AAECC (2007) 18:297–311
DOI 10.1007/s00200-007-0038-9

When cache blocking of sparse matrix vector multiply works and why

**Rajesh Nishtala · Richard W. Vuduc ·
James W. Demmel · Katherine A. Yelick**

Received: 8 November 2005 / Revised: 21 September 2006 /
Published online: 6 March 2007

Abstract We present new performance models and more compact data structures for *cache blocking* when applied to sparse matrix-vector multiply (SpM×V). We extend our prior models by relaxing the assumption that the vectors fit in cache and find that the new models are accurate enough to predict optimum block sizes. In addition, we determine criteria that predict when cache blocking improves performance. We conclude with architectural suggestions that would make memory systems execute SpM×V faster.

Keywords Performance optimization · Sparse matrix multiplication · Memory hierarchies · Performance modeling

1 Introduction and overview

We consider the problem of building high-performance implementations of sparse matrix-vector multiply (SpM×V), or $y \leftarrow y + A \cdot x$. We call x the *source vector* and y the *destination vector*. Making SpM×V fast is complicated both by modern hardware architectures and by the overhead of manipulating sparse

R. Nishtala (✉) · R. W. Vuduc · J. W. Demmel · K. A. Yelick
Computer Science Division, University of California at Berkeley,
575 Soda Hall, Berkeley, CA 94720, USA
e-mail: rajeshn@cs.berkeley.edu

R. W. Vuduc
e-mail: richie@cs.berkeley.edu

J. W. Demmel
e-mail: demmel@cs.berkeley.edu

K. A. Yelick
e-mail: yelick@cs.berkeley.edu

data structures. It is not unusual to see SpM×V run at under 10% of the peak floating point performance of a single processor [15, Fig. 1]. Moreover, in contrast to optimizing dense matrix kernels (dense BLAS) [16,1], performance depends on the nonzero structure of the matrix which may not be known until run-time.

In prior work on the Sparsity system (version 1.0) [7], Im developed an algorithm generator and search strategy for SpM×V that was quite effective in practice. The Sparsity generators employed a variety of performance optimization techniques, including *register blocking*, *cache blocking*, and multiplication by *multiple vectors*. Cache blocking differs from register blocking in that cache blocking reorders memory accesses to increase temporal locality, whereas register blocking compresses the data structure to reduce memory traffic. This paper focuses on performance models for cache blocking (Sect. 2) and asks the fundamental questions of what limits exist on such performance tuning, extending our prior models [15] by accounting for the TLB (translation look aside buffer, i.e., a buffer of most recently used virtual-to-physical address translations), enabling accurate selection of optimal *cache block* sizes. Cache blocking increases the complexity of the data structures used to represent the matrix by adding an extra set of row pointers for each block. The trade-off that needs to made is whether the benefit of the added temporal locality outweighs the added costs of accessing the extra data structures that facilitate the increase in temporal locality. We explore a subset of the matrices used by Vuduc [15], namely, those that are so sparse that register blocking has no benefit. For all the other matrices it was found that cache blocking did not have a significant impact on performance. Thus, to simplify the analysis, we only consider these especially sparse matrices where cache blocking had noticeable advantages or disadvantages. We also assume the matrices are stored in row-oriented format. If the matrices were stored using a column-oriented data structure the roles of x and y would be reversed.

We classify the set of matrices on which we see benefits from cache blocking, concluding that cache blocking is most effective when simultaneously (1) x does not fit in cache (2) y fits in cache, (3) the nonzeros are distributed throughout the matrix (as opposed to a band matrix) and (4) the non-zero density is sufficiently high but not high enough where register blocking makes an appreciable difference. If a matrix does not exhibit one of these properties then cache blocking has no significant impact on performance and thus we do not apply cache blocking to that matrix. However, if these properties exist then the choice of block size is important since blocking can either greatly increase (as high as 2.93×) or greatly decrease (as low as 0.33×) the performance compared to the unblocked versions. One of our aims in this work is to construct models that will allow us to predict the best block-size given static matrix and machine properties.

Traditional static models of cache behavior used to select tile sizes for dense kernels cannot be applied to sparse kernels due to the presence of indirect and irregular memory accesses known only at run-time. Nevertheless, there have been a number of notable attempts to model performance. Temam and Jalby [12], Heras et al. [6], and Fraguela et al. [3] have developed sophisticated probabilistic cache miss models, but assume uniform distribution of non-zero entries. These models differ from one another in their ability to account for self- and cross-interference misses. Our model in Sect. 3 differs from the prior work in that (1) we consider multi-level memory hierarchies including the TLB, and (2) we explicitly model the execution time in addition to cache misses.

Gropp et al. use bounds similar to the ones we develop to analyze and tune a computational fluid dynamics code [4]; Heber et al. present a detailed performance study of a fracture mechanics code on Itanium [5]. This paper considers tuning for matrices that come from a variety of other domains, and explores performance modeling for cache block size selection.

Due to space limitations we only present the high level intuition and summary data. We refer the reader to the full report [8] for details. The software and algorithms described in this paper are available in OSKI (the Optimized Sparse Kernel Interface) [13]. OSKI is a collection of low-level C primitives that provide automatically tuned computational kernels on sparse matrices, for use in solver libraries and applications.

2 Summary of the cache blocking optimization

Matrix data structures

The original matrix is represented in compressed sparse row format (CSR) [9]. In CSR, the nonzero elements of each row and their column indices are stored sequentially in memory (in arrays called value and col_idx in Fig. 1) with another indexing array (called block_ptr) to identify where the new rows start. In essence, cache blocking breaks the CSR matrix into multiple smaller $r_{\text{cache}} \times c_{\text{cache}}$ CSR matrices and then stores these sequentially in memory. In addition a new array (called row_start) is created which points to the start of each row of cache blocks. A diagram of this is shown in Fig. 1. One of the main goals of this work is to choose the size of these $r_{\text{cache}} \times c_{\text{cache}}$ blocks so that we get the most reuse out of caching of the source and destination vectors while keeping the cost associated with the extra overhead low. Henceforth we will refer to r_{cache} as r and c_{cache} as c and assume that there is no register blocking. Below, we discuss how (1) we compress the size of each block using the *row start/end* (RSE) optimization, and (2) further exploit the fact that each cache block is just another sparse matrix. The latter technique also allows easy recursion with multiple levels of cache blocking.

Row start / end

When matrices (especially band matrices) are blocked it is possible that within a cache block non-zeros do not exist on all the rows. The first cache block, for example, might have only nonzero elements in the first tenth of the rows and have the rest of the cache block be empty. However, the basic cache blocked data structure would loop over all zero rows without doing any useful work. In

4 × 4 cache blocking of the following matrix:

$$\begin{pmatrix} A_{00} & 0 & 0 & A_{03} & 0 & 0 & A_{06} & 0 \\ 0 & 0 & 0 & 0 & A_{14} & 0 & 0 & A_{17} \\ 0 & A_{21} & A_{22} & 0 & 0 & A_{25} & A_{26} & 0 \\ A_{30} & 0 & 0 & 0 & A_{34} & 0 & 0 & 0 \\ 0 & 0 & A_{42} & 0 & 0 & 0 & A_{46} & A_{47} \\ 0 & A_{51} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & A_{61} & A_{62} & 0 & 0 & A_{65} & 0 & A_{67} \\ 0 & 0 & A_{72} & A_{73} & A_{74} & A_{75} & 0 & 0 \end{pmatrix}$$

results in the following data structure:

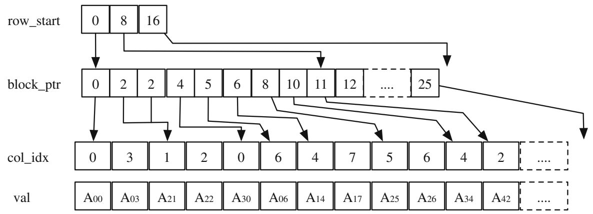

Fig. 1 Cache blocking data structures [7]

order to avoid the unnecessary accesses, a new vector that contains row start (RS) and row end (RE) information for each cache block is also created to point to the first and last nonzero rows in the cache block. This new indexing information makes the performance less sensitive to the size of the cache block. Performance results have shown that this optimization can only improve performance [8]. In our matrix suite, we found that the use of pointers to the first and last nonzero row were sufficient since the nonzero rows were clustered together and thus a list of nonzero rows would have added unnecessary overhead.

Exploiting cache block structure

As described above, the cache blocked matrix can be thought of as many smaller sparse matrices stored sequentially in memory. We exploit this fact by calling our prior sparse matrix vector multiplication routines on each smaller matrix, passing the appropriate part of the source and destination vectors as arguments. The advantage of handling the multiplication in this fashion is that the inner loops can be generated independently of the code for cache blocking and code previously written for non-cache blocked implementations can be reused. This optimization also allows easy recursion with multiple levels of cache blocking. Tests indicate that the function call overhead is negligible since the number of cache blocks for a matrix is usually small compared to the total memory operations.

3 Analytic models

In this section, we create analytic upper and lower bounds on performance by modeling various levels of the memory hierarchy. We first describe the overall performance model and then model the different parts of the memory system that contribute to this overall model. First, we create a load model; we then discuss analytic upper and lower bounds for the number of cache misses at every level. We next examine the upper and lower bounds for TLB misses, and develop a more complex relation between TLB upper and lower bounds to yield a more accurate estimate of the actual number of TLB misses. Our techniques can be extended to construct more accurate cache models, though we found in practice that this initial estimate of TLB misses is sufficiently accurate to predict good block sizes.

3.1 Overall performance model

The overall performance model is similar to the one in [14] except that we have added one more latency term to account for the TLB misses.

We model execution time as follows. First, since we want an upper bound on performance (lower bound on time), we assume we can overlap the latencies of computation and memory accesses. Let h_i be the number of hits at cache level i, and m_i be the number of misses. Then the execution time T is

$$T = \sum_{i=1}^{\kappa-1} h_i \alpha_i + m_\kappa \alpha_{\text{mem}} + m_{TLB} \alpha_{\text{TLB}}, \tag{1}$$

where α_i is the access time (in cycles or seconds) at cache level i, κ is the lowest level of cache, and α_{mem} is the memory access time. The L1 hits h_1 are given by $h_1 = \text{Loads}(r,c) - m_1$ where $\text{Loads}(r,c)$ is the number of loads with an $r_{\text{cache}} \times c_{\text{cache}}$ cache block size (see Sect. 3.2 below). Assuming a perfect nesting of the caches, so that a miss at level i is an access at level $i+1$, then $h_{i+1} = m_i - m_{i+1}$ for $i \geq 1$. The TLB and the L3 might not be nested, so we account for this by assuming that the TLB misses are not overlapped with the misses at the other levels and that they must be serviced before the cache misses can be serviced. The performance, which is expressed as Mflop/s, is $\frac{2k}{T} \times 10^{-6}$ because each of the k nonzero matrix entries leads to one floating point multiply and one floating point add.

To get an estimate of the *upper bound on performance*, let $m_i = M_{\text{lower}}^{(i)}$ in Eq. (1) (where $M_{\text{lower}}^{(i)}$ is a lower bound on misses at the ith cache level as discussed below), and convert to Mflop/s. Similarly, we can get a lower bound on performance by letting $m_i = M_{\text{upper}}^{(i)}$(where $M_{\text{upper}}^{(i)}$ is a upper bound on misses at the ith cache level as discussed below).

In order to take the TLB effects into account we estimate the number of cycles that are needed to process a TLB miss in order to make Eq. (1) match the measured performance. We incorporate it into the upper bound model by setting m_{TLB} equal to $M_{\text{model}}^{(TLB)}$, as described in Sect. 3.4. We use our previously estimated values for the latencies [8].

3.2 Load model

We assume the cache block data structure as described in Sect. 2. We can count the number of loads required for SpM×V as follows. Let A be an $m \times n$ matrix with k non-zeros. Henceforth we assume no register blocking is done which is optimal for all our sparse test matrices. We define a new variable, K_{rc}, to equal the number of cache blocks that a given cache block size $(r \times c)$ produced. In the case that the nonzeros are uniformly randomly distributed throughout the matrix, then $K_{rc} = \lceil \frac{m}{r} \rceil \lceil \frac{n}{c} \rceil$. However, this is not always true and depends on the nonzero structure of the matrix. The loads can be counted in the following manner:

$$\text{Loads}(r,c) = \underbrace{2k + 2(K_{rc}) + 2\sum_{j=1}^{K_{rc}} \delta_j + 2\left\lceil \frac{m}{r} \right\rceil}_{\text{matrix}} + \underbrace{k}_{\text{src vector}} + \underbrace{\sum_{j=1}^{K_{rc}} \delta_j}_{\text{dest vector}} \qquad (2)$$

In Eq. (2), the matrix terms can be counted as follows. We first count two loads for every element of the matrix, one for the column index and one for the actual value. For each cache block j we load the corresponding RS_j and RE_j. Then for each cache block j we load δ_j elements of the block_ptr array twice, once for when the index is used as an ending bound for a row and once again as a starting bound for the next row. Finally we load each element of the row_start twice, once when it is used as the pointer for the end of a row of cache blocks and once when it is used for the start of the next row of cache blocks. For each cache block j we calculate the result of the corresponding destination vector and make δ_j updates to the corresponding destination vector. Since we assume the matrix multiplication is done through compressed sparse row, the total number of reads to the source vector is k.

The fewest number of loads would occur if the matrix were not cache blocked; in this case $\sum_{j=1}^{K_{rc}} \delta_j$ equals m and K_{rc} equals 1. Therefore cache blocking does not decrease the number of loads. If anything, too many cache blocks would greatly increase the overhead.

3.3 Cache miss model

Here we develop upper and lower bounds on the number of cache misses, which lead to lower and upper bounds on performance in MFlops, respectively.

We start with the L1 cache. Let l_1 be the L1-cache line size, in integers. We also assume that a double precision number is represented with twice the number of bytes of an integer. In order to estimate the minimum number of cache misses that can occur we take the total amount of data that we access and divide by the line size. This will give us the total number of lines the matrix, source, and destination vectors would take assuming all the data was perfectly aligned.

Thus, a lower bound $M^{(1)}_{\text{lower}}$ on L1 misses is

$$M^{(1)}_{\text{lower}}(r,c) = \frac{1}{l_1}\left[\underbrace{2k + k + \sum_{j=1}^{K_{rc}} \delta_j + \left\lceil \frac{m}{r} \right\rceil + 2(K_{rc}) +}_{\text{matrix}} \underbrace{2n}_{\text{src vector}} + \underbrace{2m}_{\text{dest vector}}\right] \quad (3)$$

To find the lower bounds for another level of the cache simply replace l_1 with the appropriate line size. To find the upper bound we still assume that every entry in the matrix is loaded once as in the lower bound, but we assume that every access to the source and every access to the destination vectors miss because of conflict and capacity misses.

Thus, an upper bound $M^{(1)}_{\text{upper}}$ on L1 misses is

$$M^{(1)}_{\text{upper}}(r,c) = \underbrace{\frac{1}{l_1}\left[2k + k + \sum_{j=1}^{K_{rc}} \delta_j + \left\lceil \frac{m}{r} \right\rceil + 2(K_{rc})\right]}_{\text{matrix}} + \underbrace{k}_{\text{src vector}} + \underbrace{\sum_{j=1}^{K_{rc}} \delta_j}_{\text{dest vector}} \quad (4)$$

The first k indicates that we miss for every access to the source vector. The second term $\sum_{j=1}^{K_{rc}} \delta_j$ is the number of times that we access the destination vector. Since we stream through the matrix entries and access each element once the number of misses does not depend on conflict or capacity. Notice that neither the load model of Sect. 3.2 nor the cache miss model of this section predict the advantages of cache blocking since they only show an increase in data structure overhead.

3.4 TLB miss model

According to our simple load and cache miss models, cache blocking has no benefit. It turns out that the main benefit of cache blocking is increased temporal locality in the source vector which can be seen in the number of TLB misses, which we model here. Experimental data in Sect. 4 do in fact show improvements in cache misses too, though this is not captured by our model. Still, the model will turn out to be adequate for predicting good cache block sizes. To estimate the lower bounds on the TLB misses, we simply take the total size of the data that we access and divide that by the page size. This quantity is the minimum number of pages that the data resides in and the minimum number of compulsory misses for the TLB. For an estimate of the upper bound we assume that we load every matrix page once. We then assume that we take a TLB miss on every access to the source vector and destination vector. The equations are identical to the cache miss models in Eq. (3) and Eq. (4) except we replace the line size with the page size. All our test matrices incurred at least 1000 TLB misses on the Itanium 2, the only platform on which we have hardware TLB miss counters.

Modeling performance based merely on the lower and upper bound models does not take the increased locality of cache blocking into account because the lower bound on cache misses (which is used to calculate the upper bound on performance) only counts the compulsory misses. Since blocking adds overhead to the data storage, the least amount of overhead occurs when there is no blocking. To factor this in, we need a more accurate model. We had previously observed that many of the matrices have a noticeable increase in the number of TLB misses when the source vector occupies a large fraction of the TLB [8]. Because the number of TLB misses is orders of magnitude higher when the incorrect block size is chosen,[1] we chose to try to more accurately estimate the number of TLB misses through a combination of the lower and upper bound models.

From Fig. 2 we see that there are two distinct categories of block sizes that yielded good performance on our matrix suite (Table 1 in the next section) for the Itanium 2. The first category of matrices (Matrices 2–11) showed the best performance when the column block size equaled $\frac{1}{4}$th of the TLB. In the second category of matrices (Matrices 12–14) the added overhead of blocking hurt performance so the performance was best when the column block size

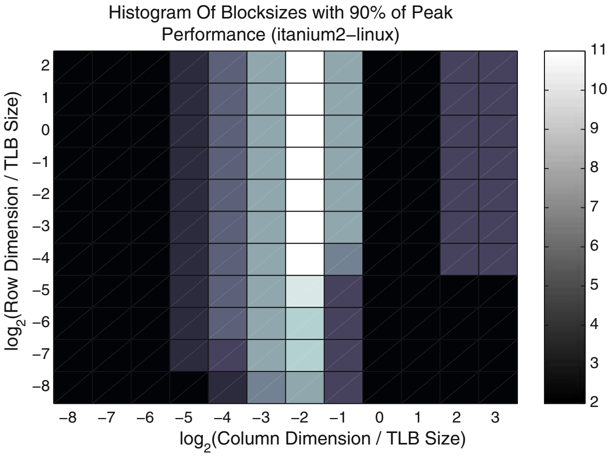

Fig. 2 Histogram of Block sizes for Itanium 2. For each row and column block size shown above, the shade of each cell represents the number of matrices whose performance was within 90% of peak if that block size was chosen. This is a graphical way of representing the optimum cache block sizes for a variety of matrices and their impact on performance. We define *TLB Size* to be the number of entries in the TLB multiplied by the page size. On the Itanium 2 this was 2MB or 256K doubles [15]

[1] This is probably due to early eviction of the source vector with the LRU page replacement policies.

exceeded the number of columns in the matrix (i.e., there was no blocking in the column direction). We also notice that the performance does not depend heavily on the row block size once it is large enough and thus we conclude that no blocking should be done in the row dimension.

To capture this behavior in our performance model we present a modified version of the TLB miss model that combines both the upper bound and lower bound to create a reasonable estimate of the number of misses. One of the main aims is to expose the penalty when there is not enough temporal locality in accessing the source vector. To account for this penalty, our TLB miss model switches to using the upper bound model as an estimate for the number of misses when the column block size is too large.[2] Since the optimal block size as a percentage of the TLB size changes from machine to machine, the optimal crossover point for switching between the lower and upper bound models will vary from platform to platform. TLB counter data was only available for the Itanium 2, thus we present the model for that platform only. The models for the other platforms will be similar.

$$M_{\text{model}}^{(TLB)}(r,c) = \begin{cases} M_{\text{upper}}^{(TLB)}(r,c) \cdot min(\frac{\frac{c\times 2}{p}}{\mathrm{E_T}}, 1) & \text{if } (\frac{2c}{p} \geq \eta \mathrm{E_T})\&(\frac{k}{nzcols} > \rho_T) \quad \text{(5a)} \\ M_{\text{lower}}^{(TLB)}(r,c) & \text{otherwise} \quad \text{(5b)} \end{cases}$$

Equation (5) shows the model used to calculate the approximate number of TLB misses for the Itanium 2. The variables are as follows: p is the page size in integers, $\mathrm{E_T}$ is the number of TLB entries in the TLB, and $nzcols$ is the number of non-zero columns. We use the variable η to define the fraction of the TLB that is filled with the source vector. According to our empirical data for the Itanium 2, for Matrices 2–11 the optimal column block size is $\frac{1}{4}$th of the TLB, thus when the column block size is $\frac{1}{2}$ of the TLB we switch to using the upper bound model. The upper bound is scaled by the fraction of the source vector that overflows the TLB. This switch is only performed when the matrix is dense enough (the average number of nonzeros per nonzero column is greater than ρ_T) ensuring us that blocking provides enough reuse. If either of these conditions fail we use the lower bound model on TLB misses. For the Itanium 2 we empirically determined the values of η and ρ_T to be $\frac{1}{2}$ and 4 respectivly. Our data in [8] shows that when this model is applied to the Itanium 2, it closely predicts the noticeable jump in the number of TLB misses for Matrices 5–8 and Matrices 10–11, the matrices for which cache blocking has the most significant benefits. Therefore this is at least good enough to predict good cache block sizes. In future work, we will refine this model further and verify it for the other platforms. The peaks of the upper bound performance model correlate better to the peaks of the actual performance in most of the matrices. Without this

[2] The actual definition of too large varies across different platforms, for the Itanium 2 we set it at $\frac{1}{2}$ of the TLB.

model the peaks of the upper bound model would show that blocking is not a good idea, which is not the case. We will evaluate the models further in Sect. 4.

4 Verification of the analytic model

We evaluate SpM×V on a set of matrices that are, in principle, large enough and sparse enough for cache blocking to have a significant effect. The properties of the 14 matrices that were chosen are referenced in Table 1. We evaluate the performance model in which we use true hardware counters through PAPI [2] to predict the performance (henceforth called the PAPI model) and compare it to the model in which we use estimates of lower and upper bound of cache and TLB misses (henceforth termed the analytic lower and upper bound models). The cache and memory latencies were derived [15] from published processor manuals, curve fitting, and experimental work using the Saavedra-Barrera memory system microbenchmark [10] and MAPS benchmarks [11]. Due to space limitations we present a summary of the full data [8].

Figure 3 shows an evaluation of the models in Sect. 3. The *Base Performance* line is the performance without cache blocking while *Best Performance* shows the performance with the optimum cache block size (found through an exhaustive search of power of two cache block sizes). The *Best RC with Analytic Model* line shows the performance if the cache block size was chosen by the analytic model. The best cache block size was found by running the analytical model for various power of two cache block sizes. The *Analytic Upper and Lower Bounds* show the performance predicted by the models. The *PAPI Model* line is the performance if the actual cache miss values found through hardware counters

Table 1 Matrix benchmark suite

	Application area	Dimension	Nonzeros	Density
1	Dense matrix	2000 × 2000	4000000	1.00
2	Statistical experimental design	231 × 319770	8953560	1.21e−1
3	Linear programming (LP)	52260 × 379350	1567800	7.91e−5
4	LP	10280 × 243246	1408073	5.63e−4
5	Latent semantic indexing	10000 × 255943	3712489	1.45e−3
6	column wise expansion of LSI	10000 × 2559430	3712489	1.45e−4
7	row wise expansion of LSI	100000 × 255943	3712489	1.45e−4
8	row wise stamping of LSI	100000 × 255943	37124890	1.45e−3
9	Queuing model of mutual exclusion	65535 × 65535	1114079	2.59e−4
10	LP problem from Italian Railways scheduling	4284 × 1092610	11279748	2.41e−3
11	Italian railways scheduling (small) (LP)	4284 × 546305	5661231	2.42e−3
12	Web connectivity graph (WG)	1000005 × 1000005	3105536	3.11e−6
13	WG after MMD reordering	1000005 × 1000005	3105536	3.11e-6
14	WG after RCM reordering	1000005 × 1000005	3105536	3.11e-6

Note that matrices 6, 7, and 8 are just modified versions of matrix 5

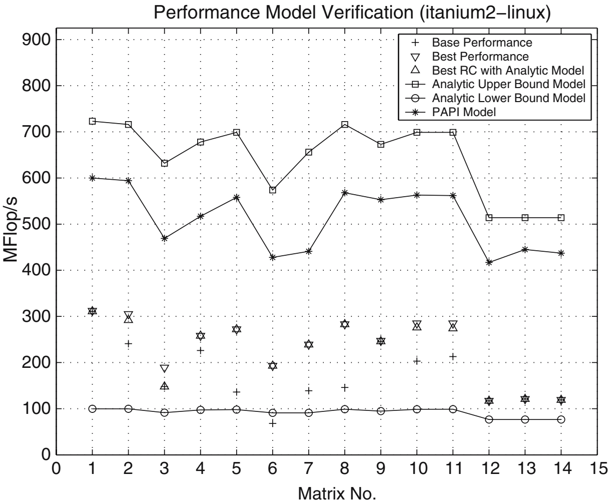

Fig. 3 Performance model for the Intel Itanium 2

were plugged into the execution time model. As shown by Fig. 3, the analytic model of Sect. 3 overpredicts performance by up to a factor of 2 on the Itanium 2, implying time still unaccounted for. However, the relative performance as a function of block size is well predicted [8], meaning we can use the model as a heuristic for choosing a good block size. Indeed, performance at the optimal block sizes chosen to maximize performance from the PAPI model are all within 90% of the best on Itanium 2, implying the model is a good heuristic if the miss models are accurate. Furthermore, except in the case of Matrix 3, the analytic model makes similarly good predictions on the Itanium 2, yielding 90% of the best performance. We also notice that the base performance for Matrix 6 on the Itanium 2 is lower than our predicted lower bound on performance. Further examination of the actual cache misses from Matrix 6 shows that actual number of misses are much closer to the modeled upper bound (much more so than other matrices) indicating that most access to the source and destination are causing cache misses. This could indicate that the memory latencies used to calculate the lower bound in performance are too optimistic. Future work would validate this hypothesis.

Figure 4 however shows that the heuristic is not as good on the Pentium 3 and the Power 4 compared to the Itanium2 since the TLB model from the Itanium 2 was used to describe the performance on the Pentium 3 and the Power 4. This motivates a different set of TLB model parameters (i.e., the η and ρ_T terms in Eq. (5)) for these platforms as described in Sect. 3.4. Availability of hardware TLB miss counters would help model validation on these and other platforms. Figure 4 also shows that the upper bound on performance underpredicts the overall performance for the Power4 on Matrix 1 which could be caused by the prefetch engine correctly predicting the access stream and therefore reducing the number of compulsory misses.

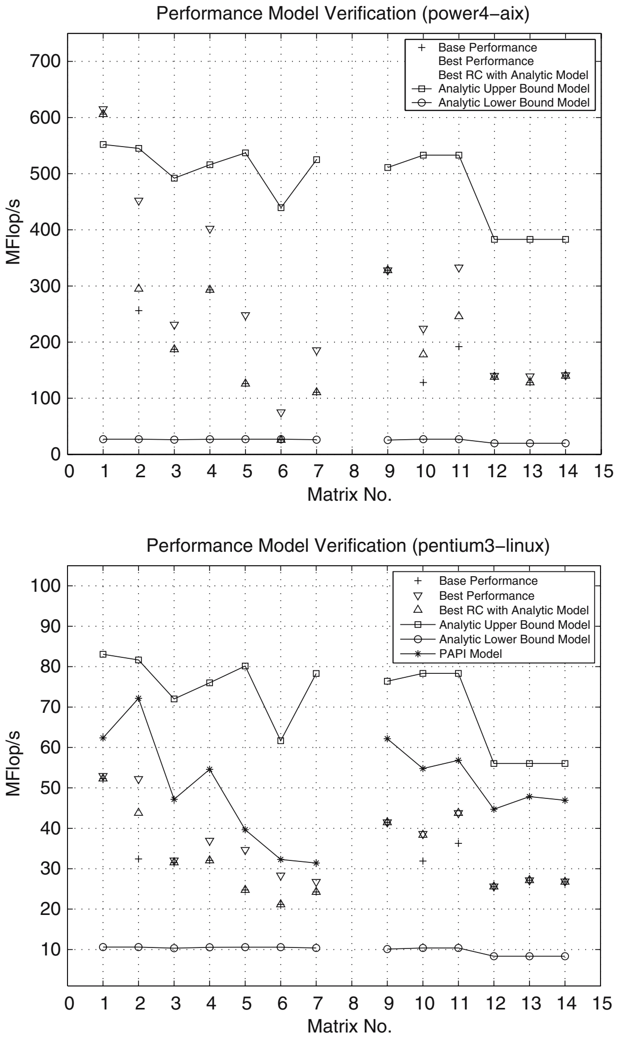

Fig. 4 Performance model for the IBM power 4 and Intel Pentium 3. Note that Matrix 8 was not run on these platforms due to memory limitations

5 Evaluation across matrices and platforms

Matrix structure

The speedups for each matrix varied across machines, but the best speedups (Table 2) were observed for the same matrices. The best speedups occurred with Matrices 2, 5–8, and 10–11. Except for Matrices 7 and 8, these matrices have small row dimensions and very large column dimensions, with nonzeros scattered throughout the matrix. Furthermore, the largest increases in cache

Table 2 **Speedups across matrices and across platforms**

	Matrix no.						
Platform	1	2	3	4	5	6	7
Itanium 2	1.00	1.27	1.28	1.14	**2.00**	**2.84**	**1.72**
Pentium 3	1.01	**1.61**	1.02	**1.15**	**1.40**	**1.33**	1.10
Power 4	1.01	**1.77**	1.24	1.37	**1.97**	**2.93**	1.68
	Matrix no.						
Platform	8	9	10	11	12	13	14
Itanium 2	**1.94**	1.00	1.40	1.34	1.00	1.00	1.00
Pentium 3	N/A	1.00	1.21	1.21	1.00	1.00	1.00
Power 4	N/A	1.00	**1.75**	1.73	1.01	1.09	1.01

This table shows the performance of the optimum cache block divided by the performance of the non-blocked implementation on that platform for that matrix. The highlighted values are the top four speedups on each platform

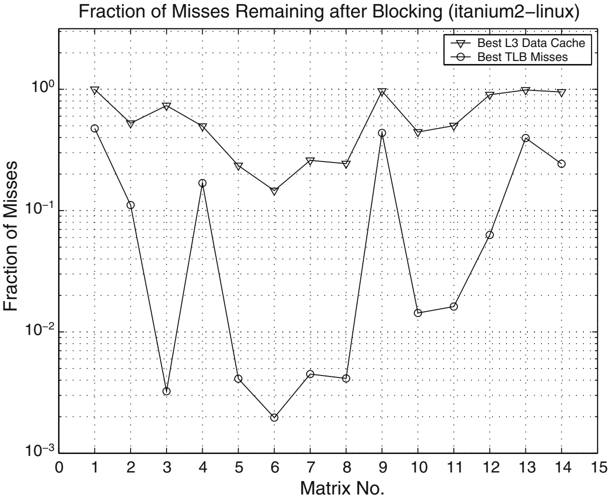

Fig. 5 Effect of cache blocking on L3 data cache and TLB on the Itanium 2. This plot shows the effect of cache blocking on the L3 Cache misses and TLB Misses, as measured by PAPI. The *Best L3 Data Cache* line is the number of cache misses with the optimum block size divided by number of cache misses that occur with the unblocked implementation. The *Best TLB Misses* line is an analogous line for the TLB. The matrices that showed the largest performance gains (Matrices 2, 5–8, and 10; see Table 2) also showed the greatest drop in L3 Cache misses implying that cache blocking is having the desired effect

misses as *c* increased occurred on the matrices with the largest speedups, implying that cache blocking increased locality.

Figure 5 shows the effect of cache blocking. As the plots show, the matrices with the largest speedups have the largest drop in the number of cache misses and TLB misses. In addition, Matrices 12–14 also show very little change in their optimum cache misses, implying that cache blocking has very little effect on these matrices. Matrices 12–14 are so sparse that there is effectively no reuse when accessing the source vector and thus blocking does not help, even though their source vector is large. Matrices with densities higher than 10^{-5} (all matrices except Matrix 3 and Matrices 12–14) were helped with cache blocking, provided that their column block size is large enough (greater than 200,000 elements, e.g., Matrix 2, Matrices 4–8, Matrices 10–11). There was enough reuse in x for the blocking to pay off.

We also find that in general matrices in which the row dimension is much less than the column dimension benefit the most from cache blocking. The smaller row dimension implies the overhead added by cache blocking is small since the number of rows themselves are limited. The larger column dimension implies that the unblocked implementations may lack locality. Even though Matrix 3 has a large column dimension, blocking did not yield much performance improvement. We performed additional experiments on random but banded matrices confirming theoretical work by Temam and Jalby [12]. As expected, cache blocking does not help when the band is relatively narrow because the natural access pattern to x is optimal, but pays off as the band grows. In this latter case, the RSE optimization smooths out differences in performance across block sizes [8].

Platform evaluation

Certain matrices such as Matrix 5 experienced significant performance gains through cache blocking on the Itanium 2 and the Power 4, but the speedup was less drastic on the Pentium 3. We expect that as the average number of cycles to access the memory grows, cache blocking will provide a good improvement in performance since cache blocking allows us to reduce expensive accesses to the main memory. The behavior of cache blocked SpM×V has a number of implications for architecture and systems. First, the TLB misses reduced by cache blocking can also be avoided by selecting large page sizes. Second, hardware support for cacheable and non-cacheable accesses to memory would be useful since only access to x is helped by caches, and not accesses to the matrix itself. Separate paths would prevent cache conflicts between matrix data and source vector data. In contrast, increased associativity only partially addresses this issue since it still allows premature eviction of “old” source vector elements by matrix elements. Future work might verify the impact of separate memory paths on the hybrid scalar-vector architecture of the Cray X1.

6 Conclusions and future work

Cache blocking significantly reduces cache misses in SpM×V particularly when x is large, y is small, the distribution of nonzeros is nearly random, and the nonzero density is sufficiently high. When these conditions appear in the matrix, we find that TLB misses are an important factor of the execution time. Our new performance bounds models incorporate the effect of TLB by implicitly modeling capacity and conflict misses ignored by our prior models [14,15]. Moreover, these new models predict optimal (or near-optimal) cache block size leading to speedups up to 2.93×.

Future work includes improving the accuracy of the miss models at all the levels in the memory hierarchy and obtain more accurate memory latencies. More accurate models should lead to even more accurate heuristics that decide when and how to cache block a sparse matrix, given the platform and matrix structure. Future work would also analyze the problem on novel architectures.

References

1. Bilmes, J., Asanović, K., Demmel, J., Lam, D., Chin, C.W. PHiPAC: A portable, high-performance, ANSI C coding methodology and its application to matrix multiply, University of Tennessee, LAPACK Working Note 111 (1996)
2. Browne, S., Dongarra, J., Garner, N., London, K., Mucci, P.: A scalable cross-platform infrastructure for application performance tuning using hardware counters. In: Proceedings of supercomputing, November (2000)
3. Fraguela, B.B., Doallo, R., Zapata, E.L.: Memory hierarchy performance prediction for sparse blocked algorithms. Parallel Proc Lett, **9**(3) (1999)
4. Gropp, W.D., Kasushik, D.K., Keyes, D.E., Smith, B.F.: Towards realistic bounds for implicit CFD codes. In: Proceedings of parallel computational fluid dynamics, pp. 241–248 (1999)
5. Heber, G., Dolgert, A.J., Alt, M., Mazurkiewicz, K.A., Stringer, L.: Fracture mechanics on the Intel Itanium architecture: a case study. In: Workshop on EPIC architectures and compiler technology (ACM MICRO 34), Austin, TX (2001)
6. Heras, D.B., Perez, V.B., Dominguez, J.C.C., Rivera, F.F.: Modeling and improving locality for irregular problems: sparse matrix-vector product on cache memories as a case study. In: HPCN Europe, pp. 201–210 (1999)
7. Im, E.-J.: Optimizing the performance of sparse matrix-vector multiplication. PhD thesis, University of California, Berkeley, May (2000)
8. Nishtala, R., Vuduc, R.W., Demmel, J.W., Yelick, K.A.: Performance modeling and analysis of cache blocking in sparse matrix vector multiply. Technical report (UCB/CSD-04-1335), University of California, Berkeley, EECS Dept. (2004)
9. Saad, Y. SPARSKIT: A basic toolkit for sparse matrix computations (1994) www.cs.umn.edu/Research/arpa/SPARSKIT/sparskit.html
10. Saavedra-Barrera, R.H.: CPU performance evaluation and execution time prediction using narrow spectrum benchmarking. PhD thesis, University of California, Berkeley, February (1992)
11. Snavely, A., Carrington, L., Wolter, N.: Modeling application performance by convolving machine signatures with application profiles (2001)
12. Temam, O., Jalby, W.: Characterizing the behavior of sparse algorithms on caches. In: Proceedings of supercomputing ’92 (1992)
13. Vuduc, R.W. OSKI: Optimized Sparse Kernel Interface (2005) http://bebop.cs.berkeley.edu/oski/
14. Vuduc, R., Demmel, J.W., Yelick, K.A., Kamil, S., Nishtala, R., Lee, B.: Performance optimizations and bounds for sparse matrix-vector multiply. In: Proceedings of supercomputing, Baltimore, MD, USA, November (2002)
15. Vuduc, R.W.: Automatic performance tuning of sparse matrix kernels. PhD thesis, University of California, Berkeley (2003)
16. Whaley, C., Dongarra, J.: Automatically tuned linear algebra software. In: Proceedings of supercomputer (1998)