Understanding the Performance of Sparse Matrix-Vector Multiplication

Georgios Goumas, Kornilios Kourtis, Nikos Anastopoulos, Vasileios Karakasis and Nectarios Koziris
National Technical University of Athens
Computing Systems Laboratory
School of Electrical and Computer Engineering
$\{$*goumas, kkourt, anastop, bkk, nkoziris*$\}$*@cslab.ece.ntua.gr*

Abstract

In this paper we revisit the performance issues of the widely used sparse matrix-vector multiplication (SpMxV) kernel on modern microarchitectures. Previous scientific work reports a number of different factors that may significantly reduce performance. However, the interaction of these factors with the underlying architectural characteristics is not clearly understood, a fact that may lead to misguided and thus unsuccessful attempts for optimization. In order to gain an insight on the details of SpMxV performance, we conduct a suite of experiments on a rich set of matrices for three different commodity hardware platforms. Based on our experiments we extract useful conclusions that can serve as guidelines for the subsequent optimization process of the kernel.

1. Introduction

Sparse matrix-vector multiplication (SpMxV) is one of the most important computational kernels, lying at the heart of very effective and popular iterative solution methods like CG and GMRES [15], which are used to solve sparse systems arising from the simulation of a large variety of physical, financial, social etc. problems. SpMxV is generally reported to perform poorly on modern microprocessors (e.g. 10% of peak performance [19]) mainly due to the fact that it is a memory-bound application [6]. Matrix-vector multiplication exhibits more intense memory access needs than other traditional algebra kernels like matrix multiplication (MxM) or LU decomposition, which are more computationally intensive. MxM and LU benefit from the so called *surface-to-volume* effect, since for a problem size of n they perform $O(n^3)$ operations on $O(n^2)$ amount of data. On the contrary, matrix-vector multiplication performs $O(n^2)$ operations on $O(n^2)$ amount of data, which means that the ratio of memory access to floating point operations is significantly higher. Seen from another point of view, there is little data reuse in the matrix-vector multiplication, i.e. very restricted temporal locality. When sparsity comes into play, the performance is further degraded. In order to avoid extra computation and storage overheads imposed by the large majority of the zero elements contained in the sparse matrix, the non-zero elements of the matrix are stored contiguously in memory, while additional data structures assist in the proper traversal of the matrix and vector elements. For example, the classic Compressed Storage Row (CSR) format [2] uses the `row_ptr` structure to index the start of each row within the non-zero element matrix `a`, and the `col_ind` structure to index the column each element is associated with. These additional data structures used for indexing greatly affect the kernel's performance, since they add additional memory access operations, memory bandwidth pressure and cache interference. Sparse matrices also incur irregular accesses to the input vector `x` (CSR format is assumed) that follow the sparsity pattern of the matrix. This irregularity complicates the utilization of data reuse on vector `x` and increases the number of cache misses on this vector. Finally, there is also a non-obvious implication in sparsity: the rows of the sparse matrices have varying lengths which are frequently small. This fact increases the loop overheads since a small number of computations is performed in each loop iteration.

A large number of research papers [1, 3, 5, 8–13, 16–20] have proposed optimization techniques to improve the performance of SpMxV (see Section 2 for details). A general conclusion is that SpMxV can be efficiently optimized by exploiting information regarding the matrix structure and the processor's architectural characteristics. In general, previous research focuses on a subset of the reported problems and proposes optimizations applied to a limited number of sparse matrices. This fact, along with the CPUs used in various previous works, may lead to contradictory conclusions and, perhaps, to confusion regarding the problems and

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

candidate solutions for SpMxV optimization. In addition, the exact reason for performance gain after the application of the proposed optimizations is rarely investigated. For example, blocking implemented with the use of the Block Compressed Storage Row (BCSR) format was proposed by Im and Yelick [8] as a transformation to tame irregular accesses on the input vector and exploit its inherent reuse as in dense matrix optimizations. One-dimensional blocking is also proposed by Pinar and Heath [13] in order to reduce indirect memory references, while, quite recently, Buttari et al. [3] and Vuduc and Moon [19] accentuate the merit of blocking (the latter with variably sized blocks), as a transformation to reduce indirect references and enable register level blocking and unrolling. However, it is not clarified if the benefits of blocking can be actually attributed to better cache utilization, memory access reduction or ILP improvement. Furthermore, White and Sadayappan [20] report that the lack of locality is not a crucial issue in SpMxV, whereas many important previous works exploit reuse on the input vector in order to improve performance [5, 11, 12, 17].

The goal of this paper is to assist in understanding the performance issues of SpMxV on modern microprocessors. To our knowledge, there are no experimental results concerning the performance behavior of this kernel, or any of its optimized versions, on current microarchitectures. In order to achieve this goal, we have categorized the problems of the algorithm as reported in literature. For each problem we conduct a series of experiments in order to quantify its effect on performance. Our experimental results provide valuable insight on the performance of SpMxV on modern microprocessors and reveal issues that will probably prove particularly useful in the process of optimization. Our experiments are performed on a large suite of 100 matrices selected from Tim Davis' collection [4]. Based on the conclusions drawn from the conducted experiments, we propose guidelines that can aid the optimization process.

The rest of the paper is organized as follows: Section 2 presents related work on SpMxV and optimization methods and Section 3 presents the basic algorithm and the reported problems. In Section 4 we present various experimental results that illuminate the performance issues of SpMxV, while Section 5 summarizes our conclusions and discusses future research work.

2. Related work

Sparse matrix-vector multiplication has attracted intensive scientific attention in the last two decades. The proposal of efficient storage formats for sparse matrices like CSR, BCSR, CDS, Ellpack-Itpack and JAD [2, 10, 15] was one of the primary concerns. Temam and Jalby [16] perform a thorough analysis of the cache behavior of the algorithm, pointing out the problem of the irregular access pattern in the input vector `x`. Toledo [17] deals with this problem by proposing a permutation of the matrix that favors cache reuse in the access of `x`. Furthermore, the application of blocking is also proposed in that work, in order to both exploit temporal locality on `x` and reduce the need for indirect indexing through `col_ind`. Software prefetching for `a` and `col_ind` is also used to improve memory access performance. The proposed techniques were evaluated over 13 sparse matrices on a Power2 processor and achieved a significant performance gain for the majority of them. White and Sadayappan [20] state that data locality is not the most crucial issue in sparse matrix-vector multiply. Instead, small line lengths, which are frequently encountered in sparse matrices, may drastically degrade performance due to the reduction of ILP. For this reason, the authors propose alternative storage schemes that enable unrolling. Their experimental results exhibited performance gains on an HP PA-RISC processor for all 10 sparse matrices used. Pinar and Heath [13] refer to irregular and indirect accesses on `x` as the main factors responsible for performance degradation. Focusing on indirect accesses, the application of one-dimensional blocking with the BCSR storage format is proposed, in order to drastically reduce the number of indirect memory references. In addition, a column reordering technique is also proposed, which enables the construction of larger dense sub-blocks. An average 1.21 speedup is reported for 11 matrices on a Sun UltraSPARC II processor.

With a primary goal to exploit reuse on vector `x`, Im and Yelick propose the application of register blocking, cache blocking and reordering [7, 8]. Additionally, their blocked versions of the algorithm are capable of reducing loop overheads and indirect referencing, while increasing the degree of ILP. Register blocking is the most promising of the above techniques. The authors also propose a heuristic to determine an efficient block size. They perform their experiments on four different processors (UltraSPARC I, MIPS 10000, Alpha 21164, PowerPC604e) for a wide matrix suite involving 46 matrices. For almost a quarter of these matrices register blocking achieved significant performance benefits. Geus and Röllin [5] apply software pipelining to increase ILP, register blocking to reduce indirect references and matrix reordering to exploit the reuse on `x`. They perform a set of experiments on a variety of processors (Pentium III, UltraSPARC, Alpha 21164, PA-8000, PA 8500, Power2, i860 XP) and report significant performance gains on two matrices originating from the discretization of $3-D$ Maxwell's Equations with FEM. Vuduc et al. [18] estimate the performance bounds of the algorithm and evaluate the register blocked code in respect to these bounds. Furthermore, they propose a new approach to select near-optimal register block sizes. Mellor-Crummey and Garvin [9] accentuate the problem of short row lengths and propose the application of the well-known unroll-and-jam compiler optimization in order to deal with it. Unroll-and-jam achieves a 1.11–2.3 speedup for two matrices taken from the SAGE package measured on MIPS R12000, Alpha 21264A, Power3-II and Itanium processors. Pichel et al. [11] model the inherent locality of a specific matrix with the use of distance functions and improve this locality by applying reordering to the original matrix. The same group proposes also the use of register blocking to further increase performance [12]. The authors report an average of 15% improvement for 15 sparse matrices on MIPS R10000, UltraSPARC II, UltraSPARC III and Pentium III processors.

Buttari et al. [3] provide a performance model for the blocked version of the algorithm based on BCSR format, and propose a method to select dense blocks efficiently. Their experimental results are performed on K6, Power3 and Itanium II processors for a suite of 20 sparse matrices and validate the accuracy of the proposed performance model. Vuduc et al. [19] extend the notion of blocking in order to exploit variable block shapes and, in order to achieve this, decompose the original matrix to a proper sum of submatrices, having each submatrix stored in the BCSR format. Their approach is tested on the Ultra2i, Pentium III-M, Power4 and Itanium II processors for a suite of 10 FEM matrices that contain dense sub-blocks. The proposed method achieves better performance than pure BCSR in all processors except Itanium II. Finally, Willcock and Lumsdaine [21] mitigate the memory bandwidth pressure, by providing an approach to compress the indexing structure of the sparse matrix, sacrificing in this way some CPU cycles. They perform their experiments on a PowerPC 970 and an Opteron processor for 20 matrices, achieving an average of 15% speedup.

Summarizing on the results of previous research on the field, the following conclusions may be drawn: (a) the matrix suites used in the experimental evaluations are usually quite small, (b) the evaluation platforms include in most cases previous generation microarchitectures, (c) the conclusions are sometimes contradictory and (d) the performance gains attained by the proposed methods are not thoroughly analyzed in relevance to the specific problems attacked. The goal of this work is to understand the performance issues of SpMxV kernel on modern microprocessors and provide solid optimization guidelines. For this reason we employ a wide suite of 100 matrices, perform a large variety of experiments and report performance data and information collected from the performance monitoring facilities provided by the microprocessors.

3. Basic algorithm and problems

The most frequently applied storage format for sparse matrices is the Compressed Storage Row (CSR) [2]. According to this format, the nnz non-zero elements of a sparse matrix with n rows are stored contiguously in memory in row-major order. The `col_ind` array of size nnz stores the column of each element in the original matrix, and the `row_ptr` array of size $n+1$ stores the beginning of each row. Fig. 1 shows an example of the CSR format for a sparse 6×6 matrix (a), along with the implementation of the matrix-vector multiplication for a dense $N \times M$ matrix (b) and the matrix-vector multiplication for a sparse matrix stored in CSR (c).

According to literature, SpMxV presents the following problems that can potentially affect its performance:

- *Memory intensity (no temporal locality in the matrix)* [3, 6, 9]. This is an inherent problem in the algorithm, regardless the matrix is sparse or dense. Unlike other important numerical codes like matrix multiplication (MxM) or LU decomposition, the kernel is memory bound, while the elements of the matrix in matrix-vector multiply are used only once.

- *Indirect memory references* [13]. This is the most apparent implication of sparsity. In order to save memory space and reduce floating-point operations, only the non-zero elements of the matrix are stored. To achieve this, the indices to the matrix elements need to be stored and accessed from memory, via the `col_ind` and `row_ptr` data structures. This fact implies additional load operations, traffic for the memory subsystem and cache interference.

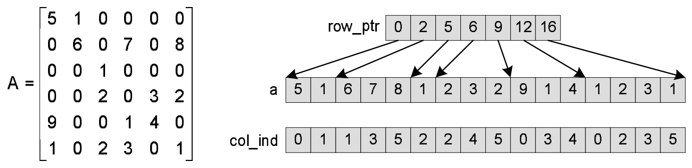
- *Irregular memory accesses for vector* `x` [5, 7, 11]. Unlike the case of dense matrices where the access to the vector `x` is sequential, in sparse matrices this access is irregular and dependent on the sparsity structure of the matrix. This fact complicates the process of exploiting the inherent reuse in the access of vector `x`.

- *Short row lengths* [3, 9, 20]: Although this problem is not obvious, it is very often met in practice. Many sparse matrices exhibit a large number of rows of short length. This fact may degrade performance due to the significant overhead of the loops, when the trip count of the inner loop is small.

4. Experimental evaluation

4.1. Experimental preliminaries

Our experiments were performed on a 100 matrices set (see Table 4). The majority of them was selected from Tim Davis' collection [4]. The first matrix is a dense 1000×1000 matrix, matrices 2-45 are also used in SPARSITY [7], matrix #46 is a 10000×10000 random, sparse matrix, matrix #87 is a 5-pt stencil, finite-difference matrix created



(a) CSR Storage format.

```
for (i=0; i<N; i++)
  for (j=0, l=i*M; j<M; j++)
    y[i] += a[l+j]*x[j];
```

(b) Dense Matrix

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

(c) Sparse Matrix

Figure 1. Example of the CSR storage format, dense and sparse matrix-vector multiplication kernels.

by SPARSKIT [14], while the rest are the largest rectangular matrices of the collection both in terms of non-zero elements and number of rows. All matrices are stored in CSR format.

The experimental platform includes three different microproccessors: an Intel Core 2 Xeon (Clockspeed: 2.6GHz, 4MB L2 cache –*Woodcrest*), an Intel Pentium 4 Xeon (Clockspeed: 2.8GHz, 1MB L2 cache –*Netburst*) and an AMD Opteron (Clockspeed: 1.8GHz, 1MB L2 cache –*Opteron*). These processors are a representative set of commodity processors currently available. The systems run Linux (kernel version 2.6) for the x86_64 ISA, while the programs were compiled using `gcc` version 4.1 with the `-O3 -funroll-loops` optimization flags. The latter switch causes the compiler to apply loop unrolling to all loops of the program.

The experiments were conducted by measuring the execution time of 128 consecutive SpMxV operations with randomly created `x` vectors for every matrix in the set and for each different microprocessor. The floating point operations per second (FLOPS) metric of each run was calculated by dividing the total number of operations ($2 \times nnz$) by the execution time. We used 64-bit integers for the representation of indices in `col_ind` and applied double precision arithmetic. 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, either because they have just been produced, or because they were recently accessed.

One of the most prominent characteristics of modern processors is *hardware prefetching*. Hardware prefetching is a technique to mitigate the ever-growing memory wall problem by hiding memory latency. It is based on a simple hardware predictor that detects reference patterns (e.g. serialized accesses) and transparently prefetches cachelines from main memory to the CPU cache hierarchy. In order to gain a better insight on the performance issues involved, we conducted experimental tests to evaluate the effect of hardware prefetching in the SpMxV kernel by disabling it. We present results for Intel processors only since there does not seem to be a (documented) way to disable hardware prefetching for AMD processors. A summary of the results obtained is presented in Table 1.

Processor	*matrices with speedup $> 10\%$*	*average speedup*	*max speedup*
Woodcrest	84	1.90	2.27
Netburst	93	2.29	2.81

Table 1. Performance impact of hardware prefetching on SpMxV kernel for Intel processors.

4.2 Experimental evaluation of serial SpMxV

4.2.1 Basic performance of serial SpMxV

Fig. 2 shows the detailed performance results for the SpMxV kernel in terms of FLOPS for each matrix and architecture on the experimental set. To gain a better understanding of the results, we consider the benchmark of a Dense Matrix-Vector (DMxV) Multiplication, for a dense matrix 1024×1024, as an upper bound for the peak performance of the SpMxV kernel. Summarized results are presented in Table 2. As expected, the more recent Woodcrest processor outperforms the other two in the whole matrix set. Moreover, while Netburst and Opteron exhibit similar behavior for each matrix, Woodcrest in some cases deviates greatly. This is apparent, for example, in #14, #16 and #54 matrices, where the performance for the Woodcrest increases by a large factor. This is, most probably, due to its larger L2 cache. Furthermore, it is clear from

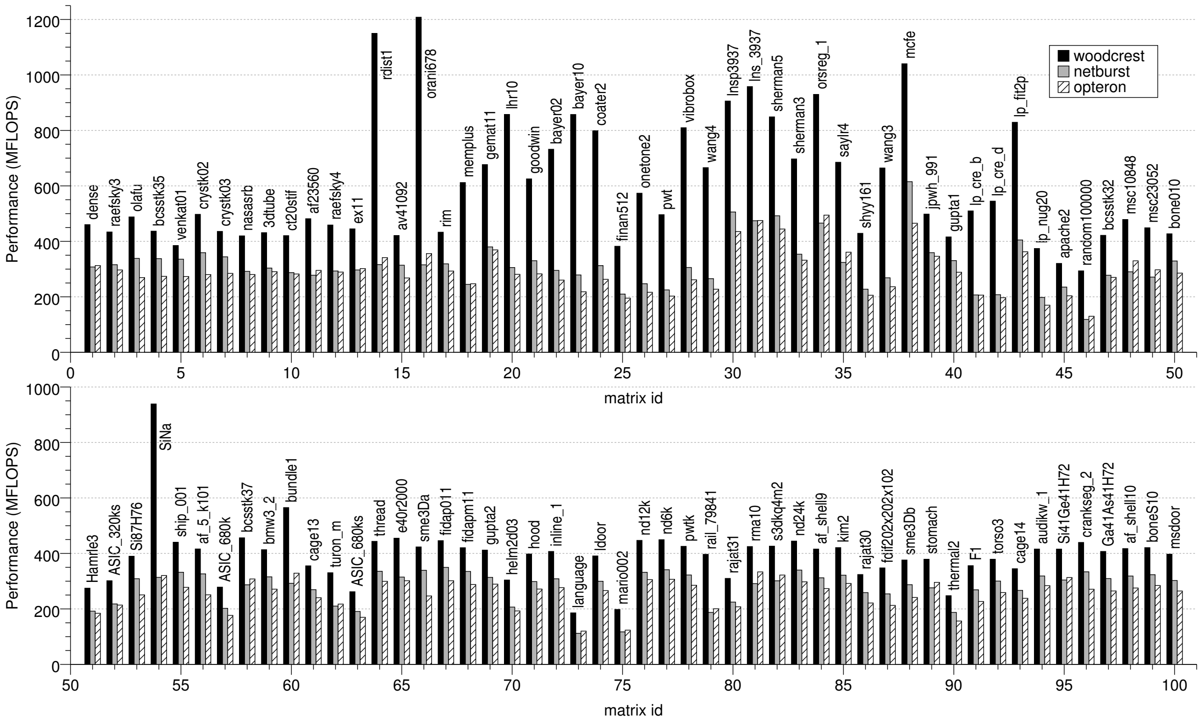


Figure 2. Performance of the SpMxV kernel: MFLOPS for each matrix and architecture.

Fig. 2 that the performance across the matrix set has great diversity. In order to further elaborate on this observation, we make a distinction between two different classes in the matrix set: the matrices whose working set fits perfectly into L2 cache, and thus experience only compulsory misses, and the matrices whose working set is larger than the L2 cache size and may experience capacity misses. The formula for the calculation of the working set (ws) in bytes is: $ws = (nnz \times 2 + nrows \times 2 + ncols) \times 8$. In Fig. 3 we present the performance attained by each matrix, with its working set marked on the x axis. The vertical line in each graph designates the size of L2 cache for each architecture. This figure clarifies that the great differences between the performance of various matrices are due to the size of their working sets. If the working set of a matrix fits in the cache, then significantly higher performance is expected. It is evident that the performance issues involved for each category are different and comparing the performance of matrices from different classes may lead to false conclusions.

Additionally, Fig. 4 presents the performance of each matrix with respect to the L2 cache miss-rate as measured from the performance counters for each processor. As anticipated, working sets that are smaller than the cache size exhibit close to zero L2 miss-rate. At a coarser level, there seems to be a correlation between the performance in FLOPS and L2 misses. Regardless, the L2 miss-rate metric does not suffice alone to understand the performance of the kernel. For example, there are cases where a great increase in the miss-rate does not have an equivalent effect on performance, whereas matrices with similar miss-rates have significantly varying MFLOPS.

Processor	max	min	average	DMxV
Woodcrest	1208.07	185.73	495.53	790.66
Netburst	615.15	112.15	297.88	658.82
Opteron	494.51	119.97	273.72	507.49

Table 2. Summarized results (MFLOPS) for the performance of the SpMxV kernel.

4.2.2 Irregular accesses

In order to evaluate the performance impact of irregular accesses on `x`, we have developed a benchmark, henceforth called *noxmiss*, which tries to eliminate cache misses on vector `x`. More precisely, *noxmiss* zeroes out the `col_ind` array, so that each reference to `x` accesses only `x[0]`, resulting in an almost perfect access pattern on `x`. Note that the *noxmiss* version of the algorithm differs from the *standard* one only in the values of the data included in the `col_ind` array, and thus executes exactly the same opera-

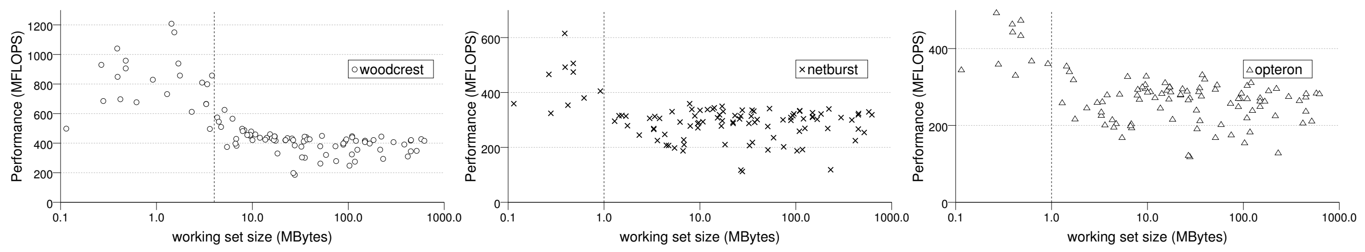


Figure 3. Performance of the SpMxV kernel in relation to the working set size for all architectures. The vertical line in each graph designates the size of the L2 cache for each architecture.

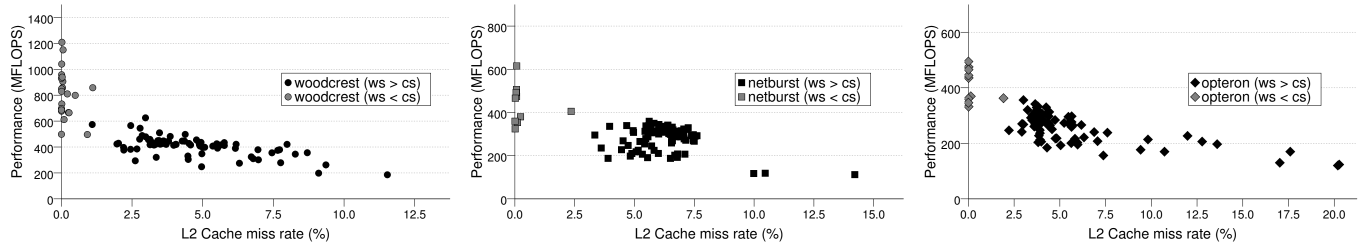


Figure 4. Performance of the SpMxV kernel in relation to the L2 cache miss-rate as reported from the performance counters.

tions. Obviously, its calculations are incorrect, but it is quite safe to assume that any performance deviation observed between the two versions is due to the effect of irregular accesses on the input vector x. Results of the experiments for the *noxmiss* are presented in Table 3.

Processor	*Speedup*		*Matrices with speedup:*		
	average	*max*	$> 10\%$	$> 20\%$	$> 30\%$
Woodcrest	1.27	1.74	28	15	11
Netburst	1.33	2.91	26	13	6
Opteron	1.28	2.37	32	16	10

Table 3. Summarized results for the *noxmiss* benchmark: Speedup and number of matrices that encountered a minimum performance gain of 10%, 20% and 30%.

It is worth noticing that only a small percentage of the matrices (no more than $1/3$ of the total matrix set), did encounter a significant amount of performance speedup of over 10% for all processors. This means that the irregular access pattern of SpMxV is not the prevailing performance problem. For the large majority of matrices it seems that the access on x presents some regularity that either favors data reuse from the caches, or exhibits patterns that can be detected by the hardware prefetching mechanisms. However, the majority of matrices that performed rather poorly on the *standard* benchmark, encountered quite significant speedup on the *noxmiss* benchmark. This leads to the conclusion that there exists a subset of matrices, where the irregular accesses on x pose a considerable impediment on performance. These matrices have a rather irregular non-zero element pattern, which finally leads to poor access and low reuse on x and tends to degrade performance.

4.2.3 Short row lengths

Short row lengths that are frequently met in sparse matrices lead to a small trip count in the inner loop, a fact that may degrade performance due to the increased overhead of the loops. In order to evaluate the impact of short row lengths on the performance of SpMxV, we focus on matrices that include a large percentage of short rows. Fig. 5 shows the performance of matrices in which more than 80% of the rows contain less than eight elements. The x axis sorts these matrices by their *ws*. The vertical line represents the cache size of each processor and the horizontal line represents the average performance across all matrices (see Table 2). The obvious conclusion that can be drawn from Fig. 5 is that matrices with large working sets and many short rows exhibit performance significantly lower than the average. This performance degradation could be attributed to the loop overhead. However, the fact that matrices with many short rows and small working sets achieve remarkably good performance, provides a hint that loop overhead should not be the

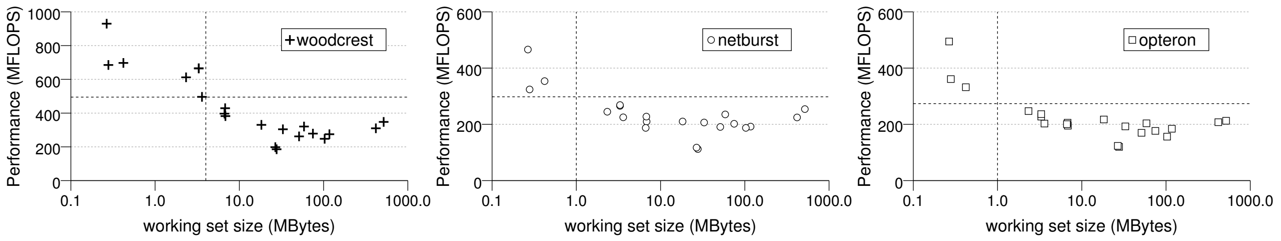


Figure 5. Performance of matrices with small number of elements (< 8) for 80% or more of their rows, in relation to their working set. Vertical line marks the L2 cache size for each processor. Horizontal line marks the average performance.

only factor. Another important observation that supports the above point is that the matrices reported in Fig. 5 coincide (with few exceptions) with the matrices that benefited when the *noxmiss* benchmark was applied to them. These facts guide us to the conclusion that short row lengths may indicate a large number of cache misses for the x vector. This can be explained by the fact that short row lengths increase the possibility to access completely different elements of x in subsequent rows.

4.2.4 Indirect memory references

There are two indirect memory accesses in the SpMxV kernel: one in row_ptr to determine the bounds of the inner loop and one for the x access (col_ind). To investigate the effect of the indirect memory references in the performance of the kernel we used synthetic matrices with a constant number of contiguous elements per row. These matrices enable us to eliminate both cases of indirect accesses by replacing them with sequential ones (*noind-rowptr*, *noind-colind*). Next, we compare the performance of the new versions with *standard* in order to attain a qualitative view on the performance impact of the indirect references. We applied the original SpMxV kernel and the modified versions on a number of synthetic matrices with $1,048,576$ elements and varying row length.

Fig. 6 summarizes the performance measured for a subset of the row lengths applied. The performance does not significantly deviate for different row lengths. It is clear that the indirect memory references in row_ptr do not affect performance. This is quite predictable since these references are rare and replace an already existing overhead in the inner loop initialization. On the other hand, the overhead in the indirect access of x through col_ind leads to a dramatic degradation in performance. In this case each memory access of x is burdened by one additional memory access which increases the ws of the problem, adds extra instructions in the code and limits the IPC of the kernel.

4.2.5 Memory intensity (no temporal locality in the matrix)

The intense memory requirements and the lack of temporal locality are two issues strongly related. In SpMxV each matrix element participates in only two FP operations. This fact increases the memory to FP operations ratio and significantly affects SpMxV's performance. Thus, the performance of the kernel is not determined by the processor speed, but by the ability of the memory subsystem to provide data to the CPU [6]. In order to further illuminate this feature of the kernel, we performed a simple, comparative set of experiments: we used 32-bit instead of 64-bit integers for the col_ind structure, in order to reduce the total size of the working set. *Woodcrest* exhibited a 1.20, *Netburst* a 1.29 and *Opteron* 1.17 speedup. It is quite impressive that the decrease of the ws by a factor of 22.4% led to a significant increase in performance. Based on these results, we can conclude that the improvement observed in the case of indirect memory accesses (see Fig. 6) is mainly due to the reduction of the ws. The same applies for the benefits of the BCSR format [3, 8, 19], which uses one index for each dense sub-block saving in this way valuable memory space in col_ind.

Furthermore, the kernel exhibits no reuse in the data structures a, col_ind and row_ptr that represent the matrix. The lack of temporal locality is traditionally believed to affect performance. However, as seen in Fig. 1, all aforementioned structures are accessed in a very regular, streaming pattern with unit stride. The hardware prefetcher is able to detect these simple access patterns and transparently fetch their corresponding cache-lines from memory (see Section 4.1 for experimental information on the effect of hardware prefetching). Thus, it is quite safe to conclude that the lack of temporal locality in the matrix causes an insignificant number of cache misses and therefore performance is not affected by this particular factor.

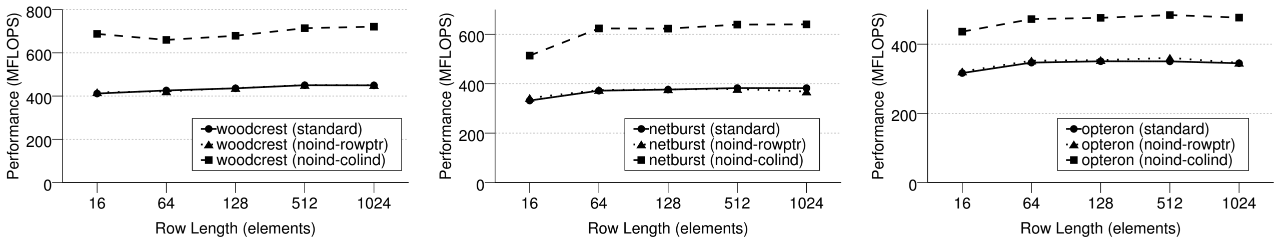


Figure 6. Performance of the *standard*, *noind-rowptr*, *noind-colind* versions for different number of elements per row.

4.3 Conclusions on the experiments and optimization guidelines

Based on the experimental evaluation of the previous sections, a number of interesting conclusions can be drawn. At first, the performance of the kernel is greatly affected by the matrix working set. As shown in Fig. 3, matrices with working sets that entirely fit in the L2 cache exhibit a significantly higher performance. However, since these matrices correspond to small problems, their optimization is of limited importance and thus we focus on matrices with large working sets that do not fit in the L2 cache. In addition, reduction of the ws for the same problem releases memory bus resources and leads to significant execution speedups. The memory intensity of the algorithm along with the effect of the indirect memory reference on x are the most crucial factors for the poor performance of SpMxV and affect all matrices. On the other hand, the irregularity in the access of x and the existence of many short rows affect performance at a smaller range and relate to a rather limited subset of the matrices. Finally, the lack of temporal locality in the matrix structures does not affect performance through issues that could be optimized (e.g. cache misses) but inherently increases the number of memory accesses.

In an attempt to quantify the effect of each of the aforementioned issues, we performed statistical analysis of our results that is summarized in Fig. 7, where a number of bars is included for each architecture. The first three bars represent the performance of a matrix vector multiplication for a dense matrix (1024×1024) stored in dense format (*dmv*) and stored in csr format for the *standard* case (*csr-dense*) and the *noind-colind* benchmark (*csr-dense-noind-colind*). The *csr-avg-nosr-reg* represents average perfromance across all matrices in the suite with working sets larger than the L2 size, while the rest of the bars correspond to all possible subsets of these matrices based on their regularity (*-irregular*/*-regular*) and on whether they are dominated by short rows or not (*-sr*/*-nosr*). The criterion for the irregularity is the presence of a significant speedup ($> 10\%$) for the *noxmiss* benchmark, while for the dominance of short rows is the presence of a large percentage ($> 80\%$) of small row lengths (< 8). Note that all matrices involved in this graph have working sets larger than the L2 size. The numbers over the bars indicate the number of matrices that belong in the particular set. Note, for example, that there exist too few matrices that are dominated by short rows and do not face performance degradation due to irregularity. This observation further supports our assumption that short rows increase the possibility for irregular accesses on x.

The most important observation from the figure is that one could set three levels of performance. The performance level determined by DMxV, the *average* performance level and the *lowest* level determined by "bad" matrices with irregularity and dominating short row lengths. Roughly speaking, the dramatic degradation (slowdown by a factor of about 2) of performance between DMxV and *average* level is due to the indirect references through `col_ind`. From that level, if a matrix exhibits some poor characteristics like irregularity and many short rows, the performance may further drop by a factor of about 1.35. On the other hand, if a matrix is not dominated by short rows and accesses x in a regular manner, its performance may exhibit a 1.1 speedup to that of the average and reach that of dense matrices stored in CSR. Note also, that the majority of the matrices falls in that last category. Thus, our experimental results lead us to the following guidelines for optimization:

1. Reduce the ws size by using the smallest possible data types (e.g. 32-bit or 16-bit integers for `col_ind`, single precision storage for x) in order to reduce the pressure on the memory subsystem. Even sacrificing CPU cycles to reduce the ws size (e.g. by applying compression) will also lead to performance improvement (as in [21]). Storage structures that increase the ws have small opportunities to succeed.

2. Reduce indirect memory referencing. This could be achieved by exploiting regular structures within the matrix such as full diagonals (as in [1]) or dense sub-

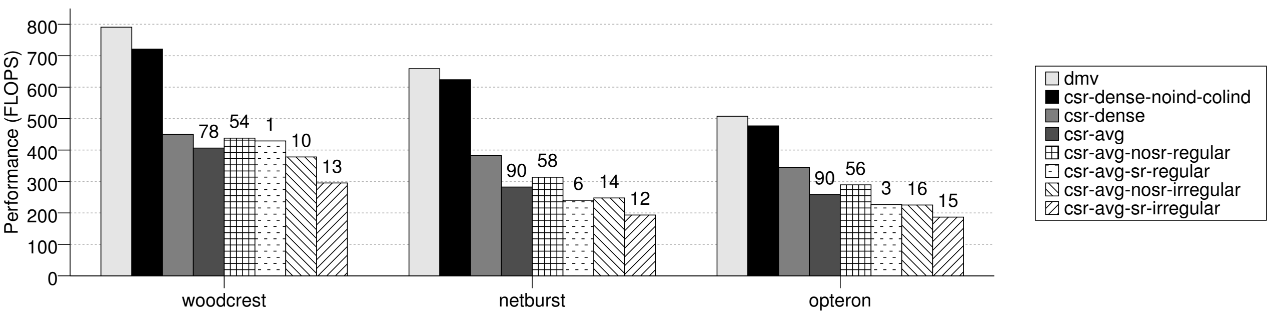


Figure 7. Conclusive performance results of the SpMxV kernel for all architectures.

blocks (e.g. BCSR format as in [3, 8, 19]).

3. Pad sparingly. Adding many non-zero elements to accomplish an optimization approach may dramatically affect performance, mainly due to the increase in the ws. The extra floating-point operations should not create such a big problem, since the CPU has idle cycles to spare. Thus, the BCSR format used in [3, 8, 19] is expected to be beneficial only in the subset of matrices that contain many dense subblocks.

4. Beware of short row lengths and loop overheads. Some optimization approaches split the matrix into a sum of submatrices (as in [1, 19]). In this case one should take care that the submatrices do not fall into the category of matrices with short row lengths. Alternatively, one may insert an additional outer loop in the multiplication kernel (as in [13]). This may also incur significant overheads especially in matrices with short rows.

5. Identify matrices with problematic access on the x vector and apply cache reuse optimizations only to them.

6. There is no need to apply software prefetching to attack the problem of the lack of temporal locality as long as the CPU supports hardware prefetching.

5 Conclusions – Future work

In this paper we presented extensive experimental results regarding the performance issues of sparse matrix-vector multiplication on modern microprocessor architectures. Our results illuminate and quantify the effect of the reported problems on the kernel's performance and can aid in forming a guideline to optimize the code. For future work, we will apply the knowledge gained from this paper in order to optimize the kernel using a short vectorization approach, which we believe that will provide performance benefits from the reduction of the working set, the indirect referencing and from the utilization of vector memory loads and floating-point operations.

References

[1] R. C. Agarwal, F. G. Gustavson, and M. Zubair. A high performance algorithm using pre-processing for the sparse matrix-vector multiplication. In *Supercomputing'92*, pages 32–41, Minn., MN, Nov. 1992. IEEE.

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

[3] A. Buttari, V. Eijkhout, J. Langou, and S. Filippone. Performance optimization and modeling of blocked sparse kernels. Technical Report ICL-UT-04-05, Innovative Computing Laboratory, University of Tennessee, 2005.

[4] T. Davis. University of Florida Sparse Matrix Collection, http://www.cise.ufl.edu/research/sparse/matrices. NA Digest, vol. 97, no. 23, June 1997.

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

[6] W. Gropp, D. Kaushik, D. Keyes, and B. Smith. Toward realistic performance bounds for implicit cfd codes. 1999.

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

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

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

[10] G. Paolini and G. Radicati di Brozolo. Data structures to vectorize CG algorithms for general sparsity patterns. *BIT Numerical Mathematics*, 29(4):703–718, 1989.

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

[12] J. C. Pichel, D. B. Heras, J. C. Cabaleiro, and F. F. Rivera. Performance optimization of irregular codes based on the

Matrix	nrows	nnz	ws(KB)	Matrix	nrows	nnz	ws(KB)
001.dense	1,000	1,000,000	15,648	051.Hamrle3	1,447,360	5,514,242	120,083
002.raefsky3	21,200	1,488,768	23,759	052.ASIC_320ks	321,671	1,827,807	36,099
003.olafu	16,146	515,651	8,435	053.Si87H76	240,369	5,451,000	90,806
004.bcsstk35	30,237	740,200	12,274	054.SiNa	5,743	102,265	1,732
005.venkat01	62,424	1,717,792	28,304	055.ship_001	34,920	2,339,575	37,374
006.crystk02	13,965	491,274	8,003	056.af_5_k101	503,625	9,027,150	152,853
007.crystk03	24,696	887,937	14,453	057.ASIC_680k	682,862	3,871,773	76,501
008.nasasrb	54,870	1,366,097	22,631	058.bcsstk37	25,503	583,240	9,711
009.3dtube	45,330	1,629,474	26,523	059.bmw3_2	227,362	5,757,996	95,297
010.ct20stif	52,329	1,375,396	22,717	060.bundle1	10,581	390,741	6,353
011.af23560	23,560	484,256	8,119	061.cage13	445,315	7,479,343	127,302
012.raefsky4	19,779	674,195	10,998	062.turon_m	189,924	912,345	18,707
013.ex11	16,614	1,096,948	17,529	063.ASIC_680ks	682,712	2,329,176	52,394
014.rdist1	4,134	94,408	1,572	064.thread	29,736	2,249,892	35,852
015.av41092	41,092	1,683,902	27,274	065.e40r2000	17,281	553,956	9,061
016.orani678	2,529	90,158	1,468	066.sme3Da	12,504	874,887	13,963
017.rim	22,560	1,014,951	16,387	067.fidap011	16,614	1,091,362	17,442
018.memplus	17,758	126,150	2,387	068.fidapm11	22,294	623,554	10,266
019.gemat11	4,929	33,185	634	069.gupta2	62,064	2,155,175	35,129
020.lhr10	10,672	232,633	3,885	070.helm2d03	392,257	1,567,096	33,679
021.goodwin	7,320	324,784	5,246	071.hood	220,542	5,494,489	91,020
022.bayer02	13,935	63,679	1,322	072.inline_1	503,712	18,660,027	303,369
023.bayer10	13,436	94,926	1,798	073.language	399,130	1,216,334	28,360
024.coater2	9,540	207,308	3,463	074.ldoor	952,203	23,737,339	393,213
025.finan512	74,752	335,872	7,000	075.mario002	389,874	1,167,685	27,383
026.onetone2	36,057	227,628	4,402	076.nd12k	36,000	7,128,473	112,226
027.pwt	36,519	181,313	3,689	077.nd6k	18,000	3,457,658	54,448
028.vibrobox	12,328	177,578	3,064	078.pwtk	217,918	5,926,171	97,704
029.wang4	26,064	177,168	3,379	079.rail_79841	79,841	316,881	6,823
030.lnsp3937	3,937	25,407	489	080.rajat31	4,690,002	20,316,253	427,363
031.lns_3937	3,937	25,407	489	081.rma10	46,835	2,374,001	38,191
032.sherman5	3,312	20,793	403	082.s3dkq4m2	90,449	2,455,670	40,490
033.sherman3	5,005	20,033	430	083.nd24k	72,000	14,393,817	226,591
034.orsreg_1	2,205	14,133	273	084.af_shell9	504,855	9,046,865	153,190
035.saylr4	3,564	12,940	286	085.kim2	456,976	11,330,020	187,742
036.shyy161	76,480	329,762	6,945	086.rajat30	643,994	6,175,377	111,584
037.wang3	26,064	177,168	3,379	087.fdif202x202x102	4,000,000	27,840,000	528,750
038.mcfe	765	24,382	399	088.sme3Db	29,067	2,081,063	33,198
039.jpwh_991	991	6,027	117	089.stomach	213,360	3,021,648	52,214
040.gupta1	31,802	1,098,006	17,902	090.thermal2	1,228,045	4,904,179	105,410
041.lp_cre_b	9,647	260,785	4,301	091.F1	343,791	13,590,452	220,408
042.lp_cre_d	8,894	246,614	4,062	092.torso3	259,156	4,429,042	75,278
043.lp_fit2p	3,000	50,284	856	093.cage14	1,505,785	27,130,349	459,204
044.lp_nug20	15,240	304,800	5,120	094.audikw_1	943,695	39,297,771	636,146
045.apache2	715,176	2,766,523	59,989	095.Si41Ge41H72	185,639	7,598,452	123,077
046.random100000	100,000	14,977,726	236,371	096.crankseg_2	63,838	7,106,348	112,533
047.bcsstk32	44,609	1,029,655	17,134	097.Ga41As41H72	268,096	9,378,286	152,819
048.msc10848	10,848	620,313	9,947	098.af_shell10	1,508,065	27,090,195	458,630
049.msc23052	23,052	588,933	9,742	099.boneS10	914,898	28,191,660	461,938
050.bone010	986,703	36,326,514	590,728	100.msdoor	415,863	10,328,399	171,128

Table 4. Matrix suite.

combination of reordering and blocking techniques. *Parallel Computing*, 31(8-9):858–876, 2005.

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

[14] Y. Saad. Sparskit: A basic tool kit for sparse matrix computation. Technical report, Center for Supercomputing Research and Development, University of Illinois at Urbana Champaign, 1990.

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

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

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

[18] 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.

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

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

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