Performance Evaluation of the Sparse Matrix-Vector Multiplication on Modern Architectures

Georgios Goumas, Kornilios Kourtis, Nikos Anastopoulos, Vasileios Karakasis, Nectarios Koziris

National Technical University of Athens
School of Electrical and Computer Engineering
Computing Systems Laboratory
Zografou Campus, Zografou 15780, Greece
{goumas, kkourt, anastop, bkk, nkoziris}@cslab.ece.ntua.gr

October 20, 2008

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 into the details of SpMxV performance, we conduct a suite of experiments on a rich set of matrices for three different commodity hardware platforms. In addition, we investigate the parallel version of the kernel and report on the corresponding performance results and their relation to each architecture's specific multithreaded configuration. Based on our experiments we extract useful conclusions that can serve as guidelines for the optimization process of both single and multithreaded versions of the kernel.

Keywords—Sparse matrix-vector multiplication, Multicore architectures, Scientific applications, Performance evalution

1 Introduction

Matrix-vector multiplication is performed in a large variety of applications in scientific and economic modeling, signal processing, document retrieval, and others. Quite commonly, the matrix that participates in the computation is sparse, as for example is the case of matrices arising from the discretization for physical processes. Sparse matrix-vector computations and, in particular, sparse matrix-vector multiplication (SpMxV) have been recently categorized as one of the "seven dwarfs", i.e., seven numerical methods that are believed to be important for science and engineering for at least the next decade [2]. SpMxV is generally reported to perform poorly on modern microprocessors (e.g., 10% of peak performance [26]) due to a number of issues concerning the algorithm itself, the storage formats, and the sparsity patterns of the matrices.

Primarily, matrix-vector multiplication is a memory-bound kernel posing more intense memory access needs than other traditional algebra kernels, like dense 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. This fact greatly degenerates the SpMxV performance expressed in MFLOPS. Furthermore, 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 [4] 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 further degrade the kernel's performance since they add additional memory access operations and cache interference. Sparse matrices also create 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 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 useful computations is performed in each loop iteration.

The great importance and the singular performance behavior of SpMxV have attracted intense scientific attention [1, 5, 8, 11, 12, 15, 17–20, 23–27]. 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 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 [11] as a transformation to tame irregular accesses on the input vector and exploit its inherent reuse, like in dense matrix optimizations. One-dimensional blocking is also proposed by Pinar and Heath [20] in order to reduce indirect memory references, while, quite recently, Buttari et al. [5], and Vuduc and Moon [26] accentuate the merit of blocking (the latter with variable-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 [27] 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 [8, 11, 18, 19, 24].

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 commodity microarchitectures. In order to achieve this goal, we have categorized the problems of the algorithm as reported in literature and experienced in practice. For each problem we conduct a series of experiments in order to either quantify or draw a qualitative conclusion of its effect on performance as accurately as possible. Our experimental results provide valuable insight into the performance of SpMxV on modern microprocessors and reveal issues that will probably prove particularly useful in the process of optimization. The code performs poorly on modern microprocessors as well. However, the issues that need to be taken into consideration in order to optimize it are better understood and quantified. In addition, we develop multithreaded versions of SpMxV since it is important to evaluate the speedup of the parallelized algorithm for SMP, CMP, and SMT machines. Although SpMxV is an easily parallelizable code that needs no synchronization or data exchange between threads, it is far from achieving the theoretically expected linear speedup. In this case, issues arising from novel multithreaded architectures affect performance considerably and need to be further illuminated and evaluated. Our experiments for both the single and multithreaded case are executed on three different microprocessors (Intel Core 2 Xeon , Intel Pentium 4 Xeon, AMD Opteron) for a large suite of 100 sparse matrices selected from Tim Davis' collection [7]. Based on the experience gained from the interpretation of the experimental results, we are able to provide solid guidelines for the optimization of both the single and multithreaded version of SpMxV.

The next of the paper is organized as follows: Section 2 discusses previous work on the optimization of SpMxV and Section 3 presents the basic kernel and its problems as reported in literature. Section 4 presents a thorough experimental evaluation of the aforementioned problems in single and multithreaded versions, which leads to a number of guidelines summarized in Section 5. We conclude this paper with overall conclusions and a discussion of directions for future work in Section 6.

2 Related work

Because of its importance, sparse matrix-vector multiplication has attracted intensive scientific attention during the past two decades. The proposal of efficient storage formats for sparse matrices like CSR, BCSR, CDS, Ellpack-Itpack, and JAD [4, 17, 22] was one of the primary concerns. Elaborating on storage formats, Agarwal et al. [1] decompose a matrix into three submatrices: the first is dominated by dense blocks, the second has a dense diagonal matrix, while the third contains the remainder of the matrix elements. By using a different format for each submatrix, the authors try to optimize execution based on the special characteristics of each submatrix. Temam and Jalby [23] 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 [24] 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 [27] 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 a HP PA-RISC processor for each of the 10 sparse matrices used. Pinar and Heath [20] refer to irregular and indirect accesses on `x` as the main factors responsible for performance degradation. Focusing on indirect accesses, the application of one-dimensional blocking with the BCSR storage format is proposed in order to drastically reduce the number of indirect memory references. In addition, a column reordering technique which enables the construction of larger dense sub-blocks is also proposed. 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 [10–12]. Moreover, 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 [8] 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. [25] estimate the performance bounds of the algorithm and evaluate the register blocked code with respect to these bounds. Furthermore, they propose a new approach to select near-optimal register block sizes. Mellor-Crummey and Garvin [15] accentuate the problem of short row lengths and propose the application of the well-known unroll-and-jam compiler optimization in order to overcome this problem. Unroll-and-jam achieves a 1.11–2.3 speedup on MIPS R12000, Alpha 21264A, Power3-II, and Itanium processors for two matrices taken from the SAGE package. Pichel et al. [18] 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 [19]. 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. [5] provide a performance model for the blocked version of the algorithm based on BCSR format and propose a method to select dense blocks efficiently. They experiment on a K6, a Power3, and an Itanium II processor for a suite of 20 sparse matrices and validate the accuracy of the proposed performance model. Vuduc et al. [26] extend the notion of blocking in order to exploit variable block shapes by decomposing the original matrix to a proper sum of submatrices storing each submatrix in a variation of the BCSR format. 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 on every processor, except for Itanium II. Finally, Willcock and Lumsdaine [28] 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.

As far as the parallel, multithreaded version of the code is concerned, past work focuses mainly on SMP clusters, where researchers either apply and evaluate known uniprocessor optimization techniques on SMPs, such as register or cache blocking [8, 11], or examine reordering techniques in order to improve locality of references and minimize communication cost [6, 18]. More specifically, Im and Yelick [11] apply register and cache blocking on an 8-way UltraSparc SMP. They also examine reordering techniques combined with register blocking. However, the results are satisfactory only in the case of highly irregular sparse matrices, but the scalability of the algorithm is still very low. Pichel et al. [18] also examine reordering techniques and locality schemes. They propose two locality heuristics based on row or row-block similarity patterns, which they use as objective functions to two reordering algorithms in order to gain locality. Results are presented in terms of L1 and L2 cache miss rate reduction based mainly on a trace-driven simulation. The effect of these reordering techniques in load balancing is also discussed. Geus and Röllin [8] examine three parallelization schemes using MPI combined with Cuthill-McKee reordering technique in order to minimize data exchange between processors. Experiments are conducted on a series of high performance architectures, including, among others, the Intel Paragon and the Intel Pentium III Beowulf Cluster. The authors also outline the problem of the interconnection bandwidth while commenting on the results. In a higher level, Catalyuerek and Ayakanat [6] propose an alternative data partitioning scheme based on hypergraphs in order to minimize communication cost. Finally, Kotakemori et al. [13] evaluate different storage formats of sparse matrices on a SGI Altix3700 ccNUMA machine using an OpenMP parallel version of the SpMxV code. The authors implement a NUMA-aware parallelization scheme, which yields almost linear speedup in every case.

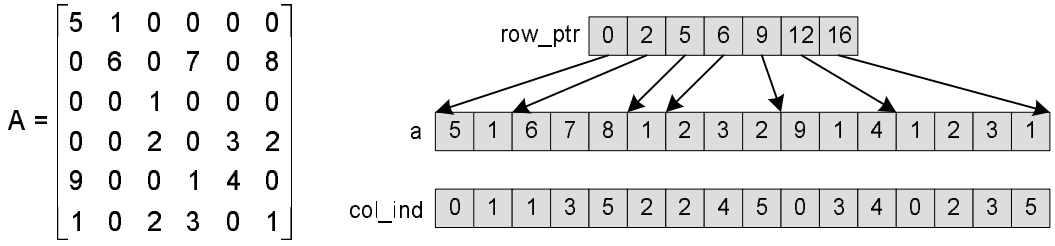
Quite recently, Williams et al. [29] have presented an evaluation of SpMxV on a set of emerging multicore architectures. Their study covers a wide and diverse range of high-end chip multiprocessors, including recent multicores from AMD (Opteron X2) and Intel (Clovertown), Sun's Niagara2 and platforms comprised of one or two Cell processors.

The authors offer a clear view of the gap between the attained performance of the kernel, and the peak performance of each architecture it is executed, both in terms of memory bandwidth and computational throughput. Although they designate memory bottleneck as the major hurdle of the algorithm from attaining high parallel performance, they do little effort in estimating its extent, e.g. through quantifying the additional benefit from NUMA-aware memory allocation, or examining the impact of intra-thread cache sharing and classifying the behavior of the algorithm according to each matrix's ability to fit in each platform's aggregate cache. Besides that, they focus to a large extent on single-threaded optimizations and evaluate their techniques on a rather small set of matrices (14).

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 previous generation microarchitectures, (c) the conclusions are sometimes contradictory, (d) the performance gains attained by the proposed methods are not thoroughly analyzed in relevance to the specific problems attacked, and (e) specific problems of the multithreaded versions are not reported. The goal of this work is to understand the performance issues of single and multithreaded SpMxV codes on modern microprocessors. 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 modern microprocessors.

3 Basic algorithm and problems

The most frequently applied storage format for sparse matrices is the Compressed Storage Row (CSR) [4]. 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. Figure 1a shows an example of the CSR format for a sparse 6×6 matrix. Figures 1b and 1c show the implementation of the matrix-vector multiplication for a dense $N \times M$ matrix and for a sparse matrix stored in CSR format, respectively.



(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.

According to the literature, SpMxV presents a set of problems that can potentially affect its performance. These problems are listed below.

(a) *No temporal locality in the matrix.* This is an inherent problem of the algorithm which is irrelevant to the sparsity of the matrix. Unlike other important numerical codes, such as Matrix Multiplication (MxM) and LU decomposition, the elements of the matrix in SpMxV are used only once [5, 15].

(b) *Indirect memory references.* This is the most apparent implication of sparsity. In order to save memory space and 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 [20].

(c) *Irregular memory accesses to vector* `x`. Unlike the case of dense matrices where the access to the vector `x` is sequential, this access in sparse matrices is irregular and depends on the sparsity structure of the matrix. This fact complicates the process of exploiting any spatial reuse in the access to vector `x` [8, 10, 18].

(d) *Short row lengths.* Although not so obvious, this problem is very often met in practice. Many sparse matrices exhibit a large number of rows with short length. This fact may degrade performance due to the significant overhead of the outer loop when the trip count of the inner loop is small [5, 27].

In the next section we will evaluate the impact of each of the above reported problems on the performance of the algorithm when executed on modern microprocessors.

4 Performance evaluation

4.1 Experimental process and preliminary evaluation

Our experiments were performed on a set of 100 matrices (see Table 1), the majority of which was selected from Tim Davis' collection [7]. The first matrix is a dense 1000×1000 matrix, matrices 2–45 are also used in SPARSITY [10], matrix #46 is a 100000×100000 random sparse matrix with roughly 150 non-zero elements per row, matrix #87 is a matrix obtained by a 5-pt finite difference problem for a $202 \times 202 \times 102$ regular grid created by SPARSKIT [21], while the rest are the largest matrices of the collection both in terms of non-zero elements and number of rows. All matrices are stored in CSR format.

The hardware platforms used for the evaluation of the kernel consist of a 2-way SMP Intel Core 2 Xeon processor (*Woodcrest*), a 2-way SMP Intel Pentium 4 Xeon (*Netburst*), and a 2-way ccNUMA AMD Opteron (*Opteron*). These processors may be considered as a representative set of commodity hardware platforms that incorporate innovative low-end technologies and support the execution of multiple software or hardware threads on the same die. The set of processors used presents a variety of microarchitectural characteristics which allow for a better understanding

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 1: Matrix suite.

of the performance of the parallel SpMxV kernel. *Netburst* is a Simultaneous Multithreading (SMT) processor, where the architectural state of the processor is duplicated, and every other processor resource is shared, statically or dynamically, between the two executing threads (see Figure 2). *Woodcrest* and *Opteron* are Chip Multiprocessors (CMP), where two processor cores are incorporated into the same die and share the higher levels of the cache hierarchy (*Woodcrest*) or the integrated memory controllers (*Opteron*). Looking out of the die, there also exist considerable architectural differences; *Netburst* and *Woodcrest* are 2-way SMP machines which access the shared memory through the same Front-Side Bus (FSB) (Figure 3a), while *Opteron* is a cache-coherent NUMA machine, where each processor has its own memory controller which controls different memory banks (Figure 3b). A more

Platform	**Netburst**	***Opteron***	***Woodcrest***
Clockspeed	2.8GHz	1.8GHz	2.6GHz
L1 - data	16KB, 8-way	64KB, 2-way	32KB, 8-way
L1 - instruction	12K μops	64KB, 2-way	32KB, 8-way
L2 - unified	1MB, 8-way	1MB, 16-way, exclusive	4MB, 16-way
#HW threads	4	4	4
Threads arrangement	2 SMP Processors $\times$ 2 Hyperthreads	2 ccNUMA Processors $\times$ 2 Cores	2 SMP Processors $\times$ 2 Cores
Shared resources within physical package	all caches, execution resources, instr. fetch-decode-schedule-retirement logic	memory controller, hyper-transport links	L2 cache

Table 2: Specifications of hardware platforms used.

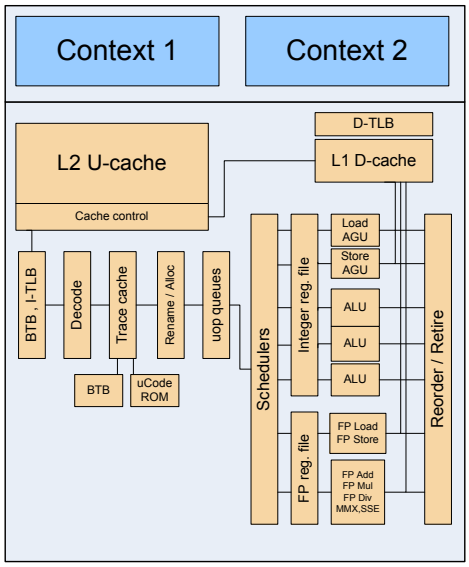
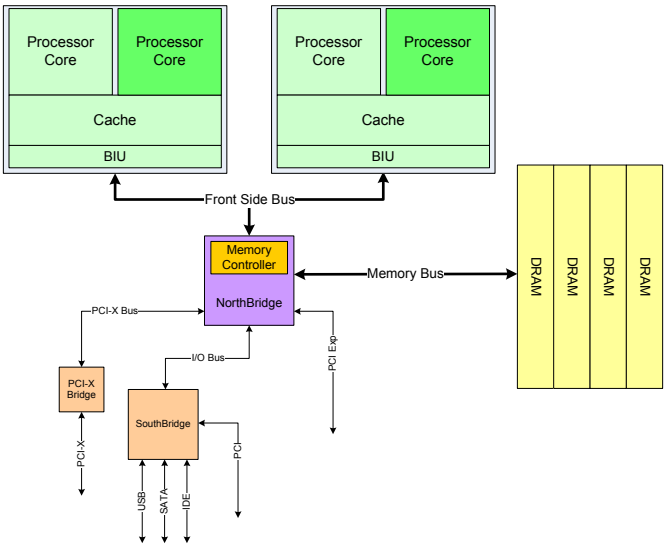


Figure 2: The HyperThreading technology of the *Netburst* microarchitecture. The architectural state of the processor is duplicated, thus forming two logical processors recognized by the operating system. Every other resource of the processor is dynamically or statically shared between the two hardware contexts.

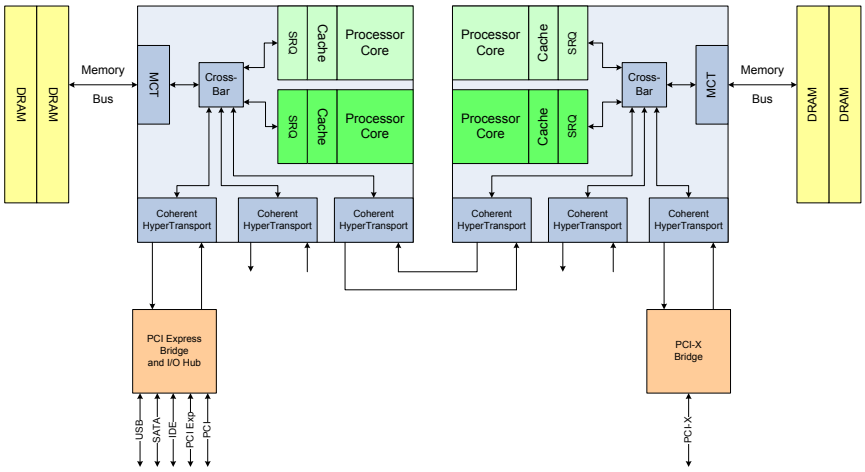
detailed description of each processor is presented in Table 2.

All systems run Linux (kernel version 2.6) for the x86_64 ISA, and all programs were compiled using `gcc` version 4.1 with the `-O3` and `-funroll-loops` optimization switches turned on. The latter switch causes the compiler to apply aggressive loop unrolling to all loops of the program. In our experience, the unroller of version 4.1 of `gcc` can provide significant speedup for tight loops. In order to confirm that loop unrolling is beneficial for the SpMxV code, we conducted experiments with the kernel compiled without unrolling. A summary of the results obtained is presented in Table 3, where it is shown that this compiler optimization provides significant speedup, especially in the case of the *Woodcrest* processor.

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. In order to evaluate



(a)



(b)

Figure 3: The Xeon SMP(a) and the ccNUMA Opteron (b) architectures. Both cores of *Woodcrest* share the L2-cache and the Bus Interface Unit (BIU), which interfaces them to the common FSB and the common memory controller (MC). In contrast, *Opteron* cores share only the HyperTransport links and a memory controller integrated to the same physical package. Each memory request is served independently for each package through the integrated memory controller. Remote memory accesses are routed to the other package's memory controller through the HyperTransport links.

Processor	*matrices with speedup* $> 10\%$	*average speedup*	*max speedup*
Woodcrest	70	1.41	2.56
Netburst	21	1.21	1.65
Opteron	13	1.21	1.46

Table 3: Performance impact of loop unrolling on SpMxV kernel.

performance, we used the floating point operations per second (FLOPS) metric of each run, which was calculated by dividing the total number of floating point operations ($2 \times nnz$) by the execution time. We applied double precision arithmetic and used 64-bit size integers for the representation of `col_ind` and `row_ptr` indices, despite the fact that this work focuses on matrices that fit completely into main memory and for most modern systems 32-bits would suffice. This decision was based on the fact that memory size increases with a very large rate and it won't be long before matrices that require 64-bit integers can be stored exclusively into main memory. 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 is present in the cache because either it has just been produced or was recently accessed. Apart from the execution time, we also measured a variety of performance monitoring events via the interface provided by each processor.

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

Table 4: Performance impact of hardware prefetching on SpMxV kernel for Intel processors.

One of the most prominent characteristics of modern microprocessors 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 cache-lines from main memory to the CPU cache hierarchy. In order to gain a better insight into the performance issues involved, we conducted experimental tests to evaluate the effect of hardware prefetching on the SpMxV kernel by disabling it. We present results only for Intel processors 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 4. Note that there was no case where hardware prefetching had a negative impact on performance.

4.2 Single-threaded evaluation

4.2.1 Basic performance of serial SpMxV

Figure 4 shows the detailed performance results for the SpMxV kernel in terms of MFLOPS for each matrix and architecture in the experimental set. To gain a better understanding of the results, we consider the benchmark of a Dense Matrix-Vector Multiplication (DMxV) for a dense 1024×1024 matrix as an upper bound for the peak

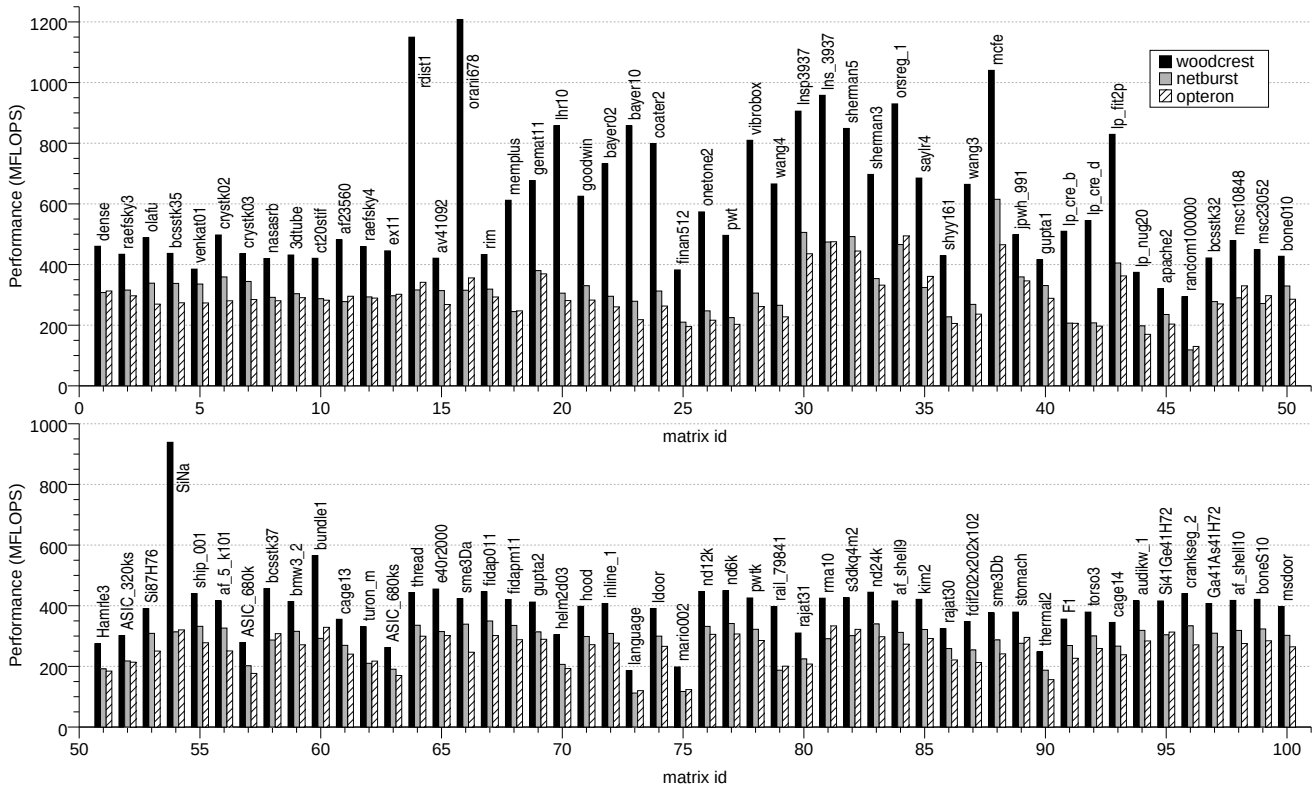


Figure 4: Performance of the SpMxV kernel: MFLOPS for each matrix and architecture.

performance of the SpMxV kernel. Summarized results are presented in Table 5. 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* deviates greatly in some cases. This is apparent, for example, in matrices #14, #16, and #54, 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 4 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; matrices whose working set fits perfectly into L2 cache, and thus experience only compulsory misses, and those whose working set is larger than the L2 cache size and may also experience capacity misses. The working set (ws) in bytes assuming double precision arithmetic and 64-bit integer indexing is computed by the formula

$$ws = (nnz \times 2 + nrows \times 2 + ncols) \times 8,$$

where $nrows$ and $ncols$ are the number of rows and columns of the input matrix, respectively, and nnz is the number of non-zero elements of the matrix. Figure 5 presents the performance attained by each matrix relative to its working set. 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, obviously, significantly higher performance should be 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.

Processor	max (MFLOPS)	min (MFLOPS)	average (MFLOPS)	DMxV (MFLOPS)
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 5: Summarized results for the performance of the SpMxV kernel.

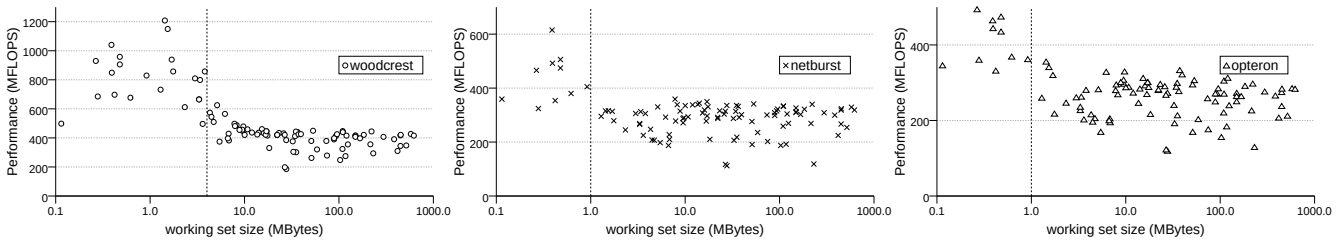


Figure 5: Performance of the SpMxV kernel in relation to the working set size for all architectures. The vertical line in each graph designates the size of the L2 cache for each architecture.

Additionally, Figure 6 presents the performance of each matrix with respect to the L2 cache miss-rate as measured from the performance counters of 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, and matrices with similar miss-rates have significantly varying MFLOPS.

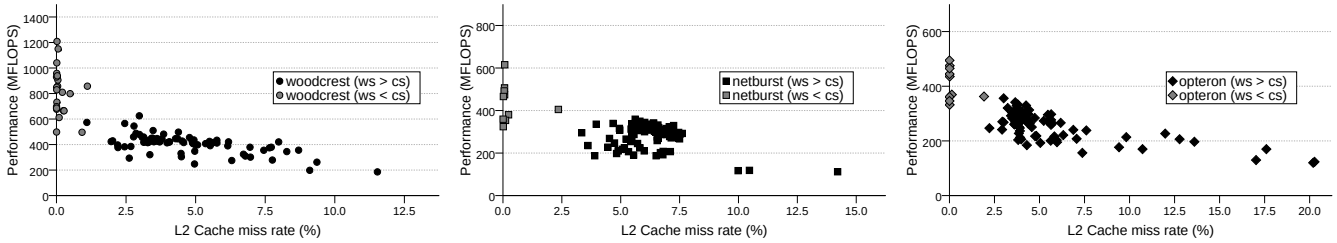


Figure 6: Performance of the SpMxV kernel in relation to the L2 cache miss-rate as reported from the performance counters.

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 operations. 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 6.

Processor	***Speedup***		***# Matrices***		
	average	*max*	*Speedup* $> 10\%$	*Speedup* $> 20\%$	*Speedup* $> 30\%$
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 6: Summarized results for the *noxmiss* benchmark. The table presents the average and maximum speedup, and the 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 to 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. Figure 7 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 5). The obvious conclusion that can be drawn from Figure 7 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 only factor. Another important observation that supports the above point is that the matrices reported in Figure 7 coincide, with few exceptions, with the matrices that benefited by the *noxmiss* benchmark. 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.

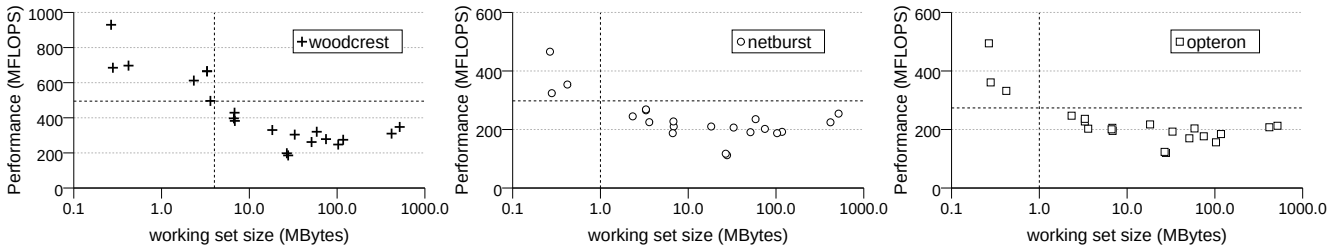


Figure 7: Performance of matrices with small number of elements, less than 8 for 80% or more of their rows, in relation to their working set. Vertical line marks the L2 cache size for each processor. Horizontal line marks the average performance.

4.2.4 Indirect memory references

Two indirect memory accesses exist 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.

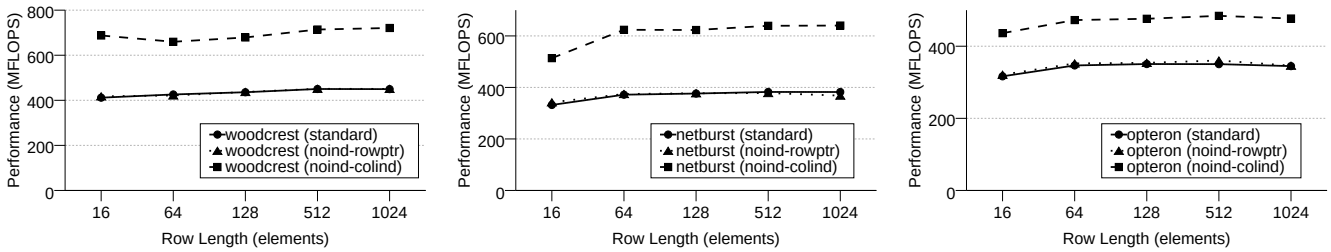


Figure 8: Performance of the *standard*, *noind-rowptr*, and *noind-colind* versions for different number of elements per row.

Figure 8 summarizes the performance measured for a subset of the row lengths applied. Note that 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. Regardless, this degradation of performance should not be attributed to the actual indirect reference per se, since the need of indirect access to `x` at the algorithm level have a series of side-effects that are tightly coupled to that need. Specifically, the `col_ind` structure increases significantly the working set of the algorithm, which can greatly affect performance as it is discussed in the next section, ruins locality of references to `x`, since it leads to irregular accesses, and finally, adds an additional instruction and a RAW

dependence in the inner loop of the algorithm, which may further degrade performance by incurring pipeline stalls. In practice, it is very difficult to decouple these repercussions of indirect references and examine the effect of each one independently. In our case, the *noind-colind* benchmark accounts for all the above considerations as a whole, except for the reference pattern on `x`, which was the same for *noind-colind* and *standard* benchmarks; this issue was separately addressed in Section 4.2.2.

4.2.5 Lack of temporal locality

Generally, the lack of temporal locality is an issue that can greatly affect performance. Nevertheless, the data structures of CSR (Figure 1) are accessed in a rather regular and streaming pattern with unit stride. Consequently, the hardware prefetcher of modern microarchitectures is able to detect such simple access patterns and transparently fetch their corresponding cache-lines from memory (see Section 4.1 for an experimental evaluation of hardware prefetching on SpMxV). Thus, it is quite safe to assume that the lack of temporal locality in the matrix causes an insignificant number of cache misses, and therefore, performance is not directly affected by this particular factor.

On the other hand, the lack of temporal locality has an important implication on the ratio of floating point operations to memory accesses, which can greatly affect performance. As a result of this lack of locality and of the one-pass nature of the algorithm, the SpMxV kernel performs $O(n^2)$ floating point operations and $O(n^2)$ memory operations, which further accentuates the memory wall problem as compared to other computational kernels, like MxM, which perform $O(n^3)$ floating point operations and $O(n^2)$ memory operations. Therefore, the performance of the kernel in the systems under consideration is not determined by the processor speed, but by the ability of the memory subsystem to provide data to the CPU [9]. In order to further illuminate this characteristic of the kernel, we performed a simple, comparative set of experiments. We used 32-bit integers instead of 64-bit for the `col_ind` structure in order to reduce the total size of the working set. This modification led eventually to a 22.4% average reduction of the working set on every matrix. Respectively, Table 7 shows the average speedup attained for each processor over all matrices. It is quite impressive that the alleviation of the memory bus pressure in terms of data volume led to an almost analogous increase in performance. These results complement the observations from the *noind-colind* benchmark, where the dramatic increase in performance could be rather safely attributed to the significant reduction of the working set. The `col_ind` structure consumes a great portion of the algorithm's working set, since its size equals the non-zero elements of the sparse matrix.

Processor	*average speedup*
Woodcrest	1.20
Netburst	1.29
Opteron	1.17

Table 7: Average speedup over all matrices achieved by each processor using 32-bit indexing in `col_ind` structure, instead of 64-bit. This corresponds to an average 22.4% reduction in the working set of the algorithm.

4.2.6 Interpretation of the experimental results

Based on the experimental evaluation of the previous sections, a number of interesting conclusions for the single-threaded version can be drawn. Firstly, the performance of the kernel is greatly affected by the matrix working set. As shown in Figure 5, 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 speedup. The memory intensity of the algorithm along with the effects of the indirect memory reference to `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 directly through issues that could be optimized, e.g., cache misses, but inherently increases the number of memory accesses.

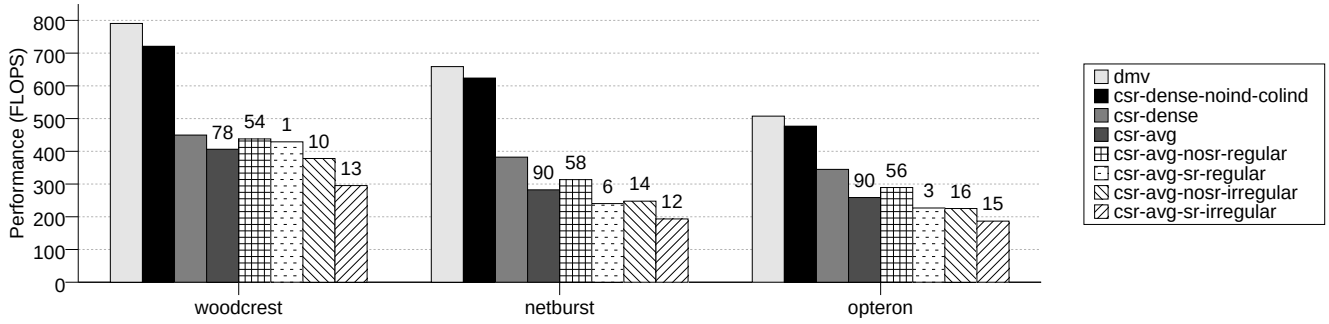


Figure 9: Conclusive performance results of the SpMxV kernel for all architectures.

In an attempt to quantify the effect of each of the aforementioned issues, we performed a statistical analysis of our results that is summarized in Figure 9, where a number of bars are included for each architecture. The first three bars refer to benchmarks applied to the dense matrix. Specifically, the first bar (*dmv*) corresponds to a dense matrix-vector multiplication benchmark with the dense matrix stored in the normal dense format. The second bar refers to a dense matrix-vector multiplication benchmark with the matrix stored in CSR format but with indirect referencing through `col_ind` disabled (*csr-dense-noind-colind*), and the third bar refers to a dense matrix-vector multiplication benchmark with the matrix stored in normal CSR format. The *csr-avg-nosr-reg* bar represents the average performance across all matrices in the suite with working sets larger than the L2-cache size, while the rest of the bars correspond to all possible subsets of these matrices based on their regularity (*-irregular* and *-regular*) and on whether they are dominated by short rows or not (*-sr* and *-nosr*). The criterion for the irregularity is the presence of a significant speedup ($> 10\%$) in 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-cache size. The numbers over the bars indicate the number of matrices that belong to the particular set. Note 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 the DMxV and the *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.

4.3 Multithreaded evaluation

The SpMxV kernel is an easily parallelizable kernel since there does not exist any loop-carried dependency that could render the parallelization of SpMxV a more painful task. Nevertheless, there exist a number of issues that can significantly affect performance and should be considered during the parallelization process. These issues will be addressed in the following sections.

For the parallelization of the SpMxV kernel, we used explicit threading through the NPTL 2.3.6 library, which implements the POSIX threads. The system call interface provided by the Linux scheduler was also used in order to explicitly assign threads to specific processing elements (logical processors). In particular, we used the `sched_setaffinity` system call. The rest of the experimental configuration (compiler and optimization flags, hardware platform, matrix set) and the experimental process used were the same as described for the single-threaded evaluation.

In order to better model the underlying architecture and reveal any architecture-specific advantages or disadvantages, we used different configurations for the multithreaded execution. Specifically, we use the notation $P \times T = nthreads$ to denote that we use P physical processors (packages) and T logical processors (hyperthreads or cores) within the same package. Thus, the notation 1×2 means that we use two threads in total which are mapped to a single physical package but in different logical processors within that package. Conversely, the notation 2×1 means that each thread is mapped to a single logical processor in different physical packages.

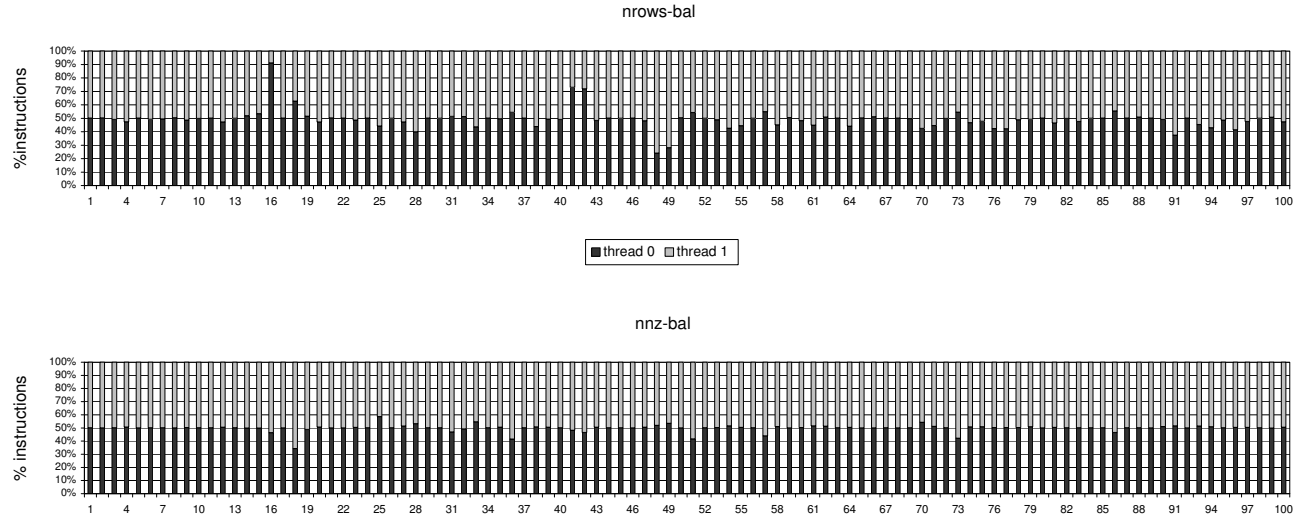


Figure 10: Load distribution (total instructions executed) between the row-based and the non-zero-based partitioning schemes. The non-zero-based scheme leads to much better load distribution.

4.3.1 Load balancing issues

An important issue that arises when parallelizing the SpMxV kernel is the load balance among the different threads since the sparsity pattern varies within a matrix. In order to measure the load balancing between the threads, we have used the following metric:

$$\textit{load-imbalance} = \frac{\#\text{ instructions of the most loaded thread}}{\#\text{ instructions of a thread}}$$

The actual number of instructions executed by each thread was obtained through the performance counters of the processors.

At first we split the sparse matrix row-wise assigning the same number of rows to each thread; we call this partitioning scheme *nrows-bal*. The results for two threads on *Netburst* are also presented in Figure 10, from which it is obvious that the two threads are not balanced. The average *load-imbalance* factor rises up to 1.26 for this scheme, and 49 matrices had a *load-imbalance* factor greater than 1.05. This is quite predictable since, in general, the non-zero elements of a sparse matrix are not uniformly distributed over its rows. Consequently, if the sparsity pattern of the matrix is biased towards the upper or lower half, this naïve scheme yields poor results.

A more sane partitioning scheme is to split the matrix row-wise such that the same amount of non-zero elements would be assigned to each thread; we will call this scheme *nnz-bal*. The split-points in the non-zero element array `a` are at positions $k \times \frac{nnz}{nthreads}$, where $k = 1, 2, \ldots, nthreads$. In Figure 10, the *load-imbalance* factor for two threads with this partitioning scheme is depicted. It is obvious that this scheme leads to better load balance. The average *load-imbalance* factor is 1.05, and only for 22 matrices did the *load-imbalance* factor surpassed 1.05. It is worth

Platform	$\mathbf{1 \times 2}$		$\mathbf{2 \times 1}$		$\mathbf{2 \times 2}$	
	nrows-bal	*nnz-bal*	*nrows-bal*	*nnz-bal*	*nrows-bal*	*nnz-bal*
Netburst	1.087	1.109 (+2.2%)	1.159	1.183 (+2.4%)	1.050	1.076 (+2.6%)
Woodcrest	1.853	1.967 (+11.4%)	1.336	1.372 (+3.6%)	2.585	2.793 (+20.8%)
Opteron	1.618	1.700 (+8.2%)	1.544	1.628 (+8.4%)	2.217	2.405 (+18.8%)

Table 8: Impact of load-balancing to different thread mappings (speedup).

noticing here, however, that although this scheme almost equally distributes non-zero elements among threads, there exist matrices that do not benefit significantly. This is mainly due to the fact that different sparsity patterns lead to different instruction streams regardless of the number of non-zero elements assigned to each thread. For example, if a thread is assigned a large number of short rows, then it will be further burdened from an increased amount of loop control instructions (see Section 4.2.3). A more sophisticated partitioning scheme, either dynamic or static, that will better consider these issues is a matter of future research. In the following, we use the *nnz-bal* scheme since it provides a rather balanced split. Table 8 depicts the speedup achieved by different thread mappings using the two partitioning schemes. The *nnz-bal*, which better balances the computations of the kernel, provides an additional improvement in performance, which might reach 20% for a four-thread configuration.

4.3.2 Shared Memory architectures

In this section, we present and discuss the aggregate results of the SpMxV kernel on every architecture and for each possible multithreading scheme. We focus specifically on issues that arise when certain architecture resources, such as processor internal resources, caches, or main memory, are shared among the processing elements. Table 9 and Table 10 present the speedup (average, minimum, maximum) and the absolute performance in MFLOPS achieved by each architecture for every multithreading scheme, respectively. Again here, we separate between matrices that perfectly fit into the *effective cache size* of the architecture and matrices that do not. The effective cache size is the total L2-cache size that is available for each multithreading scheme. For example, the effective cache size for a 1×2 scheme on *Opteron*, which has a private L2-cache for each core, is double the real L2-cache size, since every thread has the same amount of L2-cache storage as in the serial case, but half the working set. On *Netburst* and *Woodcrest*, on the other hand, the effective cache size in such case is the same as in the serial one, since their processing elements share the L2-cache. Figures 11, 12, and 13 depict the actual speedup achieved by each scheme and architecture for every matrix. Matrices are sorted according to their working sets, while vertical lines denote the effective cache size visible from each multithreading scheme.

It is obvious from the above tables and figures, that the scalability of the kernel is very poor, especially when the working set does not fit in cache, despite the minimal requirements of the algorithm for data exchange and synchronization. This fact along with the inherent memory intensity of the algorithm lead us to the assumption that the main bottleneck of the parallel version of SpMxV on a shared memory architecture should be the simultaneous

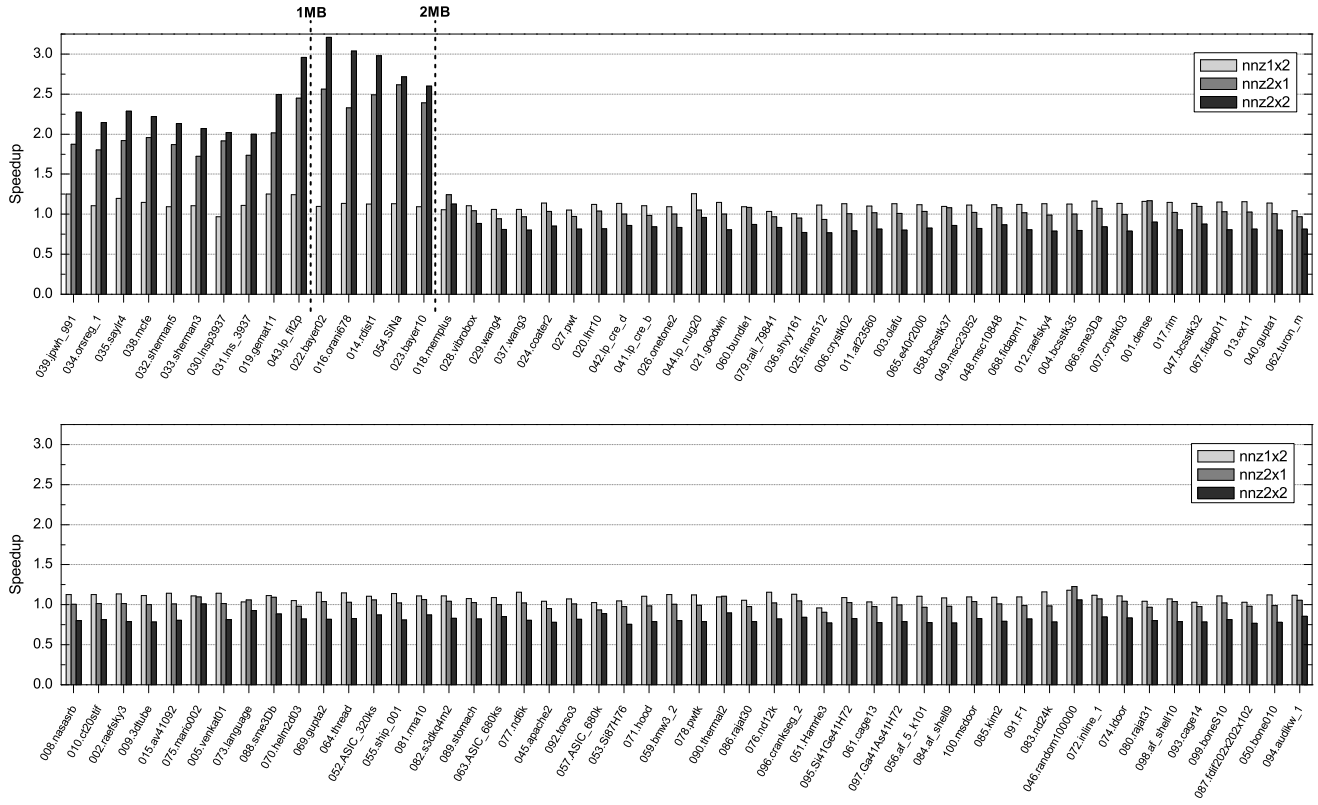


Figure 11: Speedup achieved on *Netburst* for every multithreading scheme. The first vertical line denotes the effective cache size for the serial execution and the 1×2 multithreading scheme, and the second one denotes the effective cache size for the 2×1 and 2×2 multithreading schemes.

Platform	$\mathbf{1 \times 2}$						$\mathbf{2 \times 1}$						$\mathbf{2 \times 2}$					
	$ws < cs$			$ws > cs$			$ws < cs$			$ws > cs$			$ws < cs$			$ws > cs$		
	min	*max*	*avg*	*min*	*max*	*avg*	*min*	*max*	*avg*	*min*	*max*	*avg*	*min*	*max*	*avg*	*min*	*max*	*avg*
Netburst	0.96	1.25	**1.14**	0.96	1.26	**1.10**	1.72	2.62	**2.11**	0.91	1.24	**1.02**	2.00	3.21	**2.48**	0.76	1.13	**0.83**
Woodcrest	1.55	2.67	**1.98**	1.41	5.05	**1.96**	1.17	2.19	**1.79**	1.12	1.46	**1.18**	2.84	8.02	**4.27**	1.56	5.50	**2.13**
Opteron	1.78	3.01	**2.22**	1.26	1.80	**1.61**	1.78	3.01	**2.20**	1.11	2.35	**1.53**	3.14	6.25	**4.52**	1.16	3.44	**1.81**

Table 9: Aggregate speedup results for every architecture and every multithreading scheme.

Platform	$\mathbf{1 \times 2}$		$\mathbf{2 \times 1}$		$\mathbf{2 \times 2}$	
	$ws < cs$	$ws > cs$	$ws < cs$	$ws > cs$	$ws < cs$	$ws > cs$
Netburst	503	351	824	323	965	260
Woodcrest	1612	815	1329	460	2967	849
Opteron	816	435	812	410	1495	485

Table 10: Average performance (MFLOPS) of each multithreading scheme on every platform.

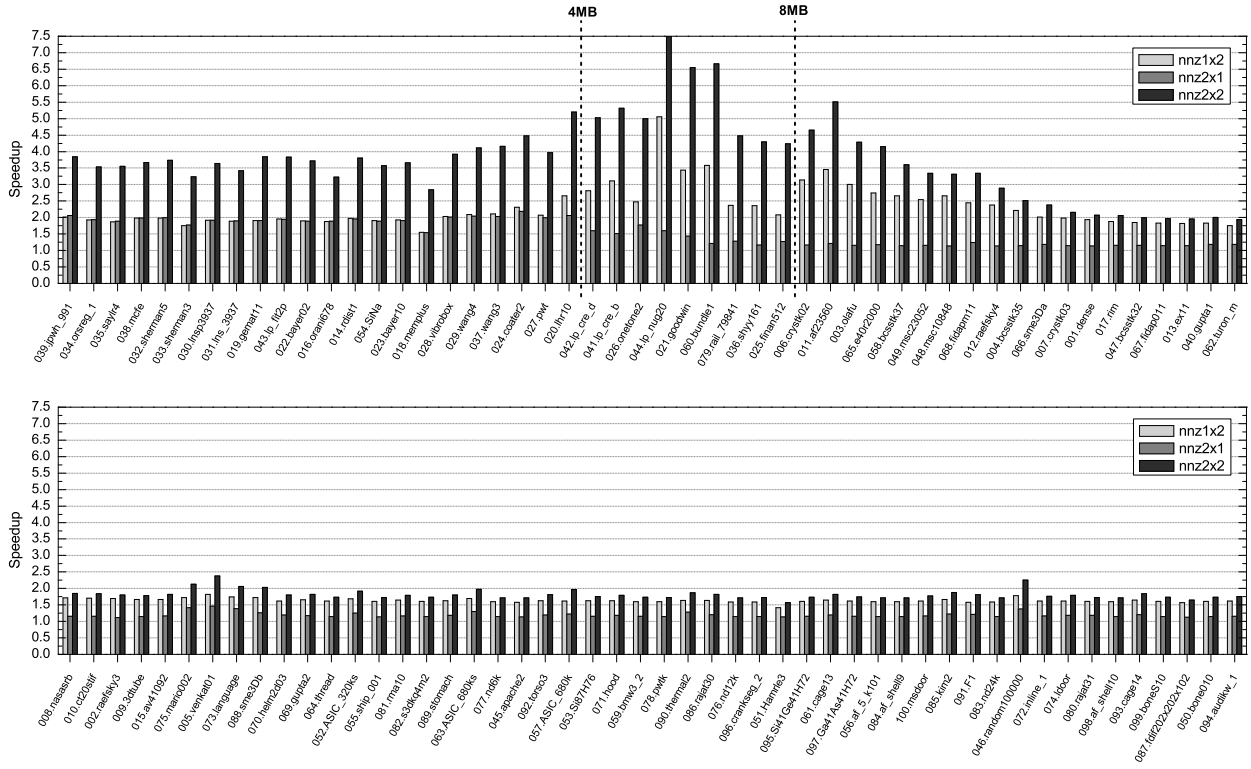


Figure 12: Speedup achieved on *Woodcrest* for every multithreading scheme. The first vertical line denotes the effective cache size for the serial execution and the 1×2 multithreading scheme, and the second one denotes the effective cache size for the 2×1 and 2×2 multithreading schemes.

access from all the processing elements to the shared bus and memory. In the following, we will focus on the performance results for each architecture, and we present results from other performance metrics in order to solidify the aforementioned assumption.

Effect of shared resources on performance: All architectures under consideration have a set of resources that are shared among processing elements at a certain level, ranging from resources inside the processor to the main memory[1]. On *Netburst*, which is a SMT machine and shares resources inside the processor, SpMxV fails to scale well with the 1×2 (1.14 speedup) and 2×2 (2.48 speedup) schemes, even when the working set of the algorithm fits in L2-cache. This is quite predictable, since both threads have the same requirements for computational resources because they execute the same code. This is an inherent limitation of SMT machines and is also discussed in [3, 14, 16].

The other two platforms, *Woodcrest* and *Opteron*, experience linear speedup in almost every case where the working set of the kernel fits in the cache. On the other hand, speedup decreases dramatically for almost every scheme when the working set exceeds the L2-cache size. In order to examine whether there is a bus contention that leads to that performance degradation, we collected results from several events logged by the processors' performance counters. It should be noted that these events were measured only during the actual execution of the

[1] Although *Opteron* is a NUMA machine, no NUMA-aware data allocation is performed in the context of this discussion, thus there still exists contention on the HyperTransport link of a single core. See Section 4.3.3 for a more detailed consideration of NUMA issues.

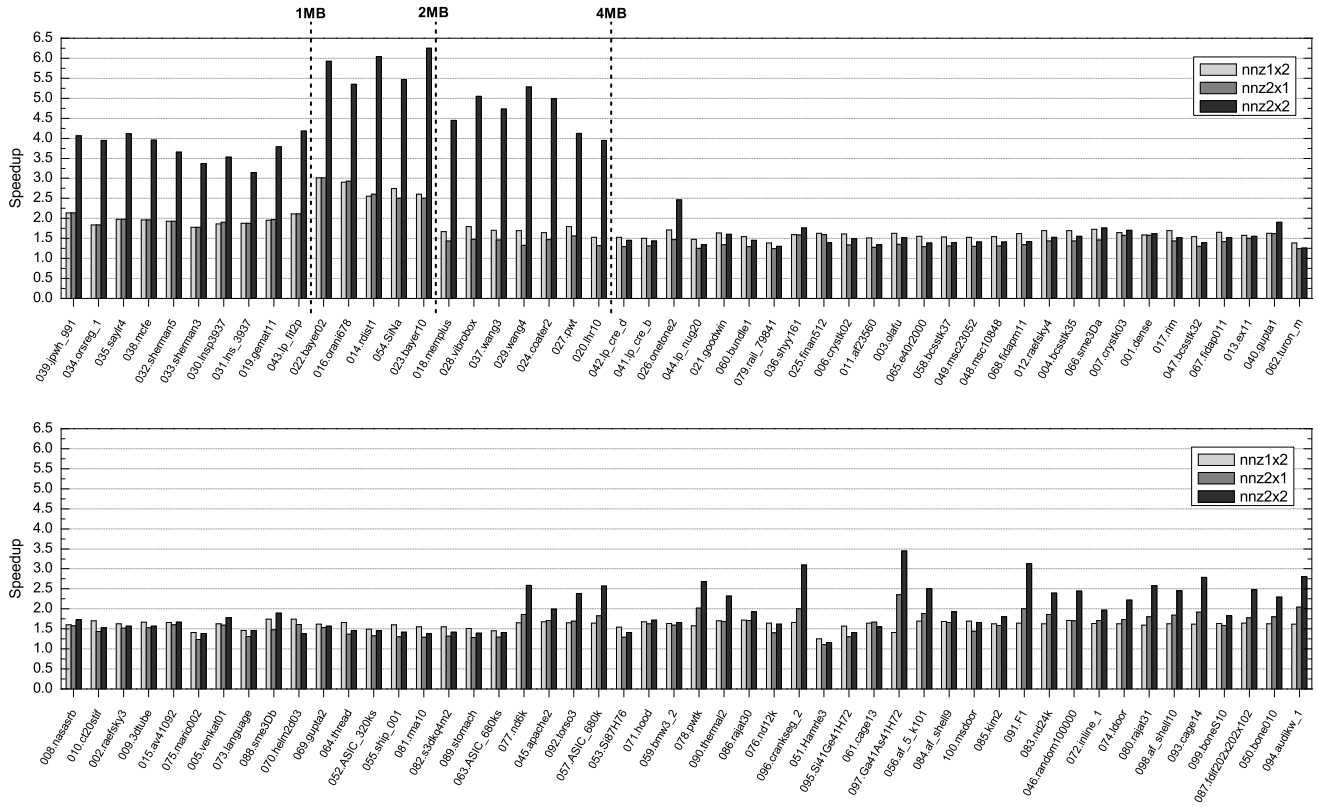
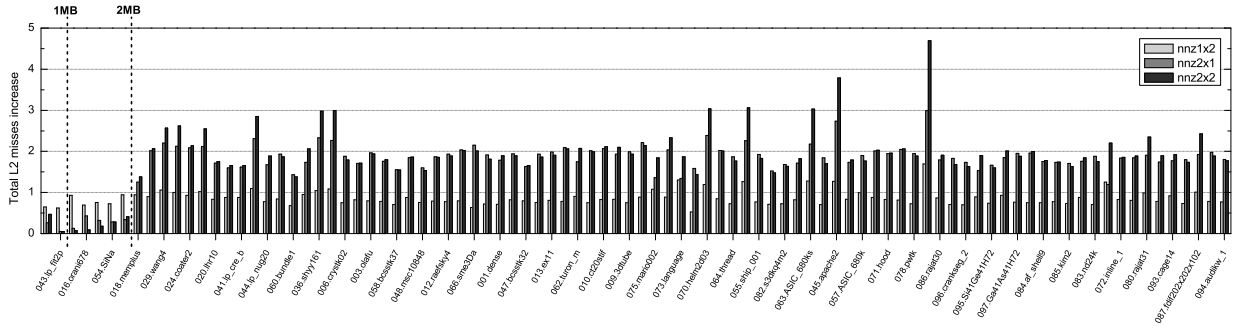


Figure 13: Speedup achieved on *Opteron* for every multithreading scheme. The vertical lines denote the effective cache size for the serial execution, the 1×2, 2×1, and 2×2 multithreading schemes, respectively.

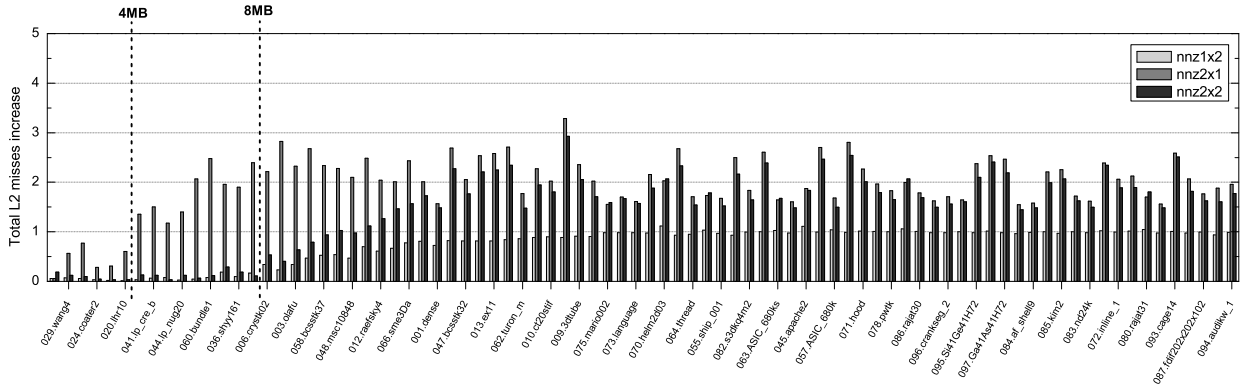
kernel—monitoring was turned off during the initialization phase.

At first, we measured the L2-cache misses on every processor. The normalized (over the serial case) results are depicted in Figure 14. Additionally, we have measured the average request bus latency on *Netburst*, which is the average time a memory request should wait on the Input-Output Queue (IOQ); the IOQ is the interface of the processor to the main memory subsystem. Larger values of this metric indicate that there exist bus contention, since memory requests should wait longer in order to gain access to the bus. More technically, in order to obtain this metric, we have divided the metric `IOQ_active_entries`, which counts the cycles where at least one request was pending on IOQ, by the metric `IOQ_allocations`, which is the total number of requests served through the IOQ. Figure 15 shows the increase over the serial case of average IOQ waiting times for each multithreading scheme considered.

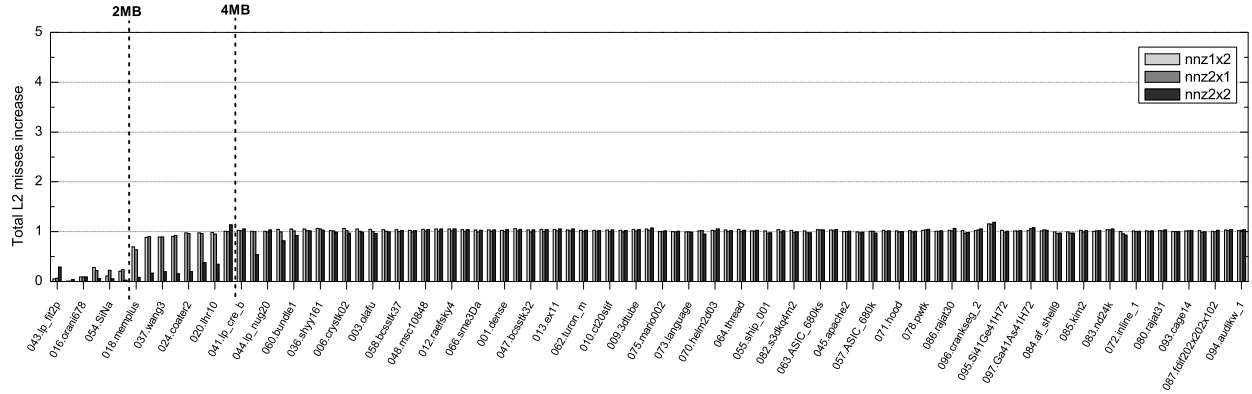
In order to quantify the bus contention problem on *Woodcrest*, we have used a different metric; namely, we have measured the bus utilization. Again in this case, we used primitive performance metrics provided by the processor to extract the desired metric. More accurately, we have divided the total number of bus cycles where a data transaction was in progress from whichever processing element (`BUS_DRDY_CLOCKS.ALL_AGENTS`) by the total number of bus cycles consumed during the execution of SpMxV (`CPU_CLK_UNHALTED.BUS`). This metric is presented in Figure 16. Figure 14



(a) *Netburst*.



(b) *Woodcrest*



(c) *Opteron*.

Figure 14: Increase of total L2-cache misses over the serial case.

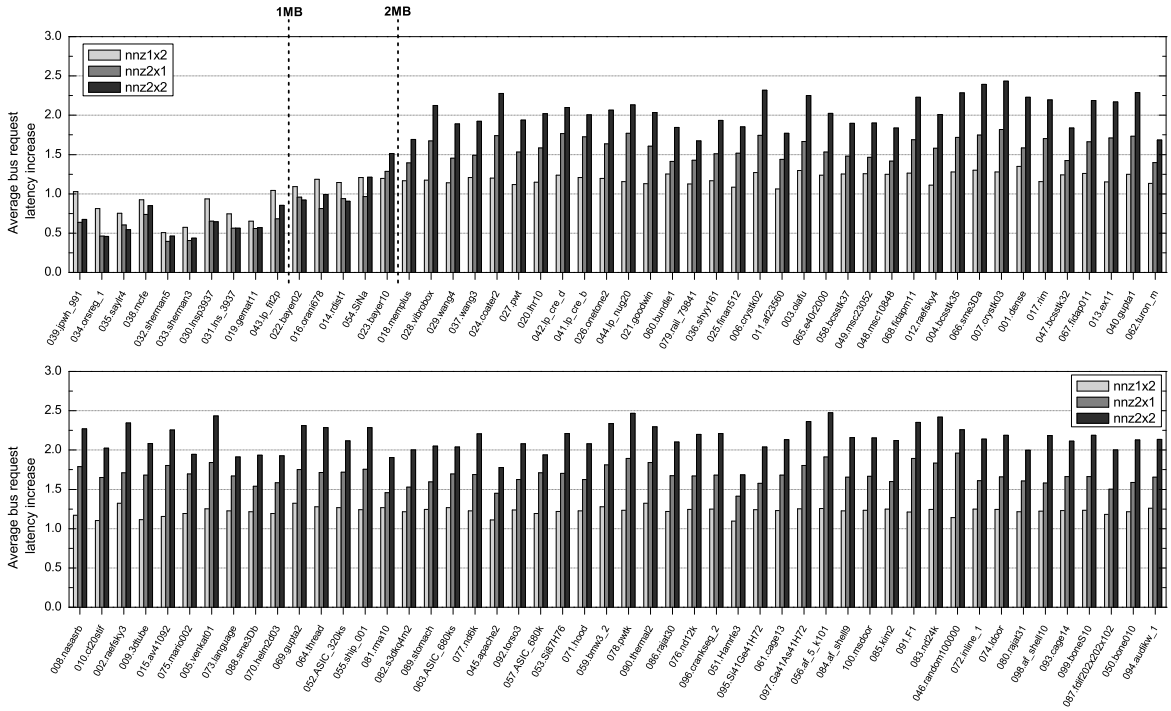


Figure 15: Increase of the total IOQ waiting time of a memory request on *Netburst*.

clarifies that when the working set of the algorithm is smaller than the effective cache size that a multithreading scheme "sees", the L2-cache misses are reduced relative to the serial version of the kernel. In this case as well, all multithreading schemes on every processor, except for the 1×2 and 2×2 schemes on *Netburst*, experience linear speedup. These schemes fail to scale on *Netburst* because the two threads have similar instruction mixtures and thus contend on shared resources inside the processor. The metrics of average bus latency on *Netburst* and bus utilization on *Woodcrest* presented on Figures 15 and 16 exhibit a similar behavior as the L2-cache miss rate.

To better contemplate the relation between the different multithreading schemes and the effective cache size, as well as their impact on performance, we consider the example of *Opteron*. When $ws < 2MB$, the working set of the algorithm fits in cache for every scheme, and thus, the kernel experiences almost linear speedup. When $2MB < ws < 4MB$, the working set does not fit in cache for the 1×2 and 2×1 schemes, but it still fits for the 2×2 scheme. Consequently, the first two schemes experience an increased amount of cache misses and lower performance, whereas the 2×2 scheme still achieves linear speedup. Finally, when $ws > 4MB$, cache misses increase, and performance drops for every scheme.

All of the above metrics, L2-miss rates, bus latency, and bus utilization, exhibit a similar behavior; when the matrices are too small to fit in the effective cache of a multithreading scheme, their values are almost equal to the serial case; for matrices with growing sizes, these metrics increase considerably. This fact along with the either way memory intensive nature of the algorithm confirms our initial assumption that the major problem of this kernel on

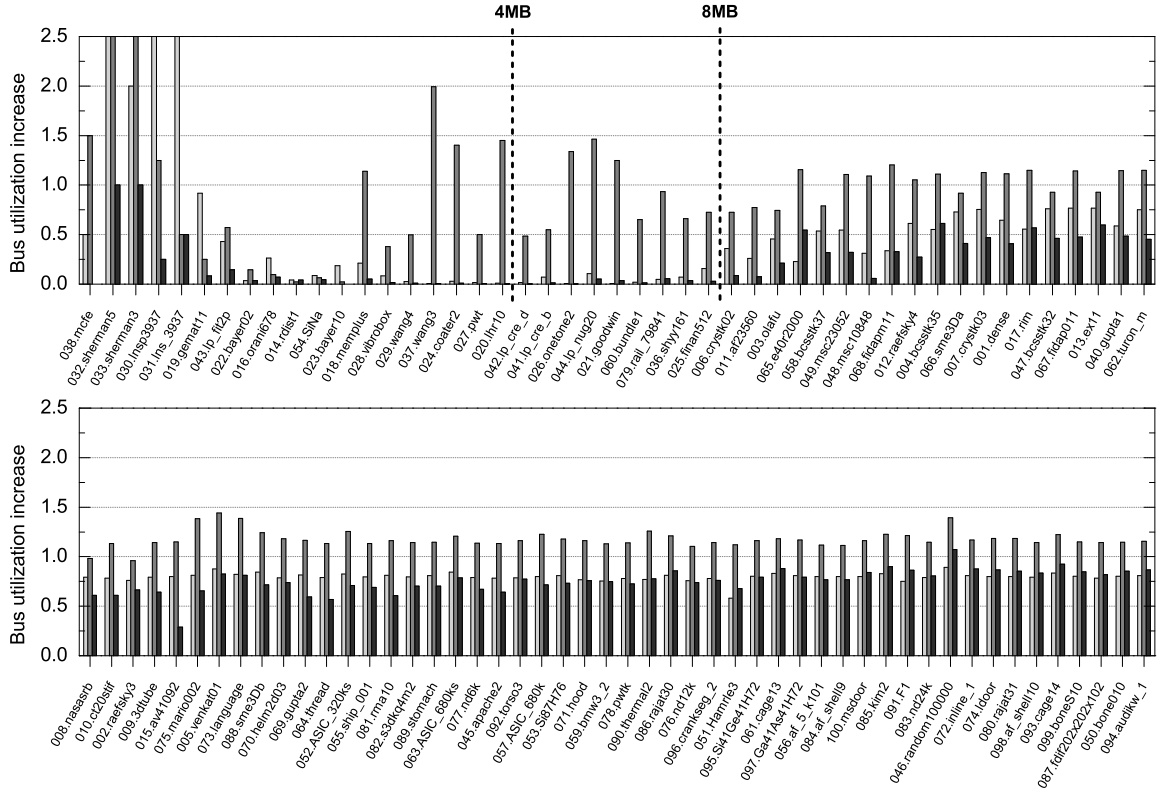


Figure 16: Increase in bus utilization on *Woodcrest*.

shared memory machines is the bus contention. It should also be noted here, that the increase in L2-cache miss rate relevant to the serial case for larger matrices is due to the bus contention as well. Large matrices lead to increased miss rates, and thus more main memory requests, which press harder the common bus, which in turn cannot service these requests in time. Consequently, subsequent load instructions also miss. Moreover, the contended bus hinders the hardware prefetcher from fetching useful data in time.

A special comment should be made for the case where a superlinear speedup is encountered when the working set is smaller than the effective cache size. Let cs_s be the effective cache size of the serial case, i.e., the physical cache size of the architecture, and cs_p the effective cache size of a multithreading scheme. If $0 < ws < cs_s$, then neither the serial nor the multithreaded version experience cache misses, thus an almost linear speedup is encountered. If $cs_s < ws < cs_p$, however, only the serial case experiences misses, since in the multithreaded one the working set has been split among threads, and thus, fits in the corresponding caches. As a result, this improvement of cache miss rate offers an extra boost to performance that adds to the already linear speedup, thus leading to superlinear behavior. Finally, though we did not have any performance metric that could reveal similar contention issues on the *Opteron* processor, there seems to be contention in the HyperTransport links, when no NUMA-aware data allocation is used. We further examine such an allocation scheme in the following section.

The shared cache as an additional benefit: For matrices whose working set does not fit in L2-cache, it can be observed from Table 10 that schemes which share the L2-cache provide better speedup than schemes which do not. For example, on *Woodcrest* the 1×2 scheme achieved much higher, and almost optimal speedup (1.96), than the 2×1 scheme. Moreover, this scheme achieved a superlinear speedup for the first 20 matrices in Figure 12, for which a 25%–90% reduction in total cache misses was encountered. For matrices that do not fit in cache, the 1×2 scheme achieved a maximum of 25% reduction of L2-cache misses over the 2×1 scheme. Similar behavior was also observed on *Netburst*, although the speedup for both cases when $ws > cs$ were rather small due to the inherent limitations of the HyperThreading technology.

4.3.3 NUMA architectures

Among the three hardware platforms under consideration, we expected *Opteron* to provide the best scalability due to the advantage that the NUMA architecture offers. Nonetheless, Figure 13 and Table 10 do not indicate any benefit from *Opteron*'s special architecture, which means that the NUMA characteristics of the architecture are not utilized effectively. In order to evaluate this assumption, we used the performance counters of the processor to measure the memory requests served by each memory controller (`DRAM_accesses` event). Indeed, the memory requests were quite unevenly distributed among the two memory controllers with the one controller serving 698 times more requests than the other for the 1×2 scheme and 452 times for the 2×2 scheme. That means, that almost all requests were served from a single controller, which was finally overwhelmed. This is graphically depicted in Figure 17.

Although the Linux kernel used for experiments supports NUMA architectures and always attempts to allocate pages on the local memory of each node, it failed to do so for the parallel version of the SpMxV kernel. This was anticipated, since the data allocation in our implementation happens in the main thread before any other thread is spawned. Consequently, all the necessary data is allocated on the local memory of a single node. In order to overcome this problem, we have implemented a "NUMA-aware" allocation of the algorithm's data structures. We used the `numa_alloc_onnode()` function of the `libnuma` library, so as to locally allocate the parts of `a`, `row_ptr`, `col_ind`, and `y` structures that are used by each thread. A copy of the input vector `x`, which is used equally from all threads, is allocated on every node's local memory in order to minimize remote memory accesses. It should be noted here, however, that the function used does not perform a strict allocation, i.e., it attempts to allocate the requested data on the specified node's local memory, but it does not guarantee that specific allocation; if a page cannot be allocated locally, it will be allocated on another node's memory.

The NUMA-aware data allocation offered a considerable performance improvement (Figure 18), especially for matrices with large working sets ($ws > cs$). The average speedup for matrices that do not fit in the effective cache reached an almost linear 1.96, instead of just 1.53 for the 2×1 scheme, and an impressive increase from 1.81 to 3.02

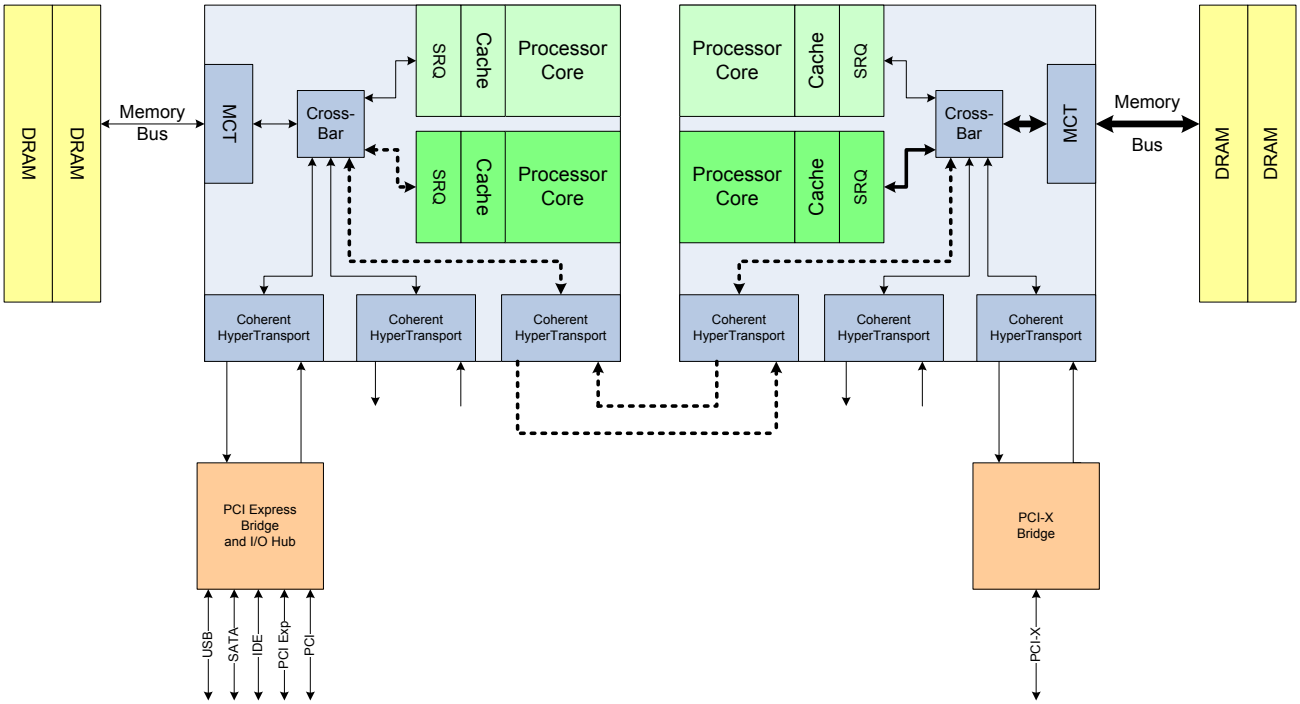


Figure 17: Contention of main memory requests on *Opteron*. All data are stored on a single memory node (right bank). As a result, one core needs to perform remote memory accesses (dashed line), which eventually lead to contention on the common memory controller and memory bus (bold line).

was observed for the 2×2 scheme. The 1×2 scheme, on the other hand, experienced a slight decrease from 1.61 to 1.55.

5 Optimization Guidelines

Summarizing the results of the preceding analysis, a number of optimization guidelines can be proposed. These guidelines, as a result of our extensive experimentation with the SpMxV kernel on modern commodity architectures, delineate our point of view of the most important performance bottlenecks and how these should be addressed effectively. The steering performance impediment that should drive any subsequent optimization at the first place is the memory intensity of the kernel. Secondly, one should take into consideration the computational part of the kernel. Specifically, we suggest the following for the single-threaded version of the kernel.

1. *Reduce as much as possible the working set of the algorithm.* Reducing the working set, e.g., by using 32-bit or 16-bit integers for the indexing structures of the matrix, by applying blocking schemes (as in [5, 12, 20, 26]) that effectively reduce the size of indexing structures, by applying compression (as in [28]), or by other means, will certainly increase the computation to memory operations ratio, thus alleviating the pressure on memory bus and give better chance to pending memory requests to be served in time.

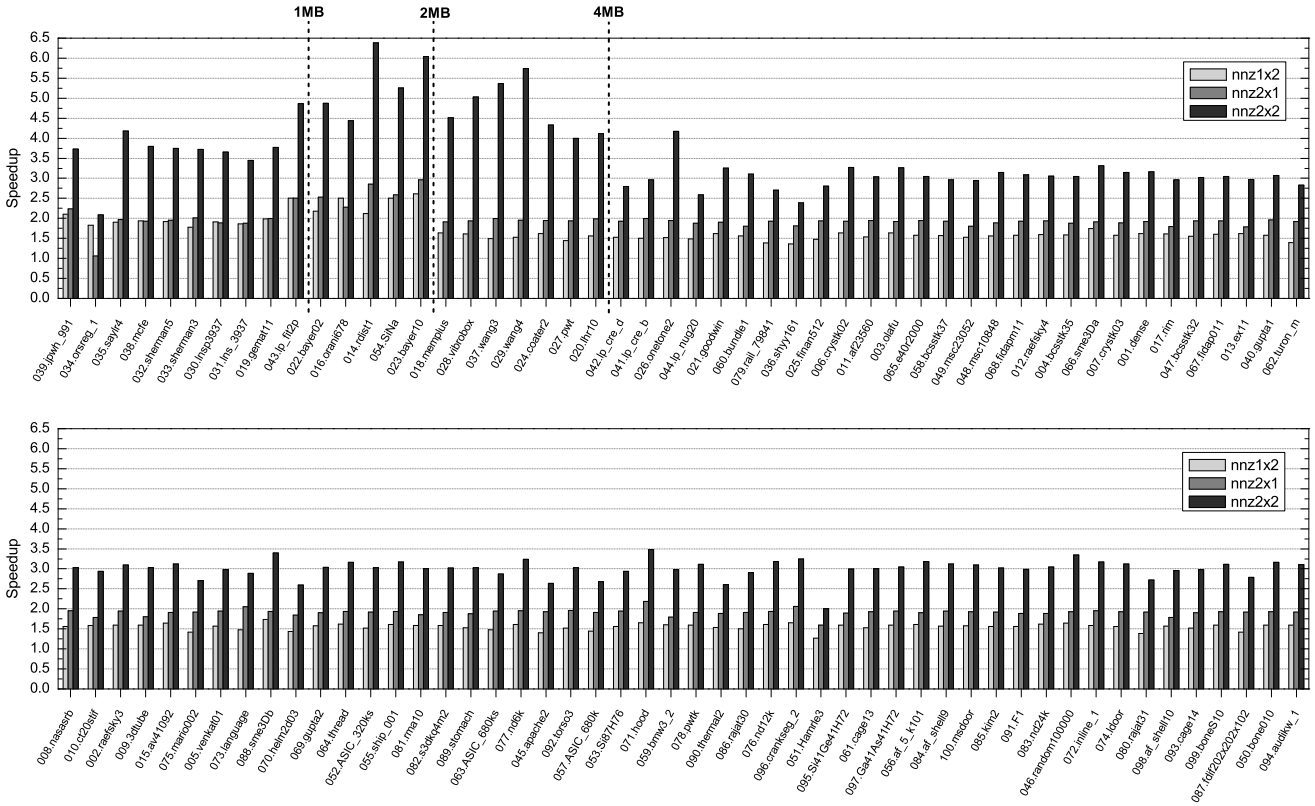


Figure 18: Obtained speedup from a "NUMA-aware" data allocation on *Opteron*.

2. *If you need padding, do it conservatively.* Some blocking schemes, which try to effectively reduce the working set of the algorithm, pad with zeros in order to artificially construct specific patterns which can be effectively computed. This padding could lead to working set increase and excessive useless computations that will ruin performance. Thus, the BCSR format used in [5, 12, 26] is expected to be beneficial only in the subset of matrices that contain many dense sub-blocks

3. *Use cache-reuse optimizations for irregular matrices only.* One needs to identify matrices with problematic access on the `x` vector and apply cache reuse optimizations only to them.

4. *Take into consideration the effect of short rows.* Some optimization approaches split the matrix into a sum of submatrices (as in [1, 26]). In this case one should take care that the submatrices do not fall into the category of matrices with short row lengths or even contain a large number of empty rows. Alternatively, one may insert an additional outer loop in the multiplication kernel (as in [20]). This may also incur significant overheads, especially in matrices with short rows.

5. *Reduce indirect memory referencing.* This could be achieved by exploiting regular structures within the matrix such as full diagonals (as in [1]) or dense subblocks (e.g., BCSR format as in [5, 12, 26]).

6. *Quantify the effect of hardware prefetching prior to applying software prefetching.* Modern architectures provide intelligent hardware prefetchers that can effectively predict access patterns and bring useful data in cache before any actual request from the processor. One could additionally utilize the performance monitoring hardware of the processor and examine whether there exist cache misses for a particular matrix that could be further reduced. If this is the case, software prefetching can be employed to prefetch data from the input vector `x`.

As far as the multithreaded version of the kernel is concerned, we suggest the following guidelines.

1. *Control, if possible, the way threads access the bus of an SMP machine.* It is apparent from the above experimental analysis that the main bottleneck for SpMxV on a SMP machine is the simultaneous access of all threads to the common bus. Thus, controlling the way threads are requesting bus resources would be beneficial to the overall performance. However, this task is not straightforward and involves a number of scheduling issues for arbitrating the threads' access to the common bus. This kind of optimization on SMP machines is a matter of our future research.

2. *Exploit any NUMA capabilities of your architecture.* The paradigm of *Opteron*which achieved a considerable performance improvement when a NUMA-aware data allocation scheme was used, is illuminating. The main advantage of NUMA machines over SMP is that they eliminate the contention on the common bus, thus exploiting the characteristics of such machine, while computing the memory intensive SpMxV kernel, could only be beneficial.

3. *Favor the use of common cache for shared thread data.* This is an implication that is not obvious. The irregularity of accesses on the input vector `x` leads to similar, though irregular, access patterns between different threads. Thus, a thread may benefit from its sibling's work running on the same core, which sometimes happens to bring a priori common data in L2-cache.

6 Conclusion – Future work

In this work, we have performed an extensive experimental evaluation of the SpMxV kernel for single and multi-threaded versions, on a variety of modern commodity architectures. SpMxV is a critical computational kernel and comprises the core part of a variety of scientific applications. However, this kernel has a set of inherent performance limitations, which, though discussed to some extent in the literature, were not deeply understood and quantified. In this paper, we took an in-depth look at the performance bottlenecks of this kernel using a set of metrics ranging from simple MFLOPS measures to advanced performance metrics obtained from modern processors' performance counters. These metrics provided us with a clear insight into the problems reported in the literature, and into the extent that these problems affect the actual performance of the kernel on modern architectures. As far as the

multithreaded version is concerned, we examined the effect of the common datapath from the main memory to the different processing elements (common bus or common memory controller), the effect of the shared cache, and the benefits that NUMA capabilities can provide to the kernel. The dominating problem of the SpMxV kernel on modern commodity architectures is the memory bottleneck. Thus, any optimization targeting the kernel should first focus at minimizing the memory traffic. When the memory traffic problem is attacked, the computational part of the kernel may become significant, thus, further optimizations targeting the computations could increase perforamance. Contrary to previous work performed on older platforms, modern microprocessors do not suffer from the irregular and indirect references to the input vector. However, these characteristics of the algorithm could still pose a performance bottleneck, but for a certain set of matrices with a close-to-random distribution of non-zero elements. An additional problem of the kernel, yet not dominant, is the presence of a large amount of very short rows, in which case the loop overhead will dominate the computational part of the kernel. The experience obtained from this in-depth experimentation was summarized as a number of optimization guidelines.

As a future work for the single-threaded version of the algorithm, we intend to evaluate existing storage formats that minimize the working set of the algorithm and propose novel ones that better achieve this goal. Optimization on the computational part of the kernel, e.g., vectorization, in combination with existing optimization techniques will also be examined and evaluated. Matters of index or data compression in order to minimize the working set of the algorithm comprise active research. Inventing and applying successful heuristics in order to select the best, in terms of SpMxV performance, storage method for a specific matrix is an additional future research aspect.

The multithreaded version of the kernel provides additional research prospects. We intend to evaluate advanced partitioning schemes and methods of assigning work to processing elements, either statically or dynamically, that could incorporate concepts such as loop overheads or cache sharing. We will also focus on multicore machines with more than four hardware threads in order to address the even more challenging problem of bus contention in that case, and we will investigate multithreading schemes that could arbitrate the way threads are accessing the common bus so as to maximize its utilization. Finally, we will implement, evaluate, and optimize the kernel on more sophisticated architectures such as the Cell processor and general-purpose GPUs, as well as consider alternative programming models, such as streaming programming. Effectively porting the SpMxV kernel to such architectures and evaluating its behavior is a particularly interesting research topic.

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, November 1992. IEEE.

[2] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker,

J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, December 18 2006.

[3] E. Athanasaki, N. Anastopoulos, K. Kourtis, and N. Koziris. Exploring the performance limits of simultaneous multithreading for memory intensive applications. *The Journal of Supercomputing*. (to appear).

[4] 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.

[5] 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.

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

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

[8] 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.

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

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

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

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

[13] H. Kotakemori, H. Hasegawa, T. Kajiyama, A. Nukada, R. Suda, and A. Nishida. Performance evaluation of parallel sparse matrix-vector products on SGI Altix3700. In *1st International Workshop on OpenMP (IWOMP)*, Eugene, OR, USA, June 2005.

[14] J. L. Lo, S. J. Eggers, J. S. Emer, Henry M. Levy, Rebecca L. Stamm, and Dean M. Tullsen. Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading. *ACM Trans. Comput. Syst*, 15(3):322–354, 1997.

[15] 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.

[16] N. Mitchell, L. Carter, J. Ferrante, and D. Tullsen. Instruction level parallelism vs. thread level parallelism on simultaneous multi-threading processors. In *Proceedings of Supercomputing'99 (CD-ROM)*, Portland, OR, November 1999. ACM SIGARCH and IEEE.

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

[18] 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.

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

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

[21] 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.

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

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

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

[25] 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, November 2002.

[26] 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.

[27] 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.

[28] 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.

[29] S. Williams, L. Oilker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *Supercomputing'07*, Reno, NV, November 2007. (to appear).