A comparative study of blocking storage methods for sparse matrices on multicore architectures

Vasileios Karakasis, Georgios Goumas, Nectarios Koziris
Computing Systems Laboratory
National Technical University
Athens, Greece
HiPEAC members
Email: {bkk,goumas,nkoziris}@cslab.ece.ntua.gr

***Abstract*—Sparse Matrix-Vector multiplication (SpMV) is a very challenging computational kernel, since its performance depends greatly on both the input matrix and the underlying architecture. The main problem of SpMV is its high demands on memory bandwidth, which cannot yet be abudantly offered from modern commodity architectures. One of the most promising optimization techniques for SpMV is blocking, which can reduce the indexing structures for storing a sparse matrix, and therefore alleviate the pressure to the memory subsystem. In this paper, we study and evaluate a number of representative blocking storage formats on a set of modern microarchitectures that can provide up to 64 hardware contexts. The purpose of this paper is to present the merits and drawbacks of each method in relation to the underlying microarchitecture and to provide a consistent overview of the most promising blocking storage methods for sparse matrices that have been presented in the literature.**

***Keywords*-sparse matrix-vector multiplication; blocking; performance evaluation**

I. INTRODUCTION

Sparse Matrix-Vector Multiplication (SpMV) is one of the most important and widely used scientific kernels arising in a variety of scientific problems. The SpMV kernel poses a variety of performance issues both in single and multicore configurations [4], [12], [16], which are mainly due to the memory-intensive nature of the SpMV algorithm. To this end, a number of optimization techniques have been proposed, such as register and cache blocking [5], [6], compression [8], [15], and column or row reordering [10]. The main purpose of all these techniques is to either reduce the total *working set* of the algorithm, i.e., the total amount of data that needs to be fetched from main memory, or create more regular memory access patterns (reordering techniques). Blocking methods fall mainly into the first category, since their main advantage is that they can considerably reduce the total working set of the algorithm, thus applying less pressure to the memory subsystem. By alleviating the memory pressure, blocking techniques leave more space for optimizations targeting the computational part of the kernel as well, such as loop unrolling and vectorization [7], which can further improve performance.

In general, blocking storage formats for sparse matrices can be divided into three categories: (a) storage formats that apply zero-padding aggressively in order to construct full blocks, (b) storage formats that decompose the original matrix into k submatrices, where the $k-1$ submatrices use blocking without padding and the k-th matrix is stored in standard Compressed Sparse Row (CSR) format [2], and (c) storage formats that use variable size blocks without padding, but at the expense of additional indexing structures. Blocked Compressed Sparse Row (BCSR) [6] and Unaligned BCSR [14] formats are typical examples of the first category. Both formats try to exploit small two-dimensional dense subblocks inside the sparse matrix with their main difference being that BCSR imposes a strict alignment to its blocks at specific row- and column-boundaries. Agarwal et al. [1] decompose the input matrix by extracting regular common patterns, such as dense subblocks and partial diagonals. Similarly, Pinar and Heath [10] decompose the original matrix into two submatrices: a matrix with horizontal one-dimensional dense subblocks without padding and matrix in CSR format containing the remainder elements. Pinar and Heath [10] present also a one-dimensional variable-sized blocking storage formats, while in [11] the Variable Blocking Row (VBR) format is presented, which constructs two-dimensional variable blocks, at the cost of two additional indexing structures. It is apparent from the above discussion that there is a variety of different blocking storage formats for sparse matrices that have been proposed so far. However, to our best knowledge, there has not yet been any study that comparatively presents and evaluates the different blocking formats on modern multicore architectures, in such a way that would expose the advantages and disadvantages of each method.

Hence, the contribution of this paper can be summarized in the following: (a) we present a consistent overview of the different blocking storage formats for sparse matrices that have been proposed in the literature so far and attempt a categorization of them according to their structure and construction principles (e.g., the use or not of padding), (b) we implement five representative blocking storage for-

mats from the different categories, which we evaluate on a set of modern multicore architectures, and (c) we show that blocking is a promising sparse matrix optimization for multicores, but the prediction of the best storage format—and the best block shape for fixed size blocking methods—for a specific matrix and architecture is quite complex problem, which becomes even harder, as the number of threads increases, since the single-threaded performance can provide only a rough guidance for the majority of matrices.

The rest of the paper is organized as follows: Section II presents and discusses the different blocking storage formats for sparse matrices. Section III discusses the performance issues involved when applying blocking to SpMV, Section IV presents the experimental evaluation of the different blocking storage formats and discusses the results, and Section V concludes the paper.

II. An overview of blocking storage formats

In this section, we consider blocking storage formats for sparse matrices that can be applied to an arbitrary sparse matrix, as opposed to storage formats which assume a special nonzero elements pattern, e.g., tri-band, diagonal sparse matrices, etc. Before proceeding with the description of any blocking format, we describe briefly the standard sparse matrix storage format, namely the Compressed Sparse Row (CSR) format [2]. CSR uses three arrays (Fig. 2a) to store a $n \times m$ sparse matrix with nnz non-zero elements: an array val of size nnz to store the non-zero elements of the matrix, an array col_ind of size nnz to store the column indices of every non-zero element, and an array row_ptr of size $n+1$ to store pointers to the first element of each row in the val array.

Blocking storage formats for sparse matrices can be divided into two large categories: (a) formats with fixed-size blocks that employ aggressively padding with zeros to construct full blocks and (b) formats that do not pad at all. The second category can be further divided depending on the strategy used to avoid padding. There have been proposed two strategies to avoid padding in the literature: (a) decompose the original matrix into two or more matrices, where each matrix contains dense subblocks of some common pattern (e.g., rectangular, diagonal blocks, etc.), while the last matrix contains the remainder elements in a standard sparse storage format [1], and (b) use variable size blocks [10], [11]. In the following, we will present each blocking method in more detail.

A. Blocking with padding

Blocked Compressed Sparse Row: The most prominent blocking storage format for sparse matrices that uses padding is the Blocked Compressed Sparse Row (BCSR) format [6]. BCSR is the blocked version of CSR, which instead of storing and indexing single nonzero elements, it stores and indexes two-dimensional fixed-size blocks with at least one nonzero element. BCSR will use padding in order to construct full blocks. Specifically, BCSR uses three arrays (Fig. 2b) to store a sparse matrix: (a) $bval$, which stores linearly in row-wise or column-wise order the values of all blocks present in the matrix, (b) $bcol_ind$, which stores the block-column indices, and (c) $brow_ptr$, which stores pointers to the first element of each block row in $bval$. Another property of BCSR is that it imposes a strict alignment to its blocks: each $r \times c$ block should be aligned at r row- and c column-boundaries, i.e., a $r \times c$ block should always start at the position (i, j), such that $\mathrm{mod}(i, r) = 0$ and $\mathrm{mod}(j, c) = 0$. This restriction leads generally to more padding (see Fig. 1), but it has two main advantages: it facilitates the construction of the BCSR format and it can have a positive impact on performance, when using vectorization [7]. A variation of BCSR is the Unaligned BCSR (UBCSR) [14], which relaxes the above restriction, in order to avoid padding.

Blocked Compressed Sparse Diagonal: The Blocked Compressed Sparse Diagonal (BCSD) format is analogous to BCSR, but exploits small diagonal subblocks inside the matrix. Like BCSR, it also uses three arrays—$bval$, $bcol_ind$, and $brow_ptr$—to store the input matrix (Fig. 2c), but in this case $bval$ stores the elements of each diagonal subblock, while $bcol_ind$ continues to store the column index of each subblock. BCSD also imposes an alignment restriction as to where each diagonal block can start. Specifically, each diagonal block of size b should start at the position (i, j), such that $\mathrm{mod}(i, b) = 0$. This restriction effectively separates the matrix into block rows or segments of size b (see Fig. 1). The $brow_ptr$ array then stores pointers to the first element of each segment in the the $bval$ array. BCSD also uses padding to construct full blocks.

A version of this format has been initially proposed in [1] as part of a decomposed method, which extracted common dense subblocks from the input matrix. A similar format, called RSDIAG, is also presented in [13], but it maintains an additional structure that stores the total number of diagonals in each segment. This format was also part of a decomposed method.

B. Blocking without padding

Decomposed matrices: A common practice to avoid padding is to decompose the original input sparse matrix into k smaller matrices, where the first $k-1$ matrices consist of elements extracted from the input matrix that follow a common pattern, e.g., rectangular or diagonal dense subblocks, while the k-th matrix contains the remainder elements of the input matrix, stored in a standard sparse matrix storage format. In this paper, we present and evaluate the decomposed versions of BCSR (BCSR-DEC) and BCSD (BCSD-DEC). For these formats $k = 2$, i.e., the input matrix is split into only two submatrices, the first containing

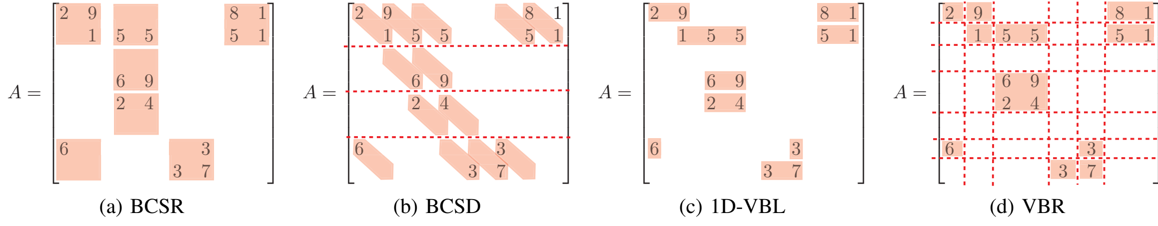


Figure 1: How the different blocking storage formats split the input matrix into blocks.

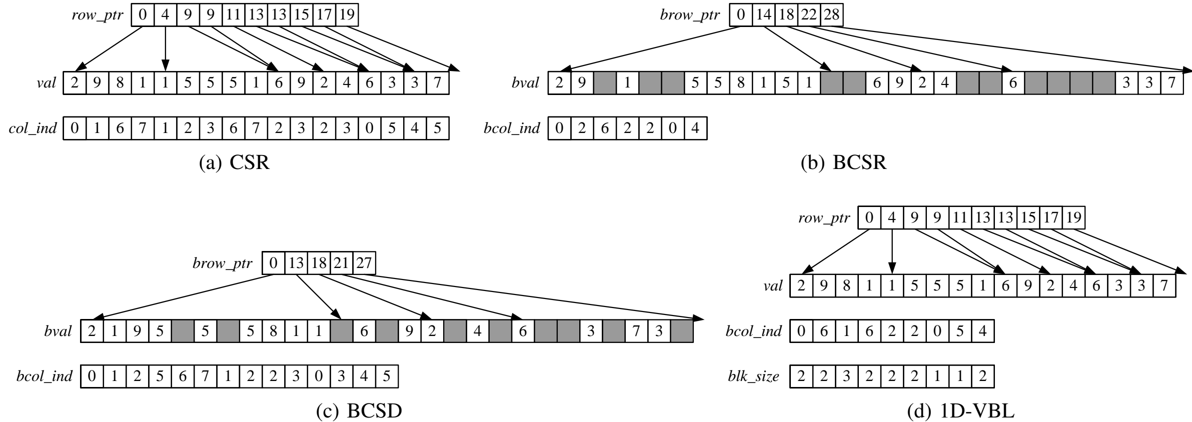


Figure 2: Data structures used by different blocking storage formats.

full blocks without padding and the second the remainder elements stored in CSR format.

Variable size blocks: An alternative solution to avoid padding when using blocking is to use variable size blocks. Two methods have been proposed in the literature that use variable-size blocks: one-dimensional Variable Block Length (1D-VBL) [10], which exploits one-dimensional horizontal blocks, and Variable Block Row (VBR) [11], which exploits two-dimensional blocks. 1D-VBL uses four arrays (Fig. 2d) to store a sparse matrix: val, row_ptr, $bcol_ind$, and blk_size. The val and row_ptr arrays serve exactly the same purpose as in CSR, while $bcol_ind$ stores the starting column of each block and blk_size stores the size of each block. VBR is more complex and it actually partitions the input matrix horizontally and vertically, such that each resulting block contains only nonzero elements. In order to achieve this, it uses two additional arrays compared to CSR to store the start of each block-row and block-column.

Figure 1 summarizes how each block format forms blocks from neighboring elements.

III. Performance issues of blocking

The SpMV kernel in modern commodity microarchitectures is in most cases bound from memory bandwidth [4]. Although there exist other potential performance problems in this kernel, such as indirect references, irregular accesses, and loop overheads, the bottleneck in the memory subsystem is categorized as the most important SpMV performance problem in both single- and multithreaded configurations [4], [7], [16]. The great benefit of blocking methods is that they can substantially reduce the working set of the SpMV algorithm, thus reducing the demands on memory bandwidth and alleviating the pressure to the memory subsystem. The reduction of the working set is mainly due to the fact that blocking methods maintain a single index for each block column instead of an index for each element. Therefore, the col_ind structure of CSR, which consists almost half of the working set of the algorithm, can be significantly reduced. A consequence of reducing the memory bandwidth demands of the SpMV kernel is that the computational part is then more exposed, since it consists a larger portion of the overall execution time. Therefore, optimizations targeting the computational part can have significant impact on performance [7]. However, each blocking method has its own advantages and pitfalls, which are summarized in the following.

Fixed size blocking with padding: The main advantage of these methods is that they allow for efficient implementations of block-specific kernels since the size—and in most cases the alignment—of blocks are known a priori. However, if the nonzero elements pattern of the input matrix is rather irregular, these methods lead to excessive padding,

overwhelming any benefit from the reduction of the size of col_ind structure. Additionally, the selection of the most appropriate block is not straightforward, especially if vectorization is used, since instruction dependencies and hardware limitations of the vector units of modern commodity architectures can significantly affect the overall performance [7].

Decomposed methods: Although decomposed methods avoid padding, they suffer from three problems: (a) there is no temporal or spatial locality (except in the input vector) between the different k SpMV operations, (b) additional operations are needed to accumulate the partial results to the final output vector, and (c) the remainder CSR matrix will have very short rows, which can further degrade the overall performance due to loop overheads and cache misses on the input vector [4].

Variable size blocking: The variable size blocking has also the advantage of not employing padding to construct blocks, but at the expense of additional data structures. Therefore, any gain in the final working set of the algorithm by eliminating padding and reducing the col_ind structure can be overwhelmed from the size of the additional data structures. In addition, the extra level of indirection that variable size blocking methods introduce can further degrade performance.

IV. Experimental evaluation

Matrix suite: The matrix suite used for our experiments is a set of sparse matrices obtained from Tim Davis' sparse matrix collection [3]. We made an effort to include matrices from different application domains, which could reveal the capabilities and possible shortcomings of the different blocking storage formats under evaluation. The matrix suite consists of 30 matrices (Tab. I). Matrices #1 (dense) and #2 (random) are special purpose matrices, while the rest 28 are divided into two large categories: matrices #3–#16 come from problems without an underlying 2D/3D geometry, while matrices #17–#30 have a 2D/3D geometry. In general, sparse matrices with an underlying geometry exhibit more regular structure, so we expect blocking methods to perform better for these matrices. Finally, all selected matrices have large enough working sets (>25 MiB in CSR format), so that none of them fits in the processor's cache.

System platforms and experimental process: For our experimental evaluation we used three representative modern microarchitectures, namely, Intel Core (*Harpertown*), Intel Nehalem (*Nehalem*), and Sun Niagara2 (*Niagara2*). The actual characteristics of each architecture are presented in Tab. II. Nehalem is the latest microarchitecture from Intel, features integrated memory controllers supporting NUMA, and each core is SMT capable by using the HyperThreading technology. Thus, for the dual processor configuration used for our experiments (*Nehalem*), this leads to a total of 16 threads. Niagara2 is an eight core SMT processor with eight hardware contexts per core, thus leading to a maximum of 64 concurrent threads. However, the implementation of each Niagara2's core is simple and straightforward compared to the rather sophisticated Intel cores and lacks also the huge L2 and L3 caches of the Intel processors. Therefore, Niagara2's performance relies on hiding main memory latency by using a large number of threads. All platforms ran GNU/Linux, kernel version 2.6.26, and all programs were compiled using `gcc`, version 4.3, with the highest level of optimizations (`-O3`). For the evaluation of each blocking storage format, we ran 100 consecutive SpMV operations using randomly generated input vectors. Finally, for the case of *Niagara2*, we omitted matrices #10, #11, #12, and #15 from our evaluation, since the standard single-threaded loading and transformation processes that we employed for all platforms were extremely time consuming in the case of these matrices on *Niagara2*. Nevertheless, this omission is not significant and cannot alter the overall conclusions.

Implementation details: We have implemented five different blocking storage formats: two with fixed size blocks using padding (BCSR and BCSD), two decomposed with fixed size blocks (BCSR-DEC and BCSD-DEC), and one with variable size blocks (1D-VBL). We also implemented the standard CSR format, in order to have a common baseline. We used four-byte integers for the indexing structures of every format, and one-byte entries for the additional data structure of 1D-VBL, which contains the block sizes. This restricts the number of maximum elements per block to 255, but in the rare case a greater block is encountered, it is split into 255-element chunks. For the fixed size blocking methods, we used blocks with up to eight elements, and we have implemented a fully unrolled block-specific multiplication routine for each particular block. We did not use larger blocks, since preliminary experiments showed that such blocks cannot offer any speedup over standard CSR.

For the multithreaded versions of the blocked kernels we used the native POSIX threads library (NPTL, version 2.7). Threads are assigned work by splitting the input matrix rowwise in as many portions as the number of threads. We apply a static load balancing scheme, according to which we split the input matrix, such that each thread is assigned the same number of nonzeros. For the case of methods with padding, the load balancing scheme accounts also for the extra zero elements used for padding. This scheme proved to provide a fair distribution of work with a less than 5% difference in workload of threads in any case.

Evaluation of blocking storage formats: In order to examine how beneficial blocking can be on modern microarchitectures, Figure 3 plots the mininum, maximum, and average speedup over CSR of all blocking methods (including different blocks for fixed size blocking) per matrix basis for the single-threaded implementation for each microarchitecture on our experimental setup. This figure presents the 'big picture' of blocking on modern architectures and some interesting observations can be made:

Matrix	Domain	# rows	# nonzeros	ws (MiB)	Matrix	Domain	# rows	# nonzeros	ws (MiB)
01.dense	special	2,000	4,000,000	30.54	16.bone010	Other	986,703	36,326,514	288.44
02.random	special	100,000	14,977,726	115.42	17.kkt_power	Power	2,063,494	8,130,343	121.05
03.cfd2	CFD	123,440	1,605,669	24.95	18.largebasis	Opt.	440,020	5,560,100	45.01
04.parabolic_fem	CFD	525,825	2,100,225	34.05	19.TSOPF_RS	Opt.	38,120	16,171,169	123.81
05.Ga41As41H72	Chemistry	268,096	9,378,286	74.62	20.af_shell10	Struct.	1,508,065	27,090,195	223.94
06.ASIC_680k	Circuit	682,862	3,871,773	37.35	21.audikw_1	Struct.	943,695	39,297,771	310.62
07.G3_circuit	Circuit	1,585,478	4,623,152	76.59	22.F1	Struct.	343,791	13,590,452	107.62
08.Hamrle3	Circuit	1,447,360	5,514,242	58.63	23.fdiff	Struct.	4,000,000	27,840,000	258.18
09.rajat31	Circuit	4,690,002	20,316,253	208.67	24.gearbox	Struct.	153,746	4,617,075	71.04
10.cage15	Graph	5,154,859	99,199,551	815.82	25.inline_1	Struct.	503,712	18,660,027	148.13
11.wb-edu	Graph	9,845,725	57,156,537	548.75	26.ldoor	Struct.	952,203	23,737,339	192.00
12.wikipedia	Graph	3,148,440	39,383,235	336.50	27.pwtk	Struct.	217,918	5,926,171	47.71
13.degme	Lin. Prog.	659,415	8,127,528	65.94	28.thermal2	Other	1,228,045	4,904,179	51.47
14.rail4284	Lin. Prog.	1,096,894	1,000,000	90.31	29.nd24k	Other	72,000	14,393,817	110.64
15.spal_004	Lin. Prog.	321,696	46,168,124	353.54	30.stomach	Other	213,360	3,021,648	25.50

Table I: Matrix suite. The working set (ws) column represents the working set of the matrix stored in CSR format.

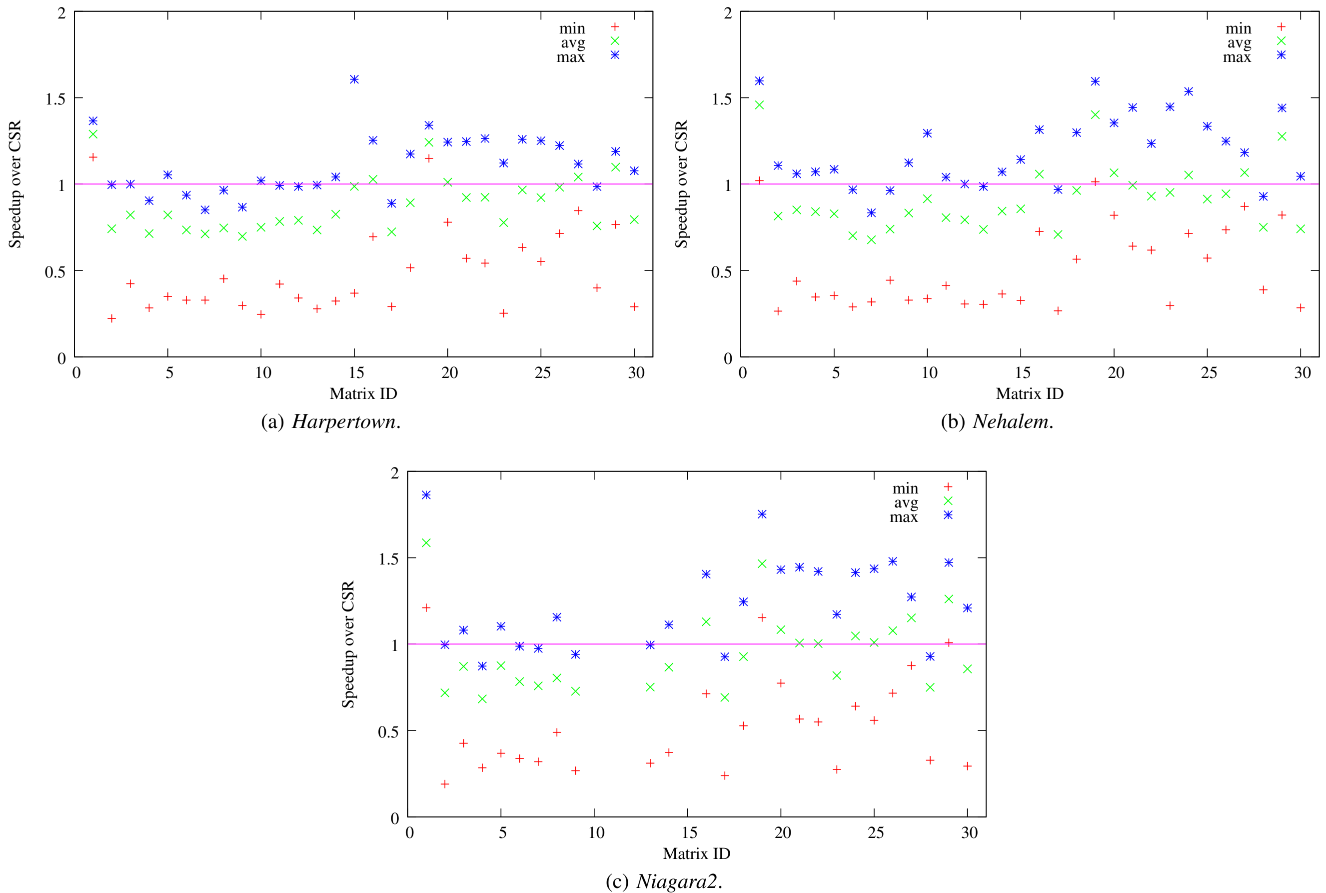
Characteristic	*Harpertown*	*Nehalem*	*Niagara2*
Processor Name	Intel Core 2 Xeon Quad E5405 (Harpertown)	Intel Xeon X5560 (Gainestown)	UltraSPARC T2 (Niagara2)
Core clock frequency	2.00 GHz	2.80 GHz	1.2 GHz
Cores	4	4	8
Hardware contexts per core	1	2	8
L1 cache	32 KiB I-cache, 32 KiB D-cache, 8-way both per core	32 KiB I-cache (4-way), 32 KiB D-cache (8-way) per core	16 KiB I-cache, 8 KiB D-cache per core
L2 cache	3 MiB unified cache (24-way) per two cores	256 KiB unified cache (8-way) per core	4 MiB unified cache (16-way), 8 banks
L3 cache	–	8 MiB unified cache (16-way), banked	–
Memory interface	1333 MHz FSB	3× Integrated DDR3 memory controllers 1333 MT/s	4× dual channel memory controllers
Multiprocessor configuration	2 processors, SMP	2 processors, NUMA	–

Table II: Experimental platforms.

Architecture	BCSR		BCSR-DEC		BCSD		BCSD-DEC		1D-VBL
	min	max	min	max	min	max	min	max	
Harpertown	0.55	1.04	0.94	1.10	0.85	0.97	0.98	1.03	0.80
Nehalem	0.57	1.14	0.89	1.18	0.90	1.04	1.01	1.10	0.74
Niagara2	0.54	1.11	0.97	1.17	0.88	1.03	1.01	1.12	1.05

Table III: Average of minimum and maximum performance of each format over all matrices (one thread, double-precision).

- Blocking can be as good as providing more than 50% performance improvement over CSR, but as poor as leading to more than 70% performance degradation in some cases, if not used with caution.
- Selecting a blocking method and a corresponding block without any prior knowledge of the input matrix and the underlying architecture should lead to lower performance than standard CSR for the majority of matrices, especially for *Harpertown* and *Nehalem*.
- Matrices that come from a problem domain with an underlying 2D/3D geometry (matrices with IDs larger than #15) are good candidates for blocking and can gain more than 20% in SpMV performance over standard CSR. However, the average performance of blocking for these matrices can still be lower than CSR's performance.
- On *Nehalem* and *Niagara2* microarchitectures blocking performs better than on the *Harpertown*. For the case of *Nehalem*, this is mainly due to the much higher attained memory bandwidth (≈10 GiB/s instead of ≈3.5 GiB/s of *Harpertown* according to the STREAM [9] benchmark). Therefore, the impact of the gain in the total working set of SpMV, which blocking generally offers, is greater in the case of *Nehalem*. For the *Niagara2* case, on the other hand, the average benefit of blocking over standard CSR is mainly due to the fact that the less sophisticated execution core of *Niagara2* cannot effectively hide some performance issues of CSR (e.g., cache misses on input vector, loop overheads, indirect references), as is the case of Intel processors [4]. Therefore, since blocking tackles these problems in a certain degree by providing better spatial locality on input vector, reduced loop overheads and indirect references through the fully unrolled block multiplication code, the



(a) *Harpertown*. (b) *Nehalem*. (c) *Niagara2*.

Figure 3: Overview of blocking. The minimum, maximum, and average speedup over standard CSR per matrix for all blocking methods (one thread, double-precision).

impact of blocking over standard CSR becomes more apparent.

Table III provides a per method look to the single-threaded execution behavior of blocking. Specifically, it presents the mininum and maximum speedup of each method relative to CSR averaged over the 30 matrices of our matrix suite. This table reveals also some interesting characteristics of blocking methods:

- The performance of methods that employ padding aggressively (BCSR, BCSD) can vary significantly. Especially in the case of BCSR, which uses two-dimensional blocks, this variation can reach up to 50% of the baseline CSR performance. It is obvious now that the mininum performance points depicted in Fig. 3 are due to the very poor BCSR performance, when selecting improper blocks.
- Decomposed methods present more stable behavior across the matrix suite and better average maximum speedups. Decomposed methods, therefore, can be considered as a safe blocking solution, when adequate knowledge of the input matrix structure and the underlying architecture is not available. Even an improper block selection can lead to a moderate 10% performance degradation from CSR on average. We should also note here that the highest average maximum speedups achieved by BCSR-DEC do not necessarily mean that it achieved the absolutely highest performance on the majority of matrices (see Fig. 4 and subsequent discussion).
- The variable block length method, 1D-VBL, is not very competitive for the single-threaded configuration in any of the considered microarchitectures, however, this is not the case as the number of threads increases, as it will be shown in the following.

Table IV presents in detail the storage method that achieved the best performance for each microarchitecture and for both double- and single-precision arithmetic. The results presented in this table are from multithreaded execution using the maximum available hardware thread contexts provided by the underlying microarchitecture, i.e., eight threads

Matrix	Harpertown		Nehalem		Niagara2	
	dp	sp	dp	sp	dp	sp
01.dense	1D-VBL	BCSR-DEC(7,1)	BCSR-DEC(2,4)	BCSR-DEC(2,4)	BCSR(8,1)	BCSR(8,1)
02.random	CSR	CSR	BCSD-DEC(6)	CSR	BCSR-DEC(1,2)	CSR
03.cfd2	BCSD-DEC(6)	BCSR(2,1)	BCSD-DEC(4)	BCSR-DEC(2,1)	BCSD-DEC(4)	BCSD(4)
04.parabolic_fem	BCSD-DEC(8)	BCSD-DEC(8)	CSR	BCSD-DEC(8)	CSR	BCSR(3,1)
05.Ga41As41H72	1D-VBL	1D-VBL	1D-VBL	BCSR-DEC(2,1)	BCSD-DEC(6)	BCSD-DEC(4)
06.ASIC_680k	BCSR-DEC(7,1)	BCSR-DEC(5,1)	1D-VBL	BCSR-DEC(6,1)	BCSR-DEC(1,8)	BCSR-DEC(1,8)
07.G3_circuit	1D-VBL	BCSD-DEC(8)	CSR	BCSD(2)	CSR	CSR
08.Hamrle3	BCSR(2,2)	BCSR(2,2)	1D-VBL	BCSR(2,1)	BCSR(2,2)	BCSR(2,2)
09.rajat31	1D-VBL	1D-VBL	BCSR-DEC(2,2)	CSR	CSR	CSR
10.cage15	BCSD-DEC(4)	BCSD-DEC(4)	CSR	CSR		
11.wb-edu	1D-VBL	1D-VBL	1D-VBL	BCSR-DEC(3,1)		
12.wikipedia-20061104	CSR	CSR	CSR	BCSR-DEC(3,1)		
13.degme	CSR	BCSR-DEC(1,2)	CSR	BCSR-DEC(6,1)	CSR	CSR
14.rail4284	BCSR-DEC(1,6)	1D-VBL	BCSR-DEC(1,8)	1D-VBL	BCSR-DEC(1,6)	BCSR-DEC(1,6)
15.spal_004	1D-VBL	1D-VBL	1D-VBL	1D-VBL		
16.bone010	1D-VBL	1D-VBL	1D-VBL	BCSR-DEC(3,2)	BCSR(3,1)	BCSR(3,2)
17.kkt_power	CSR	CSR	CSR	CSR	CSR	CSR
18.largebasis	1D-VBL	BCSR-DEC(2,2)	1D-VBL	1D-VBL	BCSR(2,2)	BCSR(2,2)
19.TSOPF_RS_b2383	1D-VBL	1D-VBL	1D-VBL	BCSD-DEC(8)	BCSR(8,1)	BCSR(8,1)
20.af_shell10	1D-VBL	1D-VBL	1D-VBL	1D-VBL	BCSR(5,1)	BCSR(5,1)
21.audikw_1	1D-VBL	1D-VBL	1D-VBL	BCSR-DEC(2,3)	BCSR(3,1)	BCSR(3,1)
22.F1	1D-VBL	1D-VBL	BCSR-DEC(1,3)	BCSR-DEC(3,1)	BCSR(3,1)	BCSR(3,1)
23.fdiff202x202x102	BCSD(8)	BCSD(7)	BCSD(8)	BCSD(8)	BCSD(6)	BCSD(6)
24.gearbox	BCSR-DEC(3,2)	BCSR-DEC(3,1)	1D-VBL	BCSR(3,1)	BCSR(3,1)	BCSR(3,1)
25.inline_1	1D-VBL	1D-VBL	1D-VBL	BCSR(3,1)	BCSR(3,1)	BCSR(3,1)
26.ldoor	1D-VBL	1D-VBL	1D-VBL	BCSR-DEC(2,2)	BCSR(7,1)	BCSR(7,1)
27.pwtk	1D-VBL	1D-VBL	1D-VBL	1D-VBL	1D-VBL	BCSR(5,1)
28.thermal2	CSR	BCSD-DEC(8)	CSR	BCSD-DEC(7)	CSR	CSR
29.nd24k	1D-VBL	1D-VBL	1D-VBL	BCSR-DEC(3,1)	BCSR(3,2)	BCSR(3,2)
30.stomach	BCSD(7)	BCSD(7)	BCSD-DEC(4)	BCSD(7)	BCSD(7)	BCSD(7)

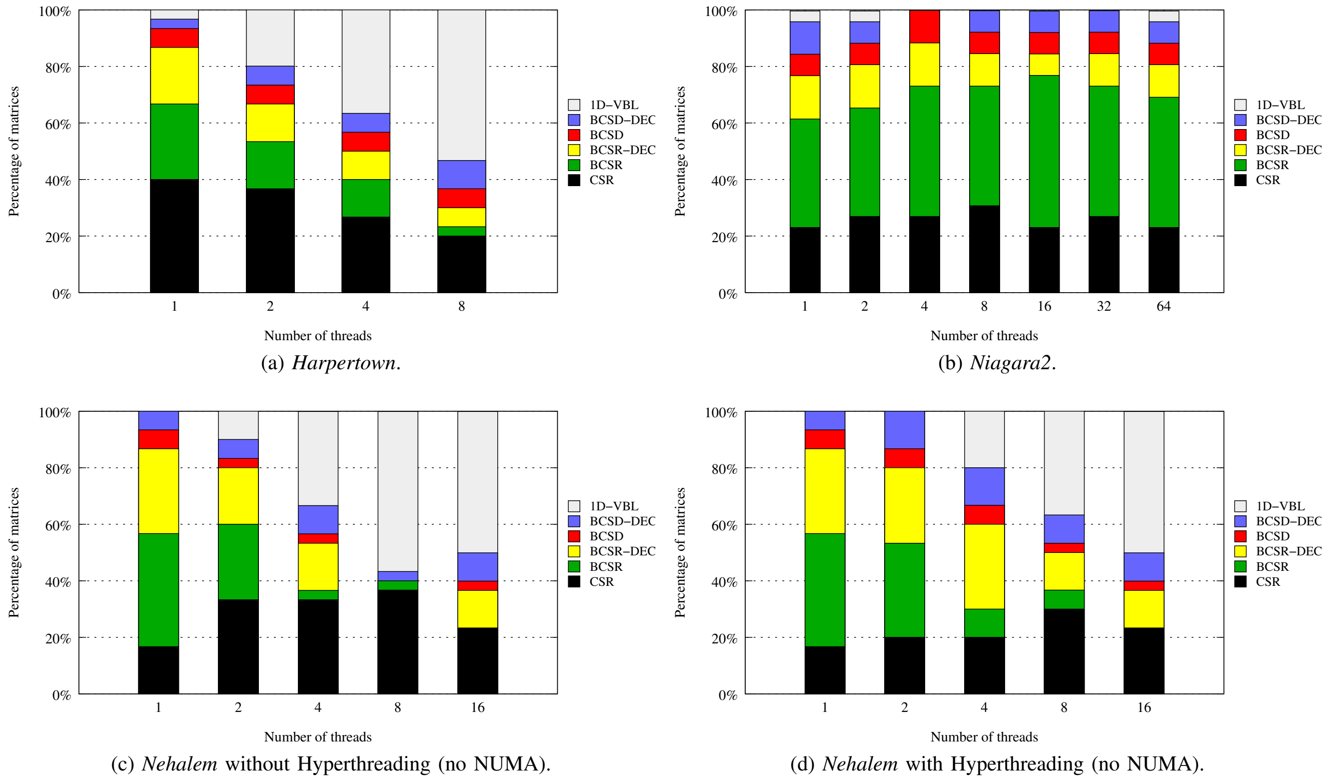
Table IV: Best storage format for each microarchitecture (all threads). The numbers inside parentheses for fixed size blocking methods denote the block that achieved the best performance. Results are depicted for both single (sp) and double (dp) precision.

for *Harpertown*, 16 for *Nehalem*, and 64 for *Niagara2*. The most important observation from this table is that there is not a single method that could yield the best performance for a specific matrix across the different microarchitectures. Additionally, even if the method remains the same across the different architectures or precision modes, the specific block that achieves the highest performance can still vary significantly. It is obvious, therefore, that predicting the correct blocking method and block for a specific combination of sparse matrix, microarchitecture, and thread configuration is rather hard, and the problem becomes even harder when considering how the best performing blocking storage formats vary as the number of threads increases. This aspect of blocking performance is discussed in the following.

Figure 4 presents the percentage of matrices that each method ‘wins’, i.e., achieved the highest performance, as the number of threads increases. Threads were assigned to hardware contexts as follows:

- On *Harpertown* up to the four-thread configuration, we assigned threads to cores, such that none of them shared the unified L2 cache.
- On *Nehalem* we tried two thread-assignment policies in order to examine the behavior of blocking methods when using simultaneous multithreading. The first policy assigns a single thread per core, while the second one assigns two threads per core. In both cases, however, the 16-thread configuration uses Hyperthreading. We should also note here that we did not implement NUMA-aware versions of any blocking method.
- On *Niagara2*, finally, we assigned threads sequentially to cores, i.e., threads 1–8 were assigned to the first core, 9–16 to the second, and so forth.

Figure 4 reveals a number of important aspects of the different blocking method as the number of cores increases and renders the problem of selecting the correct method for a specific architecture and thread configuration even more complex. First, a difference in behavior of blocking methods on Intel processors and the *Niagara2* is to be observed. On *Niagara2*, BCSR achieves the overall best performance for more than 40% of matrices independent of the number of threads, and in general, the distribution of wins between different formats does not change significantly as the number of threads increases. The dominance of BCSR on *Niagara2* is not a surprise, since its execution core is not as sophisticated as Intel cores (shallower pipeline, no data hardware prefetching, lower clock frequency, etc.), and,



(a) *Harpertown*.

(b) *Niagara2*.

(c) *Nehalem* without Hyperthreading (no NUMA).

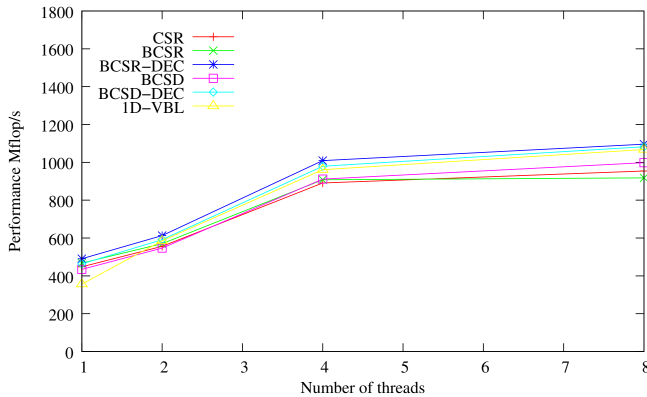
(d) *Nehalem* with Hyperthreading (no NUMA).

Figure 4: Percentage of matrices that each method achieved the highest performance relative to the number of threads (double-precision).

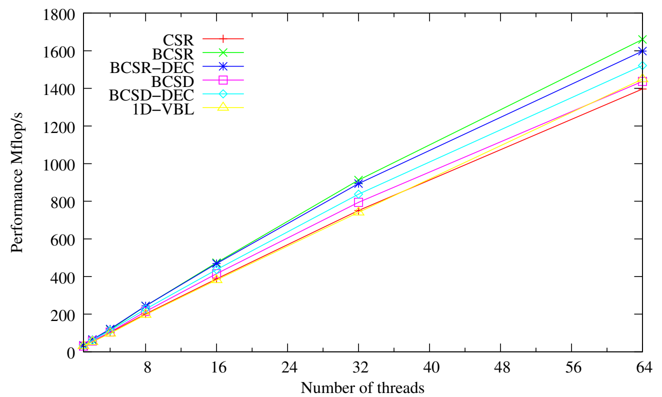
additionally, BCSR is generally a more lightweight kernel than the other blocking kernels (no additional data structures and floating-point operations, better spatial locality, better register reuse, easily optimized code, etc.).

In the case of *Harpertown* and *Nehalem*, however, the distribution of best performing blocking methods varies dramatically as the number of threads increases. The variable size blocking method, 1D-VBL, achieves the best overall performance for more than half the matrices of our matrix suite at eight- and 16-thread configurations, respectively, despite the moderate performance for one- and two-thread configurations. On the other hand, BCSR performs rather poorly as the number of threads increases and the memory subsystem is further stressed. The portion of BCSR-DEC 'wins' decreases also considerably. The main problem of BCSR and methods with aggressive padding is that they sacrifice memory bandwidth, in order to construct full blocks, hoping that the better computational characteristics of the block kernels will overwhelm the difference and lead to higher performance. However, when the pressure to the memory subsystem becomes excessive, as is the case of the dual SMP *Harpertown* configuration and the non-NUMA implementation that we used on *Nehalem*, the effective memory bandwidth decreases considerably and becomes the major performance bottleneck of these methods. On the other hand, 1D-VBL, which for more than half the matrices has the least working set, takes advantage of its lower demands on memory bandwidth and manages to achieve the best overall performance. The distribution of best performing blocking methods is similar when using Hyperthreading on *Nehalem*, but in this case the performance advantage of 1D-VBL is not so dominant. This is expected, since the 1D-VBL kernel need to perform more computations and the threads contend for processor resources.

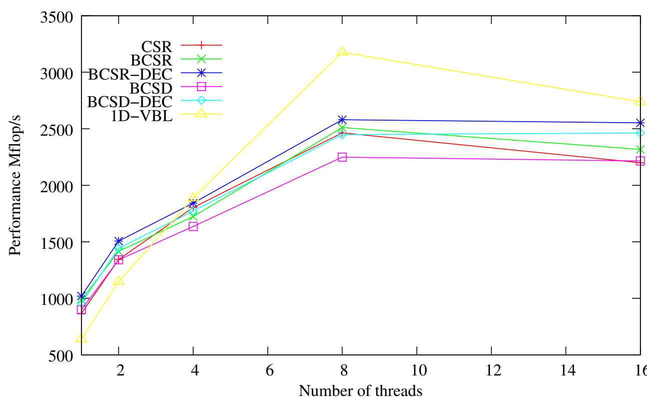
Finally, we present on Fig. 5 how each blocking method scales with the number of threads. First, we should note that the best average performance ($\approx$3.2 Gflop/s) is achieved by *Nehalem* at eight-thread configuration without Hyperthreading using the 1D-VBL format. The performance of *Niagara2* is noteworthy, since due to its almost linear scaling as the number of threads increases, it achieves an average $\approx$1.7 Gflop/s, compared to the rather moderate $\approx$1.1 Gflop/s of *Harpertown*. From this figure it is obvious how better than the other blocking formats 1D-VBL scales for *Harpertown* and *Nehalem*, a fact which is in accordance with the distribution of best performing formats presented in Fig. 4. Similarly, BCSR is the format that scales best on *Niagara2*. The degradation of performance on *Nehalem* as we move



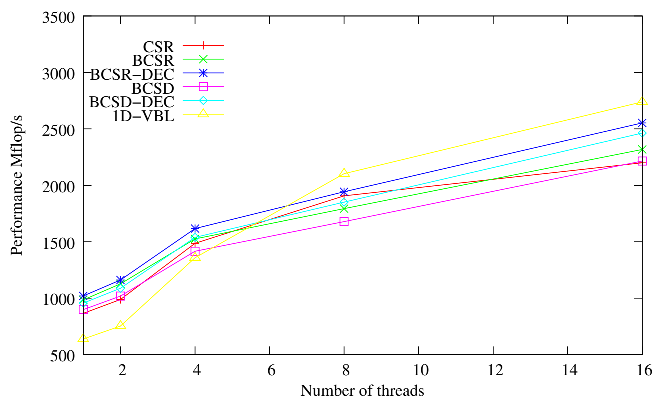
(a) *Harpertown*.



(b) *Niagara2*.



(c) *Nehalem* without Hyperthreading (no NUMA).



(d) *Nehalem* with Hyperthreading (no NUMA).

Figure 5: Performance scaling of each format. Each point represents the average maximum performance achieved from each storage format over all matrices (double-precision).

from eight to 16 threads (Fig. 5c) is due to excessive pressure to the memory subsystem, which is further aggravated by resource sharing inside the processor due to Hyperthreading, which is inevitably used when moving to 16 threads.

V. Conclusions and Future Work

In this paper, we conducted a comparative study and evaluation of blocking storage formats for sparse matrices on modern multicore architectures. Blocking methods for sparse matrices have been introduced in the literature in the past, but it lacked an extensive evaluation on contemporary multicore architectures. In this paper, we provided a consistent overview of the different blocking methods for sparse matrices and evaluated the performance of five representative blocking methods. The conclusions of our evaluation are interesting, since we show that blocking is still a promising sparse matrix optimization for multicores, but the prediction of the correct blocking method to use and the correct tuning of that method is a hard and intricate problem that becomes even more complex as the number of cores and threads increases. Therefore, performance models and advanced heuristics should be sought, which could accurately predict the correct blocking optimization and also adapt to new microarchitectures. Our future research will focus on using machine learning techniques to aid at the optimization of SpMV on modern and future multicore architectures.

Aknowledgements

We would like to thank Intel Hellas S.A., which provided us with the Nehalem execution platform that we used in our experiments. This work was supported by the Greek Secreteriat of Research and Technology (GSRT) and the European Commission under the program 05AKMWN95.

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 *Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, pages 32–41, Minneapolis, MN, United States, 1992. IEEE Computer Society.

[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, 1994.

[3] T. Davis. The University of Florida sparse matrix collection. NA Digest, vol. 97, no. 23, June 1997. http://www.cise.ufl.edu/research/sparse/matrices.

[4] G. Goumas, K. Kourtis, N. Anastopoulos, V. Karakasis, and N. Koziris. Understanding the performance of sparse matrix-vector multiplication. In *Proceedings of the 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008)*, Toulouse, France, 2008. IEEE Computer Society.

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

[6] E.-J. Im and K. A. Yelick. Optimizing sparse matrix computations for register reuse in SPARSITY. In *Proceedings of the International Conference on Computational Sciences – Part I*, pages 127–136. Springer-Verlag, 2001.

[7] V. Karakasis, G. Goumas, and N. Koziris. Exploring the effect of block shapes on the performance of sparse kernels. In *IEEE International Symposium on Parallel and Distributed Processing (Workshop on Parallel and Distributed Scientific and Engineering Computing with Applications)*, Rome, Italy, 2009. IEEE.

[8] K. Kourtis, G. Goumas, and N. Koziris. Improving the performance of multithreaded sparse matrix-vector multiplication using index and value compression. In *Proceedings of the 2008 37th International Conference on Parallel Processing*, Portland, Oregon, United States, 2008. IEEE Computer Society.

[9] J. D. McCalpin. STREAM: Sustainable memory bandwidth in high performance computing, 1995. http://www.cs.virginia.edu/stream/.

[10] A. Pinar and M. T. Heath. Improving performance of sparse matrix-vector multiplication. In *Proceedings of the 1999 ACM/IEEE conference on Supercomputing*, Portland, OR, United States, 1999. ACM.

[11] Y. Saad. SPARSKIT: A basic tool kit for sparse matrix computations, 1994.

[12] O. Temam and W. Jalby. Characterizing the behavior of sparse algorithms on caches. In *Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, pages 578–587, Minneapolis, MN, United States, 1992. IEEE Computer Society.

[13] R. Vuduc. *Automatic Performance Tuning of Sparse Matrix Kernels*. PhD thesis, University of California, Berkeley, 2003.

[14] R. W. Vuduc and H.-J. 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 Berlin/Heidelberg, 2005.

[15] J. Willcock and A. Lumsdaine. Accelerating sparse matrix computations via data compression. In *Proceedings of the 20th annual International conference on Supercomputing*, pages 307–316, Cairns, Queensland, Australia, 2006. ACM.

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