Exploring the Effect of Block Shapes on the Performance of Sparse Kernels

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

Abstract

In this paper we explore the impact of the block shape on blocked and vectorized versions of the Sparse Matrix-Vector Multiplication (SpMV) kernel and build upon previous work by performing an extensive experimental evaluation of the most widespread blocking storage format, namely Block Compressed Sparse Row (BCSR) format, on a set of modern commodity microarchitectures. We evaluate the merit of vectorization on the memory-bound blocked SpMV kernel and report the results for single- and multithreaded (both SMP and NUMA) configurations. The performance of blocked SpMV can significantly vary with the block shape, despite similar memory bandwidth demands for different blocks. This is further accentuated when vectorizing the kernel. When moving to multiple cores, the memory wall problem becomes even more evident and may overwhelm any benefit from optimizations targeting the computational part of the kernel. In this paper we explore and discuss the architectural characteristics of modern commodity architectures that are responsible for these performance variations between block shapes.

1. 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 [6], [7] compression [9], [10], [15], column or row reordering [11], and others.

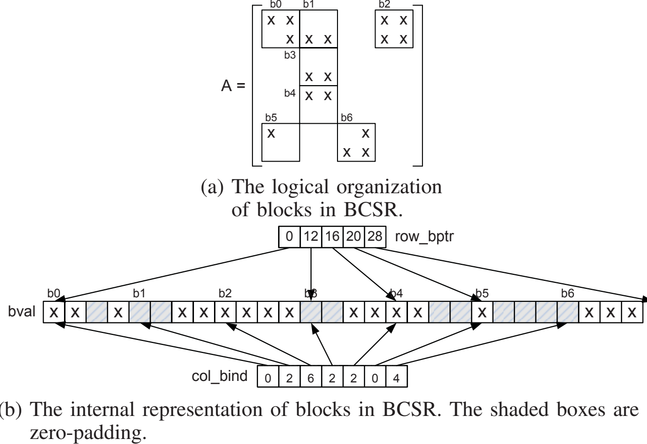
The register blocking technique using the BCSR format [7], though initially not intending to tackle directly the memory wall problem but rather to attack the indirect memory references, seems currently to be one of the most promising optimization techniques for SpMV. BCSR groups neighboring non-zero elements into one block and is able to keep one index per block, instead of one index per element, which is the CSR case. In this way, it can achieve significant reduction of the algorithm's working set, especially for matrices with a large number of dense sub-blocks, greatly alleviating the pressure to the memory subsystem. The BCSR format allows also for other straightforward optimizations targeting the computational part of the kernel, such as loop unrolling and vectorization.

Recent work [4], [5], [16] has identified the memory subsystem, and more specifically the memory bandwidth, as the main performance bottleneck of SpMV. Nevertheless, having attacked the above problem with an appropriate storage format, such as BCSR, focusing on the computational part of the kernel, which now constitutes a larger portion of the total execution time, seems a relevant approach. Vectorization on SpMV has been studied recently [16] along with other optimizations, but produced modest results. Although vectorization is known to be a successful optimization for linear algebra kernels, this does not seem to be the case for SpMV due to its memory-bound nature. Thus, the results discussed in [16] are not surprising. Nevertheless, we move one step ahead with a goal to understand the architectural implications that are responsible for this execution behavior, and explore the possibility and conditions that must hold in order to have performance gain from vectorization. In our study, we conduct a series of experiments for both single- and multithreaded configurations of the BCSR algorithm on a representative set of modern, commodity microprocessors with different memory subsystems and SIMD capabilities and examine the impact of various block shapes. Although the benefit of vectorization is small on average, there exist several cases—especially in a multithreaded configuration—that vectorization can indeed achieve significant speedups.

The rest of the paper is organized as follows: Section 2 provides some background information about the BCSR storage format. Section 3 describes the architectural implications on the use of different block shapes for BCSR, Section 4 presents our experimental results, and Section 5 concludes the paper and describes future work.

2. Storage Formats for Sparse Matrices

The standard storage format for sparse matrices is the Compressed Sparse Row (CSR) format [1]. This format uses three one-dimensional arrays to store a $n \times n$ sparse matrix with nnz non-zero elements: an array `val` of size nnz to store the non-zero elements of the matrix, an array



(a) The logical organization of blocks in BCSR.

(b) The internal representation of blocks in BCSR. The shaded boxes are zero-padding.

Figure 1: The BCSR storage format.

```
for (i=0; i<nr_block_rows; i++) {
  y0 = y1 = 0;
  for (j=row_bptr[i], j0=j/(2*2), k=col_bind[j0];
       j<row_bptr[i+1]; j+=2*2, k=col_ind[++j0]) {
    y0 += bval[j]*x[k] + bval[j+1]*x[k+1];
    y1 += bval[j+2]*x[k] + bval[j+3]*x[k+1];
  }
  y[2*i] = y0; y[2*i+1] = y1;
}
```

Figure 2: The standard BCSR SpMV implementation for a 2×2 block.

`col_ind` of size nnz to store the column indices of every nonzero element, and an array `row_ptr` of size $n + 1$ to store pointers to the first element of each row in the `val` array.

The blocked version of the CSR format is the Blocked Compressed Sparse Row (BCSR) format [7]. BCSR maintains three matrices, `bval`, `col_bind`, and `row_bptr`, which instead of storing and pointing to individual elements of the matrix, they store—in row-wise order—and point to dense $r \times c$ sub-blocks of the matrix. Figure 1 shows how blocking is applied on a sparse matrix and the data structures used by the BCSR format. When an incomplete block is encountered, BCSR pads with explicit zeros. The BCSR format can significantly reduce the working set of the kernel, by dividing the matrix into blocks and storing only the block-column indices. However, increasing the block size beyond a certain limit inserts excessive padding, the cost of which can overwhelm any benefit gained from the reduction of the indexing structures. In general, selecting blocks that minimize the working set is a good strategy to attain high performance [5]. BCSR blocks are row- and column-aligned at r and c elements boundaries, respectively. Although this alignment may seem restrictive and, generally, lead to more padding [14], it can greatly favor vectorization as it will be explained in the following. Figure 2 shows the SpMV kernel for BCSR with 2×2 blocks.

3. Architectural implications on the execution of blocked and vectorized SpMV kernels

The shape of a block selected to group neighboring nonzero elements in BCSR is of vital importance for the performance of SpMV. The primary criterion for the selection of an efficient block shape should be the final working set of the algorithm, since the standard CSR SpMV implementation suffers from a bottleneck in memory bandwidth. Therefore, block shapes that lead to excessive padding may eventually increase the working set of the algorithm, leading to lower performance. In this paper, however, we put aside the impact of the working set and study possible differences in performance using blocks of different shapes that lead to almost the same working set. Moreover, when the nonzero elements are organized in blocks, applying vectorization seems straightforward and promising. In the next paragraphs, we discuss some architectural characteristics that can interact with the block shape and, eventually, influence performance.

Instruction-Level Parallelism. The modern commodity superscalar microarchitectures rely heavily on instruction level parallelism to obtain high performance from a single core. Thus, any instruction sequence containing RAW dependencies that can produce stalls should be avoided. In the case of large "horizontal" blocks, such as the 1×8 block, half of the floating point instructions executed—the additions—depend on each other, since all partial sums should be accumulated to a single register or memory location. This instruction stream cannot be efficiently organized or scheduled, in order to fully utilize the execution core, therefore, such blocks may lead to reduced performance.

Data alignment. Due to hardware limitations, when coding using the SIMD instructions, data fetched from memory should be properly aligned, in order to achieve higher performance. Specifically for the SSE [8] instruction set, vector loads and stores are faster when the requested data are aligned at 16-byte boundaries. The SpMV kernel in BCSR format loads the input vector x and the nonzero elements from the `bval` array and stores the result to the output vector y. In such an access pattern, it is generally quite easy to achieve the desired alignment if the starting addresses of these arrays are properly aligned, since we move sequentially when computing a single block. In order to investigate any differences in performance caused by inappropriate alignment, we implemented the kernel using unaligned load and store instructions for different blocks. The results were rather unsatisfactory, since in every case the vectorization could not offer any speedup, and for some blocks we even encountered a degradation of performance. Consequently, proper alignment of data should be considered as a prerequisite for performance when trying to vectorize SpMV. For this reason, BCSR compared to Unaligned BCSR (UBCSR) [14] is a more appropriate data structure for vectorization, since the logically aligned blocks

of BCSR can be easily aligned in memory without any extra padding. Another not so obvious implication of the alignment requirements is that blocks not having at least one even dimension, such as the 3×1 and 3×3 blocks, cannot be efficiently vectorized, since they cannot be naturally aligned without effectively collapsing to larger blocks.

Dependencies between vector elements. An important issue, when it comes to the SIMD execution model is that both the algorithm and the data structures used must allow for streaming computations. The SpMV kernel has a streaming nature, i.e., fetch data from x and `bval`, multiply, and write back to y without any temporal reuse in any of the arrays. However, not every block shape allows for such a straightforward implementation. For example, when trying to vectorize $1 \times c$ blocks, the elements of the vector containing the products must be added together and then accumulated to y (see Fig. 3).

This horizontal dependency between the vector elements can be resolved either by scattering the elements to different registers and then adding them together or by using specialized instructions, which add the elements of a vector horizontally. Both ways, however, are not so efficient for different reasons: scattering the vector elements to different registers requires more instructions and increases the register pressure, while horizontal-add instructions are quite complex and expensive with high latency and low throughput [8]. Which of the two ways is more effective is a platform-dependent issue. A possibly positive side-effect of the horizontal-add instructions is that they can effectively hide a RAW dependency between two instructions, since this dependency is moved inside the vector. In practice, however, very few were the cases that an horizontal vectorized block outperformed its vertical counterpart.

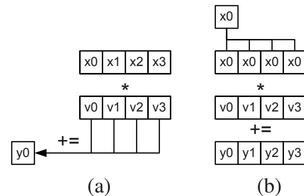


Figure 3: The horizontal dependency between vector elements for $1 \times c$ layouts (a) and the better parallelization of the $r \times 1$ layouts (b).

Half vectors. When using single-precision arithmetic, i.e., each vector can hold up to four elements, some block shapes, e.g., 2×1, 6×1, etc. lead to the use of half vectors, i.e., vectors with only two elements. The problem in this case is not only that the SIMD units of the processor will be under-utilized, but the limitations that may be imposed by the microarchitecture when manipulating half registers. For example, if there are dependencies between the upper and lower halves of two registers, it is architecture-dependent whether their values can be forwarded. Another performance obstacle for such blocks is that full-vector loads from memory cannot be used, since only an 8-byte alignment can be guaranteed. For that reason, partial vector loads should be used and possibly vector shuffle instructions to properly arrange the elements inside the vector, which can further degrade performance.

Duplicate loads. When trying to vectorize $r\times1$ blocks, the load from the input vector x must be a scalar load, and then the loaded value must be propagated to the rest of vector positions in order to perform the multiplication in parallel. This two-step operation may hinder performance, since it requires more instructions, and in current microarchitectures shuffle instructions usually have greater latency. For that reason, it may be preferable to use special instructions, if available, e.g., the SSE3 `movdup` family of instructions, which automatically propagate the loaded data to the different positions of the SIMD register.

Streaming stores. In the SpMV kernel, as it is depicted in Fig. 2, the stores to y are performed only once and their values are never reused. Consequently, any cache misses incurred by these stores could evict useful data and pollute the cache. Thus, a possible optimization would be to use *streaming stores* to the vector y, which bypass the cache and store directly y to the main memory.

Implications on two-dimensional blocks. The above considerations of horizontal and vertical blocks can have an impact on the performance of two-dimensional blocks as well, especially in the vectorized version of the kernel. In effect, the implementation of a two-dimensional block is the unrolled version of a corresponding one-dimensional block. For example, a 2×2 block can be implemented as a two times unrolled version of either the 1×2 or the 2×1 block. However, since differences in performance can be significant between the different one-dimensional blocks, selecting the appropriate one-dimensional block to unroll could determine the performance of two-dimensional blocks. Additionally, unrolling the code of vertical blocks, implies that elements inside the block should be stored column-wise.

4. Experimental evaluation

In order to quantify the effect of the block shape on the performance of the single-threaded SpMV, we used a representative set of commodity microarchitectures, consisting of an Intel Pentium IV Xeon (*Netburst*), an Intel Core 2 Duo Xeon (*Woodcrest*), and an AMD Dual-Core Opteron (*Opteron*). Table 1 summarizes the architectural characteristics of each microarchitecture.

As a base implementation of BCSR, we used the implementation provided by the OSKI [13] sparse kernel optimization library and tweaked the block-specific BCSR multiplication routines by using the vector instructions of the processor. Both the standard and the vectorized versions of OSKI were compiled using `gcc` version 4.2 with the `-O3` and the `-funroll-loops` switches turned on. For the SSE instructions we either explicitly used the compiler intrinsics or implemented our own, whenever an intrinsic

	Netburst	*Woodcrest*	*Opteron*
Clock speed	2.80 GHz	2.66 GHz	1.8 GHz
L1-cache	Trace cache + 16 KB, 4-way, Data, 64 byte cacheline	32 KB, 8-way, Instr. + 32 KB, 8-way, Data, 64 byte cacheline	64 KB, 2-way, Instr. + 64 KB, 2-way, Data, 64 byte cacheline
L2-cache	1 MB, 8-way, Data, 64 byte cacheline	4 MB, 16-way, Data, 64 byte cacheline	1 MB, 16-way, Data, 64 byte cacheline
SIMD capabilities	SSE, SSE2, SSE3	SSE, SSE2, SSE3, SSSE3	SSE, SSE2, SSE3
SIMD execution units	3 (pipeline shared, issued from one port)	3 (issued from three ports)	3 (pipeline shared)

Table 1: Characteristics of the microarchitectures used.

was not available for the desired instruction. All experiments were run on a GNU/Linux platform running the 2.6 kernel. The matrix suite for the experiments comprised of 59 matrices for double-precision and 40 matrices for single-precision arithmetic selected from Tim Davis' collection of sparse matrices [2]. For more information about the matrices used, the reader is referred to [4]. The reason for selecting different sets for double- and single-precision is mainly technical. In order to save padding, the OSKI library breaks the strict alignment of BCSR blocks when the last block-column exceeds the bounds of the original matrix, by truncating the blocks of the before-last block-column. Although this is not a concern for the unvectorized BCSR, it forces at least an unaligned vector load from input vector when using vectorization, which can harm performance (see Section 3). Changing the OSKI internal representation of BCSR would lead to extensive changes to the BCSR module, thus we decided to use only matrices whose number of columns was divisible by two or four for double- and single-precision, respectively. However, the set of matrices is still representative, since it contains matrices from different categories, as it is depicted in Table 2. For the implementation of the vectorized version on *Netburst*, we used duplicate loads from memory and streaming stores, since these optimizations offered an additional 2%–6% speedup. We also avoided the use of horizontal adds for large horizontal blocks, since these instructions have a very large 13-cycle latency and 4- to 6-cycle throughput [8]. On the other hand, the implementation on *Woodcrest* was more straightforward, since neither duplicate loads nor streaming stores offered any speedup, and horizontal adds are much better implemented having a modest 3-cycle (6-cycle for single precision) latency and 2-cycle throughput for both single- and double-precision [8].

For each microarchitecture, we have conducted extensive experiments for the vectorized and unvectorized versions of the BCSR kernels, for single- and double-precision arithmetic, and for all $r \times c$ block shapes, such that $r \cdot c \leq 8$. Larger blocks were not considered, since preliminary experiments showed that BCSR failed to provide any significant speedup over the basic CSR for such blocks, due to excessive padding. We performed 128 iterations of each multiplication and the results presented are the average over all the iterations. We should note here 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 it was recently accessed. We have made two decisions for the illustration of our experimental results, which aid in understanding the performance issues involved. First, in several cases we focus on the results collected for the dense 1000×1000 matrix (#001). The dense matrix demonstrates several performance issues in a more clear way and provides the upper limits for several candidate optimizations. Second, we show our experimental results for single precision, where the vector units of the microprocessors are wider. Though in practice double precision is used for the majority of the cases, single precision exhibits better the vectorization capabilities of current commodity processors and gives a picture of future microarchitectures, where wider vectorization units are expected. However, all observations and conclusions hold for double precision as well, unless differently stated.

Problem category	*Matrix id*
Chemistry	014 (d), 020, 023, 097
Circuit Simulation	018 (d), 051, 057 (d), 063, 080 (d), 086 (d)
Computational Fluid Dynamics	002, 005, 009 (d), 011, 013 (d), 017 (d), 021, 024, 032, 035, 036
Graphs	073 (d)
Financial	025
Linear Programming	040 (d), 042, 044, 069
Materials	007
Semiconductor Device	029, 037
Structural	003 (d), 045, 048, 049, 055, 059 (d), 064, 066, 071 (d), 072, 078 (d), 096 (d)
Miscellaneous 2D/3D	015 (d), 062, 075 (d), 076, 077, 083, 085, 089, 092
Other	028, 067 (d), 068 (d), 087, 099 (d)
Dense	001
Random	046

Table 2: Problem categories of matrices used. Those matrices marked with a 'd' were only used for double-precision experiments. See in text for the reason behind this selection.

Processor	*Single-precision*	*Double-precision*
Woodcrest	19.90%	9.22%
Netburst	11.77%	13.71%
Opteron	−0.37%	−0.07%

Table 3: The effect of vectorization on BCSR. The figures in the table are average values over all the matrices used in each case.

4.1. Preliminary results: the effect of vectorization

Table 3 illustrates the impact of vectorization for single- and double-precision for each of the microarchitectures considered. It is noteworthy that the vectorization can provide non-negligible speedups even for the memory-bound SpMV kernel, especially in the case of *Woodcrest*, whose SIMD engine is more sophisticated and better integrated to the microarchitecture. On the other hand, *Opteron* failed to gain any performance benefit from vectorization. This is due to the fact that the AMD microarchitecture implements the SSE instruction set by splitting each 128-bit instruction into two 64-bit macro-ops, which is effectively equivalent to executing two standard 64-bit instructions [3]. For that reason, we will present results only from *Woodcrest* and *Netburst* for the single-threaded configuration. Figure 4 presents the effect of vectorization using single-precision for every matrix considered. It should be noted here that vectorization can provide speedup in cases where simple BCSR may fail to do so.

4.2. The effect of the block shape

Figure 5 shows the vectorized and unvectorized BCSR performance achieved on the dense matrix using various blocks and single-precision arithmetic. In this figure, the blocks are sorted on the x-axis with increasing size, i.e., decreasing working in our case. We have included two versions of the two-dimensional blocks, ‘a’ and ‘b’, which denote whether we unrolled an horizontal block (version ‘a’) or a vertical block (version ‘b’) to implement them. This distinction is only relevant for the vectorized version of the kernel. A number of observations can be made from these figures that confirm to a large extent the implications discussed in Section 3.

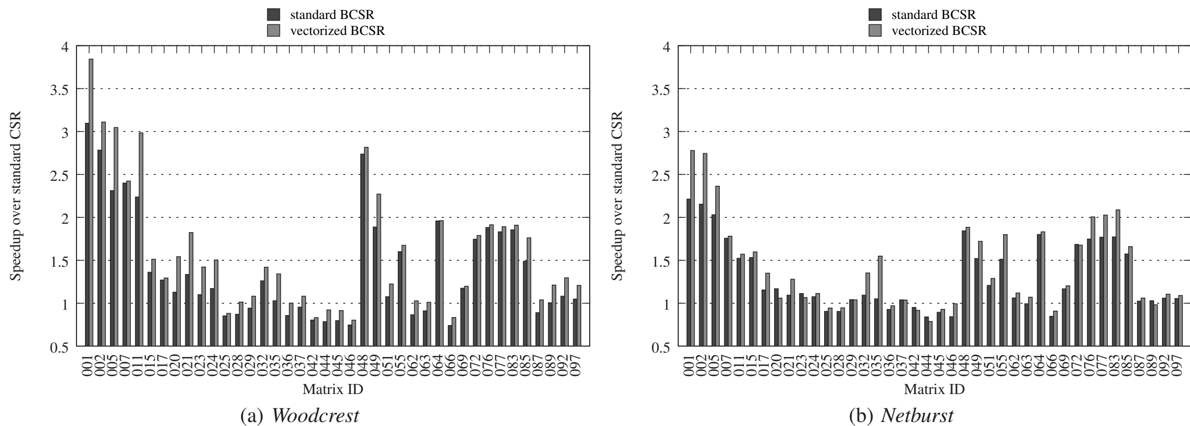
- Larger blocks, i.e., smaller working sets in this case, do not always lead to higher performance. For example, 4×1 blocks exhibit more than 15% higher performance than 1×6 and 1×7 blocks on *Woodcrest*, despite having 12% larger working set. The differences are even greater in the vectorized case.
- Horizontal blocks achieve significantly lower performance than vertical blocks, especially for the vectorized version of the kernel.
- Significant performance differences exist for two-dimensional blocks depending on their implementation.
- The use of full vectors, such as in 4×1 and 8×1 blocks, offer a significant performance boost and can outperform larger blocks (see the *Netburst* case).
- Vectorizing blocks, especially horizontal ones, that cannot be naturally aligned in memory can severely degrade performance. The great differences in performance between such horizontal and vertical blocks is that the implementation of horizontal blocks involves a large number of register shuffle operations, in order to properly align the vector elements before doing the actual computation.

To further support the results of the dense matrix, Fig. 6 presents the performance of BCSR against the working set of two representative matrices (kim2, #085, and msc23052, #049), where BCSR can achieve significant speedup. The points in the graphs represent the performance of the 22 block shapes used, including the different implementations for two-dimensional blocks, for both the vectorized and unvectorized versions of the kernel. Although there is a clear tendency to higher performance when the working set of the algorithm decreases, a closer look at the graphs reveals that there exist significant variations in performance among blocks that lead to similar working sets. Furthermore, the block shape that leads to the absolutely smaller working set does not always achieve the best performance. Similar behavior was observed for the rest of the matrices that BCSR achieved speedup.

4.3. Multithreaded implementation and results

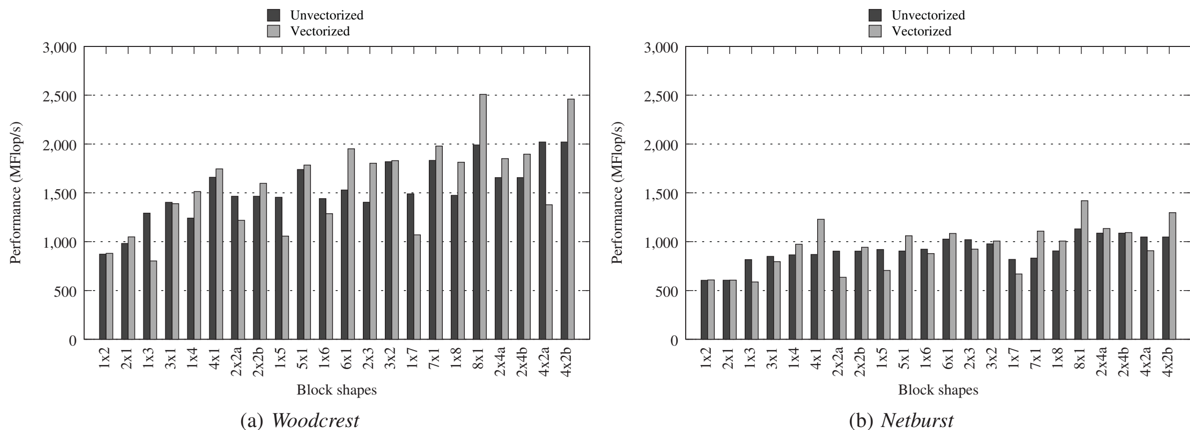
We have also implemented a multithreaded version of SpMV using the BCSR storage format, in order to investigate the effect of block shape on the current multicore microarchitectures. We have chosen not to implement a multithreaded version of the actual multiplication routines of OSKI, since this would require extensive changes and a possible redesign of some modules. Instead, we have split the matrix row-wise and assigned each thread a chunk of the initial matrix converted to the BCSR format. The actual splitting of the initial matrix is based on the non-zero elements, such that each chunk is assigned almost the same number of non-zero elements. We impose two restrictions to the splitting procedure: each chunk shall contain whole rows, and the number of rows shall be divisible by the vector size, so that vectorization alignment requirements are always met when storing to y. For the multithreaded implementation, we have used the NPTL 2.7 library.

The platforms used for our multithreaded experiments were (a) a two-way SMP Intel Core 2 Quad Core Xeon (*Clovertown*) and (b) a two-way ccNUMA AMD Dual-Core Opteron (*Opteron*). The architectural characteristics of the *Opteron* have already been described in Section 4. *Clovertown* is clocked at 2 GHz, has separate 8-way set



(a) *Woodcrest* (b) *Netburst*

Figure 4: Effect of vectorization on the performance of BCSR for individual matrices. For each matrix, the block with the best performance is selected.



(a) *Woodcrest* (b) *Netburst*

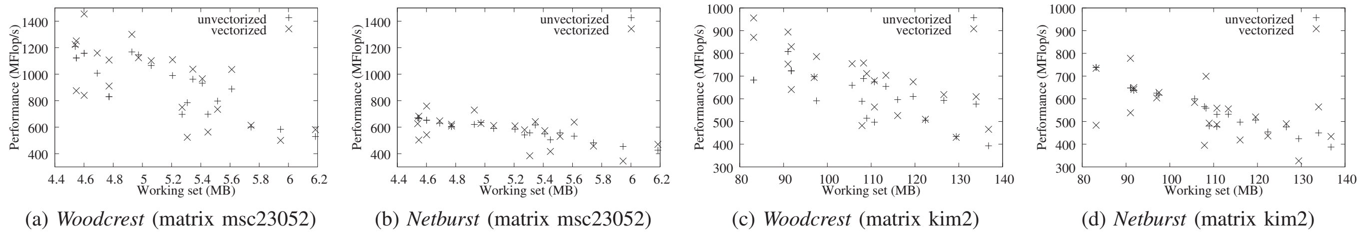
Figure 5: Vectorized and unvectorized BCSR performance for different block shapes on the dense matrix (single-precision).

associative, 32 KB instruction and data L1-caches, and a 16-way set associative, 4 MB L2-cache.

Figures 7a and 7b show the BCSR performance on the 1000×1000 dense matrix as the number of cores increases. We only depict results of blocks with a total size of eight elements, since these lead to the smallest working set. It should be noted here that in the two- and four-thread configurations we have assigned threads to cores, such as none of them shares the cache. When the number of cores increases, the differences in performance among different block shapes can be dramatic reaching up to 6 GFlop/s for both versions of the kernel in favor of vertical blocks. Similarly, vectorization can offer a significant gain in performance, 14 GFlop/s vs. 11 GFlop/s.

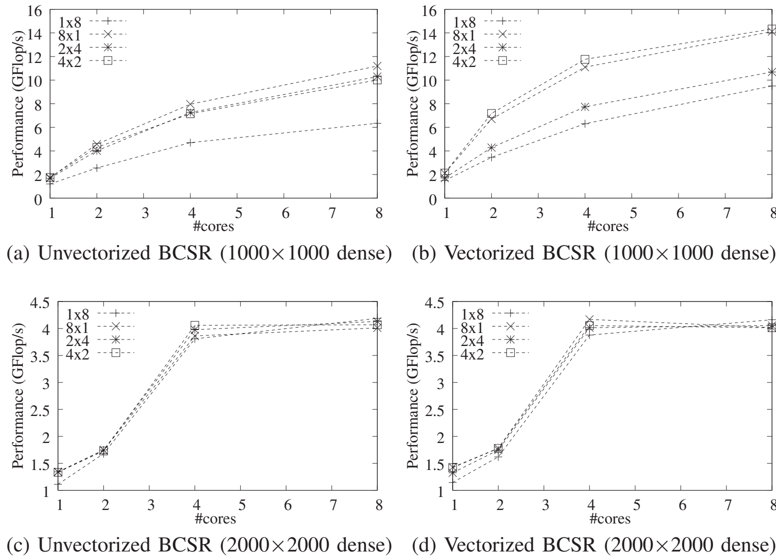
However, these differences may be deceptive, since the problem is no longer memory-bound even from the two-core configuration; the working set of the specific matrix with 8-element blocks is about 8 MB, which when split in two, fits in each thread's cache. In order to expose the memory wall problem, especially in this SMP architecture where 8 cores should access the common bus, we applied the BCSR on a 2000×2000 dense matrix, whose working set using blocks with 8-element blocks is approximately 32 MB. The results are depicted in Figs. 7c and 7d. The memory wall problem is now obvious, since no speedup is achieved for more than four threads. Moreover, the significant speedup encountered for two and four threads can be attributed to the fact that each thread has been assigned a smaller portion of the input matrix, which should fit in the same cache. The most important observation though is that any benefit that a "good" block, or the vectorization itself, would offer is disappeared, when the problem becomes very memory-bound, as is the case of the 2-way SMP quad core configuration. Table 4 also presents the average benefit of vectorization for all matrices as the number of cores increases. It is obvious that as the problem becomes more memory bound (8 cores), the vectorization cannot offer significant speedups.

To further investigate the effect of the memory bottleneck to the behavior of BCSR with different block shapes, we have implemented a NUMA-aware version of the SpMV



(a) *Woodcrest* (matrix msc23052) (b) *Netburst* (matrix msc23052) (c) *Woodcrest* (matrix kim2) (d) *Netburst* (matrix kim2)

Figure 6: The effect of block shape for two representative sparse matrices. Each point represents a distinct block shape for vectorized and unvectorized implementation.



(a) Unvectorized BCSR (1000×1000 dense) (b) Vectorized BCSR (1000×1000 dense)

(c) Unvectorized BCSR (2000×2000 dense) (d) Vectorized BCSR (2000×2000 dense)

Figure 7: Effect of block shape on the multithreaded BCSR for dense matrices (single-precision, *Clovertown*). The memory-wall problem becomes evident for the 2000×2000 matrix.

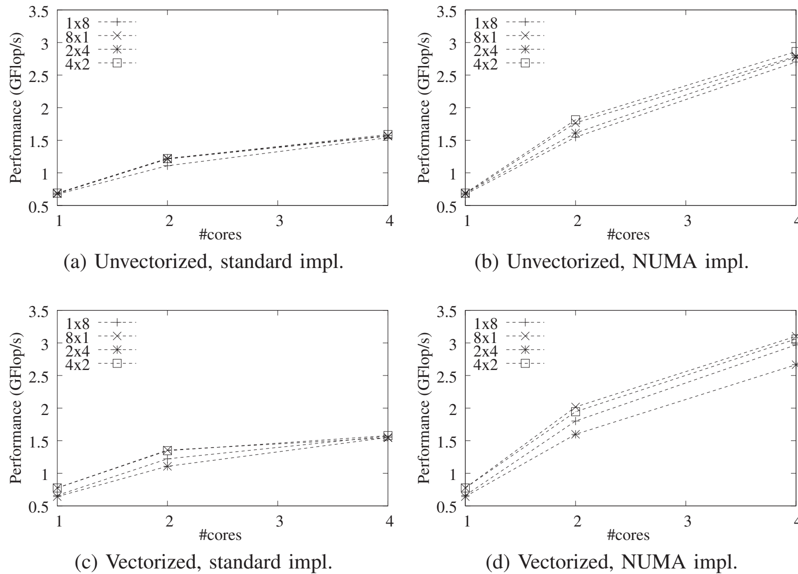
# Cores	*Single-precision*	*Double-precision*
2	9.4%	3.5%
4	9.9%	8.2%
8	3.8%	3.8%

Table 4: The average effect of vectorization as the number of cores increases (*Clovertown*).

algorithm and tested it on *Opteron*. For the NUMA implementation, we used the `libnuma` 1.0.2 library, and explicitly allocated memory on the memory node, where each thread belongs, according to its processor affinity. The results for the NUMA configuration against the standard algorithm for the large dense matrix are depicted in Fig. 8. Apart from the much higher performance that the NUMA-aware algorithm achieved in any case, it is obvious, especially in the vectorized case, that the NUMA implementation can further expose the differences in performance for block shapes (again here the vertical blocks are favored), since the NUMA implementation mitigates the memory bottleneck. Specifically for the four-core configuration, the performance between blocks varies between 2% and 3% for the non-NUMA configuration, while a 6%–16% variation in performance is observed for the NUMA configuration.

5. Conclusions and future work

In this paper, we have explored the effect of the block shape in the BCSR storage format on the performance of the SpMV kernel on modern commodity multicore microarchitectures, especially when using vectorization. We showed that a block shape can significantly affect the SpMV performance, especially when using vectorization, since subtle microarchitectural implications become apparrent. These differences can be further accentuated when the memory intensiveness of the algorithm has already been tackled, either by splitting the workload to a large number of cores or by exploiting the NUMA capabilities of a machine. Furthermore, we showed that vertical block shapes and columnwise storage inside two-dimensional blocks can yield better performance results. As a future work, we opt for implementing and evaluating a performance model for blocking methods for sparse matrices, that apart from working set, it



(a) Unvectorized, standard impl. (b) Unvectorized, NUMA impl.

(c) Vectorized, standard impl. (d) Vectorized, NUMA impl.

Figure 8: Effect of block shape on BCSR using NUMA-aware memory allocation (2000×2000 dense matrix, single-precision).

will also account for the performance differences between block shapes. Such a model could be used to select the most appropriate pair of blocking method and block shape for a specific matrix and architecture among all available blocking methods and their possible block shapes.

References

[1] 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.

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

[3] A. Fog. *The microarchitecture of Intel and AMD CPU's:An optimization guide for assembly programmers and compiler makers*. Copenhagen University College of Engineering, July 2007.

[4] G. Goumas, K. Kourtis, N. Anastopoulos, V. Karakasis, and N. Koziris. Understanding the Performance of Sparse Matrix-Vector Multiplication. In *Euromicro PDP 2008*, Toulouse, France, February 2008.

[5] W. Gropp, D. Kaushik, D. Keyes, and B. Smith. Toward realistic performance bounds for implicit CFD codes. In *Proceedings of Parallel CFD'99*. Elsevier, 1999.

[6] 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.

[7] E. Im and K. Yelick. Optimizing Sparse Matrix Computations for Register Reuse in SPARSITY. *LNCS*, 2073:127–136, 2001.

[8] Intel Corp. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, November 2007.

[9] K. Kourtis, G. Goumas, and N. Koziris. Improving the Performance of Multithreaded Sparse Matrix-Vector Multiplication using Index and Value Compression. In *ICPP*, Portland, Oregon, USA, Sep. 2008.

[10] D. Moloney, D. Geraghty, C. McSweeney, and C. McElroy. Streaming sparse matrix compression/decompression. In *High Performance Embedded Architectures and Compilers, First International Conference, HiPEAC 2005, Barcelona, Spain, November 17-18, 2005, Proceedings*, volume 3793 of *LNCS*, pages 116–129. Springer, 2005.

[11] A. Pinar and M. T. Heath. Improving the Performance of Sparse Matrix-Vector Multiplication. In *Supercomputing'99*, Portland, OR, November 1999. ACM SIGARCH and IEEE.

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

[13] R. Vuduc, J. W. Demmel, and K. A. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. In *Proceedings of SciDAC 2005*, Journal of Physics: Conference Series, San Francisco, CA, USA, June 2005. Institute of Physics Publishing.

[14] 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 *LNCS*, pages 807–816. Springer, 2005.

[15] 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.

[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 *Supercomputing'07*, Reno, NV, November 2007.