and 35.0% over CSR and BCSR average, respectively. An interesting aspect of CSR-DU variants performance is that the best version for 8 threads (*seq*=4) has the lowest performance in the serial case (7% slowdown compared to CSR).

Figure 15c shows the performance improvement of BCSR and CSR-DU over CSR for each individual matrix, when 8 threads are utilized. An important observation is that the CSR-DU method does not exhibit significantly reduced performance over CSR for any matrix in our suite. For BCSR, a significant number of matrices take a performance hit compared to CSR, due to a significant size increase.

Selecting the optimal storage format is not a straightforward task, since perfor-

*the block shapes considered were: 1×2, 1×3, 1×4, 2×1, 2×2, 2×3, 2×4, 3×1, 3×2, 4×1, 4×2

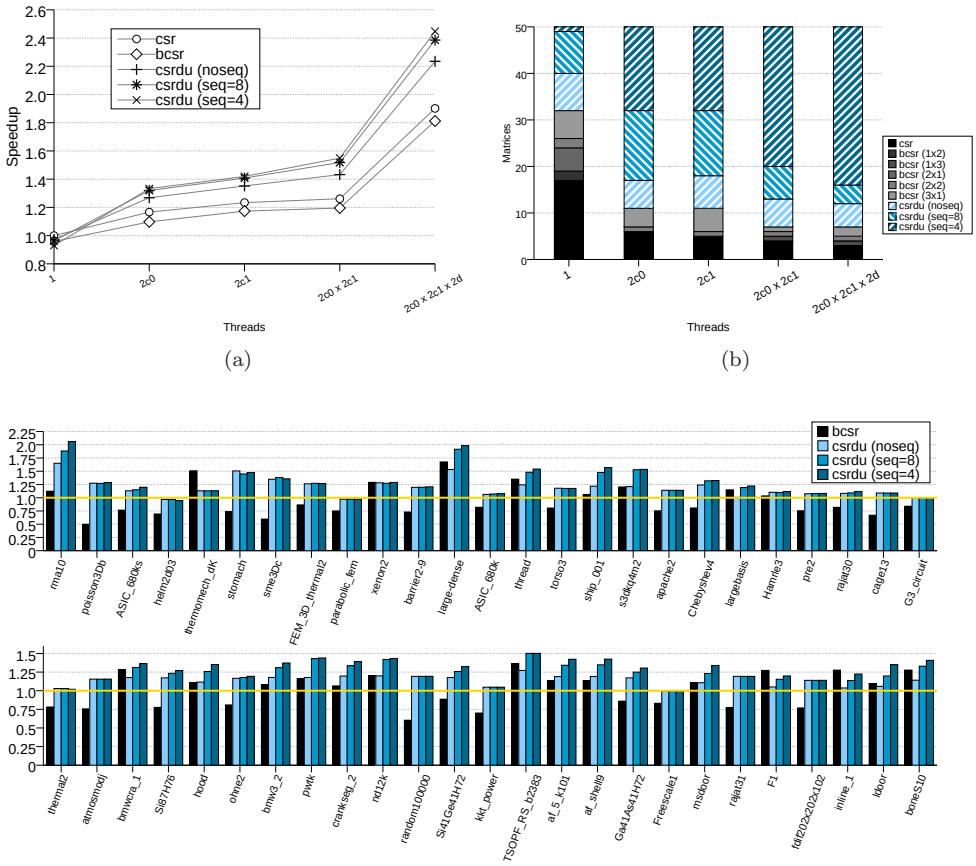
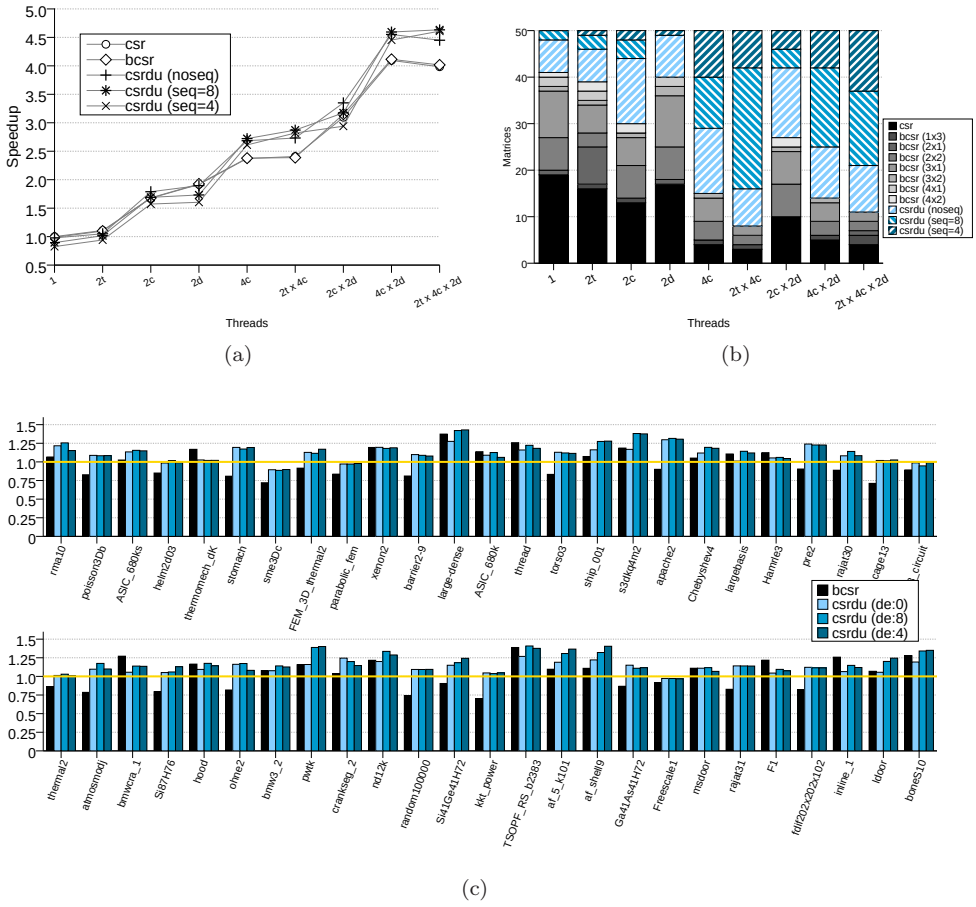


Fig. 15: CSR-DU performance results for the Harpertown system: (a) Average speedup of CSR, BSCR and CSR-DU methods over serial CSR for different thread configurations. (b) Distribution of best performing methods for different thread configurations. (c) performance improvement of BCSR and CSR-DU methods over CSR for individual matrices when all 8 cores are utilized. For (a) and (c), the BCSR performance is the best performing case for each matrix over all considered block shapes.

mance depends on the system architecture and the matrix structure in ways that are not always visible or simple. Figure 15b depicts a breakdown of the best performing methods distribution for our matrix suite. Concentrating on the full core utilization case for Harpertown, CSR-DU is a good universal choice for our matrix suite, since it achieves best performance for 43 matrices. For the same case, BCSR achieves the best performance for 4 matrices and CSR for 3. When only one thread is utilized, the best performing methods distribution is more balanced (17 for CSR, 15 for BCSR and 18 for CSR-DU). In general, we argue that as long as the main bottleneck is memory bandwidth, CSR-DU is a more promising option than BCSR and CSR. When the memory bandwidth bottleneck becomes less severe, the compu-

tation overhead imposed by decompression is not amortized, making CSR-DU less attractive.



(a) (b) (c)

Fig. 16: CSR-DU performance results for the Nehalem system: (a) Average speedup of CSR, BCSR and CSR-DU methods over serial CSR for different thread configurations. (b) Distribution of best performing methods for different thread configurations. (c) performance improvement of BCSR and CSR-DU methods over CSR for individual matrices, when all 8 cores are utilized (4c×4d). All methods considered are NUMA-aware. For (a) and (c), the BCSR performance is the best performing case for each matrix over all considered block shapes.

Nehalem results are presented in Figure 16. Figure 16a demonstrates the average speedup of considered methods over serial CSR. When all cores are utilized (4c×2d), the best average performance for CSR-DU is 4.6 — a 12.2% and 11.7% improvement over CSR and BCSR averages respectively — and is achieved by seq=8. Hence, even for the Nehalem system, where the memory bottleneck is less severe, there is a margin for performance improvement by applying CSR-DU. Interestingly, when

all SMT threads are utilized (2t×4c×2d), CSR-DU average for seq=4 and seq=8 is improved; all other methods result in a slowdown.

Figure 16c presents BCSR and CSR-DU improvement for each matrix in our suite over CSR, when all cores are utilized (4c×2d). CSR-DU performs worse than CSR for five matrices (helm2d03, sme3Dc, parabolic_fem, G3_circuit, Freescale1), although in some cases the difference is marginal. Thus, we argue that, CSR-DU is fairly stable, even when the memory bottleneck is alleviated by the system's architectural characteristics. The number of matrices that exhibit reduced performance for BCSR is 24, which again can be attributed to padding.

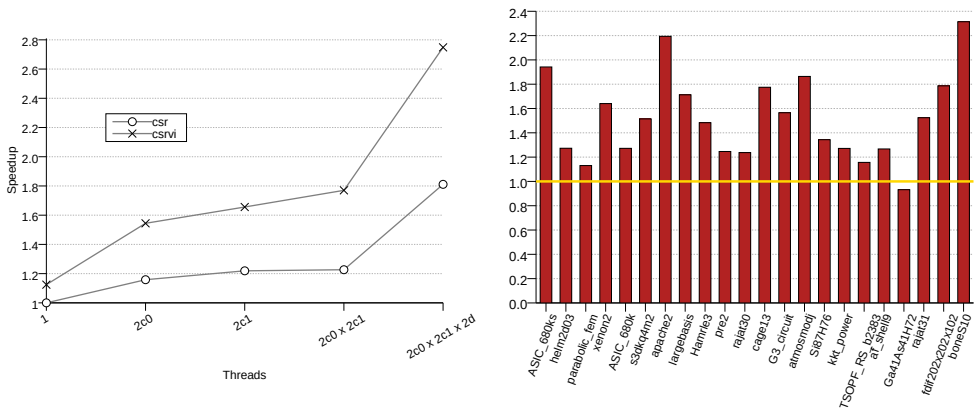
Figure 15b illustrates the distribution of best performing methods. For a single thread, CSR achieves the best performance for 19 matrices, BCSR for 22 and CSR-DU for 9. However, as the number of cores utilized increases, CSR-DU becomes the most attractive option: for 4c×2d CSR-DU performs best for 36 matrices, BCSR for 9 and CSR for 5.

5.6 CSR-VI performance evaluation

Contrary to CSR-DU, CSR-VI can be applied meaningfully only to matrices with a large number of common values. To elaborate on the applicability of the method for a given matrix, we define the *total-to-unique* (*ttu*) values ratio, as the fraction of the number of non-zero elements (*nnz*) to the number of unique values (*uvals*), i.e., $ttu = \frac{nnz}{uvals}$. A high value of *ttu* 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 our matrix suite, and discard artificial matrices random100000 and large-dense which have randomly created values. The resulting matrix suite has 22 matrices, which is a significant portion of the original set (see Table II). For these matrices, CSR-VI achieves an average matrix size reduction of 39.2%, the maximum and the minimum being 58.8% (boneS10) and 20.3% (Ga41As41H72), respectively.

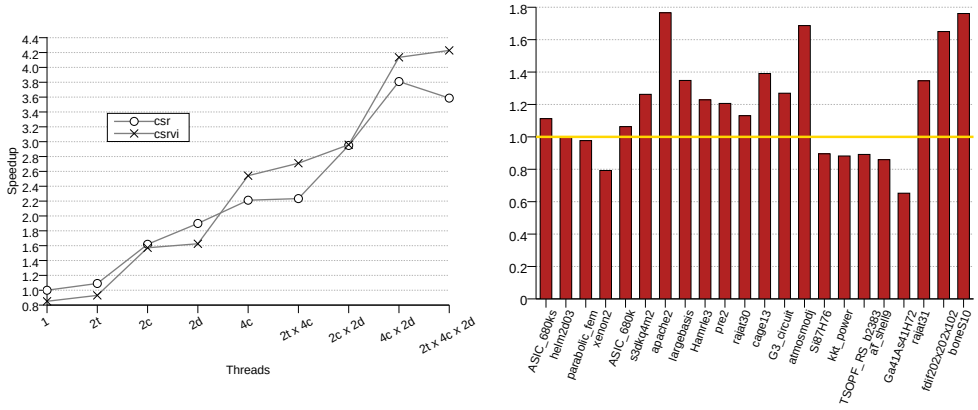
Performance results for Harpertown are shown in Figure 17. As expected, performance gains from value compression are larger than those from index compression, since we are considering 32-bit indices and 64-bit values for our reference CSR implementation. Even in the serial case, CSR-VI achieves a 12.4% performance improvement over CSR (Figure 17a). As the number of utilized cores increases, memory bandwidth bottleneck becomes more intense and working set reduction becomes more beneficial. For 8 threads the average CSR-VI speedup is 2.75, which is a 51.7% improvement over the corresponding CSR case. Figure 17b shows the performance improvement of CSR-VI over CSR for each individual matrix, when using 8 threads. CSR-VI leads to reduced performance for only Ga41As41H72, which is the matrix with the lower *ttu* value (5.1) in our suite.

CSR-VI performance for Nehalem is shown in Figure 18. The ample memory throughput capabilities of Nehalem limit the potential CSR-VI benefits. CSR is able to utilize a large portion of these capabilities due to hardware prefetching. This technique, employed by modern processors, detects easily-predicted memory access patterns (e.g., sequential) and prefetches successive data into the cache hierarchy. Contrarily, CSR-VI performs random accesses on the vals_unique array; these accesses cannot be predicted, leading to increased memory latencies.



(a) (b)

Fig. 17: Performance of the CSR-VI method on the Harpertown system: (a) average speedup of CSR and CSR-VI methods for different thread configurations. (b) performance improvement of the CSR-VI method over CSR for individual matrices, when all 8 cores are utilized.



(a) (b)

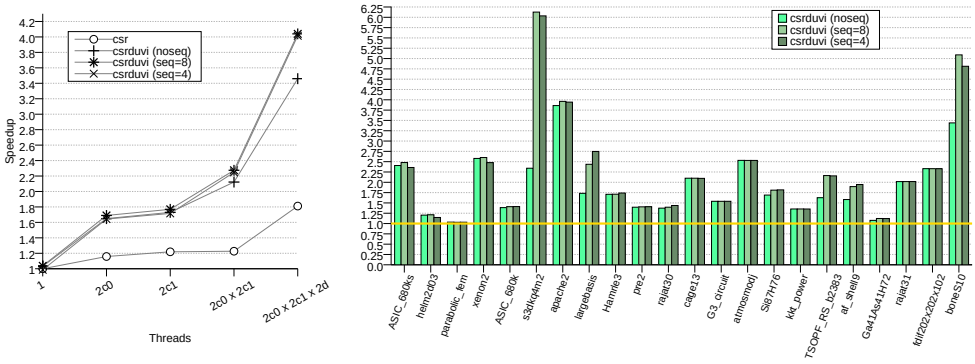
Fig. 18: Performance of the CSR-VI method on the Nehalem system: (a) average speedup of CSR and CSR-VI methods for different thread configurations. (b) performance improvement of the CSR-VI method over CSR for individual matrices, when all 8 cores are utilized. All methods considered are NUMA-aware.

As can be seen in Figure 18a, CSR-VI performs worse than CSR in the serial case (slowdown of 15%). In the `4c×2d` case, however, CSR-VI reaches a speedup of 4.13, which is a 8.6% improvement over the corresponding CSR average speedup. When all available SMT threads at each core are utilized (`2t×4c×2d`) CSR-VI average is increased to 4.23. For `4c×2d`, CSR-VI performs worse than CSR for 6 matrices

(Figure 18b).

5.7 CSR-DUVI performance evaluation

Combining CSR-VI and CSR-DU leads to an average size reduction of 52.4% for *noseq*, 56.2% for *seq*=8 and 57.3% for *seq* = 4. Experimental results for Harpertown are illustrated in Figure 19. As can bee seen in Figure 19a, CSR-DUVI serial performance is similar to CSR. In the `2c0×2c1×2d` case, however, CSR-DUVI results in a significant parallel speedup increase. More specifically, *seq*=8 achieves a speedup of 4.04, which improves upon CSR and CSR-VI by 123% and 47%, respectively. Evidently, part of this large improvement is due to matrices which now fit, in whole or in a significant portion, into L2 cache. Moreover, as can be seen in Figure 19b CSR-DUVI improves performance over CSR for all matrices.



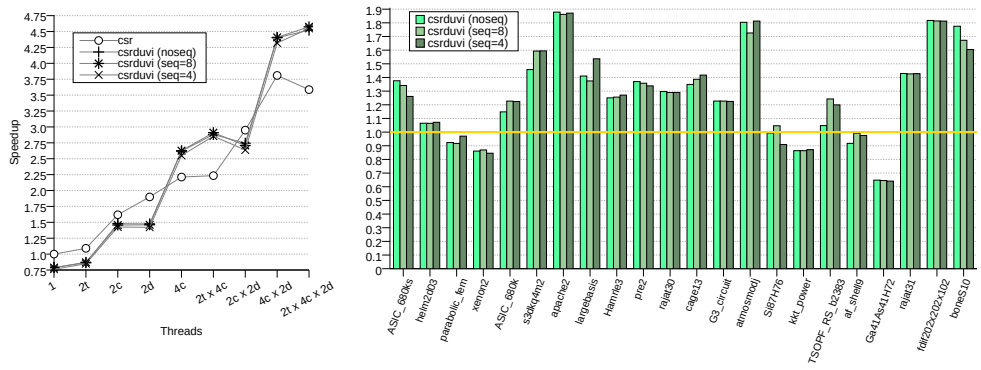
(a) (b)

Fig. 19: Performance of the CSR-DUVI method on the Harpertown system: (a) speedup of CSR and CSR-DUVI methods over serial CSR for different thread configurations. (b) performance improvement of CSR-DUVI methods over CSR for individual matrices, when all 8 cores are utilized.

Nehalem results are presented in Figure 20. The best CSR-DUVI average speedup for `4c×2d` (4.41) and `2t×4c×2d` (4.57) is achieved by *seq*=8 (Figure 20a). The respective improvements over CSR-VI are 6.6% and 8.2%, and over CSR's best performing case (`4c×2d`) 15.7% and 20%. Finally, there are 5 matrices with reduced CSR-DUVI performance over CSR (Figure 19b).

5.8 Performance evaluation summary

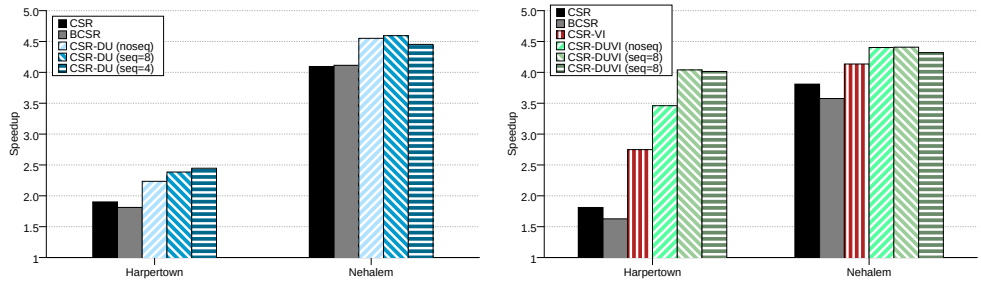
Figure 21 summarizes previous results. It illustrates the average speedup over single-threaded CSR for all methods examined in both systems, when all 8 cores are utilized. A general conclusion is that there exists a compression level, different for each architecture, beyond which the tradeoff between storage and computation becomes less attractive and can even lead to performance degradation. For the Harpertown processor, which has limited available bandwidth, this point is difficult to reach and it only becomes visible in our experiments in the CSR-DUVI method



(a) (b)

Fig. 20: Performance of the CSR-DUVI method on the Nehalem system: (a) speedup of CSR and CSR-DUVI methods over serial CSR for different thread configurations (b) performance improvement of CSR-DUVI methods over CSR for individual matrices, when all 8 cores are utilized. All methods are NUMA-aware.

where the performance of *seq*=8 is better than the *seq*=4 case. On the other hand, the architectural characteristics of the Nehalem system (NUMA, larger memory throughput capabilities) make this point easier to reach (e.g., small improvement of CSR-DUVI over CSR-VI).



(a) (b)

Fig. 21: Average speedups over the serial CSR for various methods, when all cores are utilized ($2c0 \times 2c1 \times 2d$ for Harpertown and $4c \times 2d$ for Nehalem): (a): CSR, BCSR and CSR-DU over all matrices, (b): CSR, BCSR, CSR-VI, CSR-DUVI over the 22 matrices with $ttu \geq 5$. The Nehalem versions are NUMA-aware, to take full advantage of the underlying hardware.

A question that rises, regarding the applicability of compression methods for improving SpMxV performance, is whether they will remain profitable on the face of new systems that focus on memory throughput performance. An argument in favor of such methods is that as memory throughput increases, so does the number of the cores on the system (e.g., the Nehalem architecture targets 8-core processors, which

have not yet been introduced in the market). Moreover, memory performance tends to improve with smaller rates than CPU performance (memory wall problem).

6. RELATED WORK

6.1 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 JD (Jagged Diagonal), CDS (Compressed Diagonal Storage) and Ellpack-Itpack [Barrett et al. 1994; Saad 2003]. These formats try to exploit regularities in the structure of the sparse matrix, 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 efforts [Toledo 1997; Pinar and Heath 1999; Im and Yelick 2001; Vuduc et al. 2002; Vuduc and Moon 2005] 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 [White and Sadayappan 1997; Mellor-Crummey and Garvin 2004] are concerned with the performance problems that arise in matrices with a large number of rows with small length.

6.2 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 BCSR [Saad 1994; Im and Yelick 2001] that store only per-block index information. Traditionally the main focus of BCSR has been serial performance improvement (e.g., via register blocking) and not working set reduction. For this reason, BCSR has several disadvantages if used as a compression technique. First and foremost, depending on the matrix structure, it may increase the total size of the matrix due to padding (Table II). Secondly, it relies on constant-shape blocks and thus has limited capabilities of adapting to more complex matrix structures. Pinar and Heath [1999] describe a one-dimensional variable block scheme, similar to the one used for CSR-DU sequential units. They also discuss column reordering techniques that aim to align the non-zero elements in a row in consecutive locations as much as possible. CSR-DU could benefit from similar reordering techniques towards two directions: (a) creating larger sequential units and (b) creating denser units that require smaller delta values. Although not equivalent, the latter is strongly related with matrix bandwidth reduction techniques than have been extensively studied in the past because they relate to SpMxV cache performance [Temam and Jalby 1992; Pichel et al. 2004].

One of the few works that explicitly targets the compression of the index data is [Willcock and Lumsdaine 2006]. 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, i.e., 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 compared 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 [Kourtis et al. 2008b]. Another recent work that targets performance improvement by reducing the index data volume is [Belgin et al. 2009], which proposes a matrix representation that exploits repeated block patterns. The authors search for frequently met block patterns and generate specialized inner loops for those, on top of a dispatch logic. They provide an evaluation of a parallel version, but they focus primarily on serial performance.

6.3 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. [2004] exploit matrix symmetry by storing only half the matrix, i.e., reducing matrix data volume by 50%. However, our methods can lead to larger than 50% size reduction. For example, CSR-DUVI applied to the symmetric matrix `boneS10` leads to a reduction of 88.1%. In the context of specialized hardware accelerators for SpMxV, Moloney et al. [2005] discusses compression techniques for both index and value 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 [2000], proposes the use of lower precision representation for data that do not pose problems in the convergence procedure, while Langou et al. [2006] propose mixed precision algorithms, which deliver double precision arithmetic, while performing the bulk of the work in single precision. Even though these approaches 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 and Ratanaworabhan [2007] propose a method for the efficient compression of double precision floating point values targeting network data transfers.

6.4 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 communica-

tion cost [Im and Yelick 1999; Geus and Röllin 1999; Pichel et al. 2004; Catalyuerek and Aykanat 1996]. Williams et al. [2007] present 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-bit indices are used when this is applicable.

7. CONCLUSIONS

To improve the performance of the multithreaded SpMxV kernel, by alleviating the contention on the memory subsystem, we proposed two sparse matrix storage formats: CSR-DU and CSR-VI, that apply compression to the index and value data of the matrix respectively. More specifically, CSR-DU applies a coarse-grained delta encoding compression scheme for column indices, and optionally supports dense one-dimensional blocks of variable length. CSR-VI, on the other hand, uses indirect indexing for the numerical value data, and can be meaningfully applied to matrices that exhibit a large percentage of common values. Moreover, we also considered the combination of these two formats (CSR-DUVI), that employs both the aforementioned techniques.

We performed an experimental evaluation on two different multicore systems: a typical SMP system (Harpertown), and a NUMA system (Nehalem). We used a rich matrix set, with working sets large enough to preserve the memory bound nature of the kernel. All methods demonstrated a noticeable improvement over CSR and BCSR in both systems, when all available cores were employed. The improvements were, as expected, larger on the Harpertown system, which also favors more aggressive compression techniques, due to its restricted memory bandwidth. Additionally, our proposed methods exhibited performance stability, since only a small subset of our suite resulted in a significant slowdown compared to CSR.

In conclusion, as the number of processing cores increases, simultaneous access on the shared system's memory will be the primary performance restraining factor for applications with streaming memory access patterns, such as SpMxV. Hence, we argue that our approach designates a general optimization methodology for memory intensive problems, where compression sacrifices CPU cycles to alleviate memory pressure. As we have demonstrated for the SpMxV kernel, compression can potentially lead to substantial performance improvements in multithreaded execution, even if it leads to slowdowns in the uniprocessor case. However, not all memory intensive applications are suitable for this technique. For compression to be beneficial in this context, data should be fetched from memory multiple times, so that compression overhead is amortized. Possible suitable kernels can be sought in application domains such as graph and database algorithms.

REFERENCES

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

Asanovic, K., Bodik, R., Catanzaro, B. C., Gebis, J. J., Husbands, P., Keutzer, K., Patterson, D. A., Plishker, W. L., Shalf, J., Williams, S. W., and Yelick, K. A. 2006. The landscape of parallel computing research: a view from Berkeley. Tech. Rep. UCB/EECS-2006-183, EECS Department, University of California, Berkeley. December 18.

Barker, K., Davis, K., Hoisie, A., Kerbyson, D. J., Lang, M., Pakin, S., and Sancho, J. C. 2008. A performance evaluation of the Nehalem quad-core processor for scientific computing. *Parallel Processing Letters*.

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

Belgin, M., Back, G., and Ribbens, C. J. 2009. Pattern-based sparse matrix representation for memory-efficient smvm kernels. In *ICS '09: Proceedings of the 23rd international conference on Supercomputing*. ACM, New York, NY, USA, 100–109.

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

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

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

Davis, T. 1997. University of Florida sparse matrix collection. *NA Digest 97,* 23, 7.

Geer, D. 2005. Chip makers turn to multicore processors. *IEEE Computer 38,* 5, 11–13.

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

Goumas, G., Kourtis, K., Anastopoulos, N., Karakasis, V., and Koziris, N. 2008. Performance evaluation of the sparse matrix-vector multiplication on modern architectures. *The Journal of Supercomputing*.

Gschwind, M., Hofstee, H. P., Flachs, B. K., Hopkins, M., Watanabe, Y., and Yamazaki, T. 2006. Synergistic processing in cell's multicore architecture. *IEEE Micro 26,* 2, 10–24.

Hennessy, J. and Patterson, D. 2007. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann.

Hsu, L., Iyer, R., Makineni, S., Reinhardt, S., and Newell, D. 2005. Exploring the cache design space for large scale CMPs. *ACM SIGARCH Computer Architecture News 33,* 4, 24–33.

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

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

Keyes, D. 2000. Four horizons for enhancing the performance of parallel simulations based on partial differential equations. *Euro-Par 2000 Parallel Processing: 6th International Euro-Par Conference, Munich, Germany, August/September 2000: Proceedings*.

Kourtis, K., Goumas, G., and Koziris, N. 2008a. Improving the performance of multithreaded sparse matrix-vector multiplication using index and value compression. *Parallel Processing, 2008. ICPP '08. 37th International Conference on*, 511–519.

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

Langou, J., Langou, J., Luszczek, P., Kurzak, J., Buttari, A., and Dongarra, J. 2006. 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*. ACM, New York, NY, USA, 113.

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

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

Moloney, D., Geraghty, D., McSweeney, C., and McElroy, C. 2005. Streaming sparse matrix compression/decompression. In *High Performance Embedded Architectures and Compilers, First International Conference, HiPEAC 2005, Barcelona, Spain, November 17-18, 2005, Proceedings*. 116–129.

Pichel, J. C., Heras, D. B., Cabaleiro, J. C., and Rivera, F. F. 2004. Improving the locality of the sparse matrix-vector product on shared memory multiprocessors. *Parallel, Distributed, and Network-Based Processing, Euromicro Conference on 0*, 66.

Pinar, A. and Heath, M. 1999. Improving performance of sparse matrix-vector multiplication. In *Supercomputing'99*. ACM SIGARCH and IEEE, Portland, OR.

Saad, Y. 1994. SPARSKIT: A basic tool kit for sparse matrix computations. Tech. rep., Computer Science Department, University of Minnesota, Minneapolis, MN 55455. June. Version 2.

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

Temam, O. and Jalby, W. 1992. Characterizing the behavior of sparse algorithms on caches. In *Supercomputing'92*. IEEE, Minnesota., MN, 578–587.

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

Vuduc, R., Demmel, J., Yelick, K., Kamil, S., Nishtala, R., and Lee, B. 2002. Performance optimizations and bounds for sparse matrix-vector multiply. In *Supercomputing, ACM/IEEE 2002 Conference*. 26–26.

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

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

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

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