Performance Optimization and Modeling of Blocked Sparse Kernels[*]

Alfredo Buttari[†], Victor Eijkhout[‡], Julien Langou[§] and Salvatore Filippone[¶]

December 20, 2006

Abstract

We present a method for automatically selecting optimal implementations of sparse matrix-vector operations. Our software 'AcCELS' (Accelerated Compress-storage Elements for Linear Solvers) involves a setup phase that probes machine characteristics, and a run-time phase where stored characteristics are combined with a measure of the actual sparse matrix to find the optimal kernel implementation. We present a performance model that is shown to be accurate over a large range of matrices.

1 Introduction

Sparse linear algebra computations such as the matrix-vector product or the solution of sparse linear systems lie at the heart of many scientific disciplines ranging from computational fluid dynamics to structural engineering, electromagnetic analysis or even the study of econometric models. The efficient implementation of these operations is thus extremely important; however, it is extremely challenging as well, since simple implementations of the kernels typically give a performance that is only a fraction of the peak speed.

At the heart of the performance problem is that sparse operations are far more bandwidth-bound than dense ones. Most processors have a memory subsystem considerably slower than the processor, and this situation is not likely to improve substantially any time soon. Consequently, optimizations are needed, likely to be intricate, and very much dependent on architectural variations even between closely related versions of the same processor.

The classical approach to the optimization problem consists in hand tuning the software according to the characteristics of the particular architecture which is going to be used, and according to the expected characteristics of the data. This approach yields good results but poses a serious problems where portability is concerned, since the software becomes tightly coupled to the underlying architecture.

The Self Adaptive Numerical Software efforts [4, 13] aim to address this problem. The main idea behind this new approach to numerical software optimization consists in developing software that is able to adapt its characteristics according to the properties of the underlying hardware and of the input data.

[*] This research was partly support by SciDAC: TeraScale Optimal PDE Simulations, DE-FC02-01ER25480
[†] Innovative Computing Laboratory, University of Tennessee, Knoxville, TN
[‡] Texas Advanced Computing Laboratory, The University of Texas at Austin, Austin, TX
[§] Department of Mathematical Sciences, University of Colorado at Denver and Health Sciences Center, CO
[¶] Tor Vergata University, Rome, Italy

We remark that the state of kernel optimization in numerical linear algebra is more advanced in dense linear algebra. The ATLAS software [13] gives near optimal performance on the BLAS kernels. Factorizations of sparse matrices (MUMPS [1, 15], SuperLU [8], UMFPACK [3, 17]) also perform fairly well, since these lead to gradually denser matrices throughout the factorization. Kernel optimization leaves most to be desired in the optimization of the components of iterative solvers for sparse systems: the sparse matrix-vector product and the sparse ILU solution.

In this document we describe the theory and the implementation of an adaptive strategy for sparse matrix-vector products. The optimization studied in this paper consists in performing the operation by blocks instead by single entries, which allows for more optimizations, thus possibly leading to faster performance than the scalar – reference – implementation. The optimized parameter is the choice of the block size, which is a function of the particular matrix and the machine.

An approach along these lines has already been studied in [12, 7] and, more recently, extended in [11]. We employ essentially the same optimizations, but relax one restriction in that research namely block-column alignment (see section 3.1 for further details). However, we have developed a more accurate performance model, which leads to better predictions of the block size, and consequently higher performance. Both the models presented in this paper and the model discussed in [12, 7, 11] are built with a technique that combines the results of a compile-time and a run-time analysis phases. This approach has been first presented in [6]. We will compare the accuracy of the models and the resulting performance numbers.

Other authors have proposed various techniques for accelerating the sparse matrix-vector product. For instance, Toledo ([10] and the references therein) mentions the possibility of reordering the matrix (in particular with a bandwidth-reducing algorithm) to reduce cache misses on the input vector. Pinar and Heath [9] also consider reordering the matrix; they use it explicitly to find larger blocks, which leads to a Traveling Salesman Problem.

While the reordering approach may undeniably yield an improvement, we have two reasons for not considering it. For one, in the context of a numerical library for sparse kernels, permuting the kernel operations has many implications for the calling environment. Secondly, our blocking strategy can equally well be applied to already permuted matrices, so our discussion will be orthogonal to this technique.

Blocking approaches have also been tried before. Both Toledo [10] and Vuduc [12] propose a solution where a matrix is stored as a sum of differently blocked matrices, for instance one with the 2×2 blocks, one with 2×1 blocks, and the third one with the remaining elements.

Our code will be released as a package 'AcCELS' (Accelerated Compressed-storage Elements for Linear Solvers); the AcCELS package is also planned for inclusion in a future release of the PSBLAS library [5].

In addition to the matrix-vector product, we also give a block-optimized version of the triangular solve operation. This routine is useful in direct solution methods (for the backward and forward solve) and in the application of some preconditioners.

In Section 2, we discuss general issues related to sparse linear algebra. In Section 3, we present a storage format that is appropriate for block sparse operations, and provide implementations for the matrix-vector product and the sparse triangular solve. We then give results and a performance analysis for the matrix-vector product. Because of the very similar structure of the operations, this discussion carries over to the Incomplete LU (ILU) solve.

2 Optimization of sparse matrix-vector operations

Matrix-vector multiplication and triangular system solving are very common operations in sparse linear algebra computations. These two operations typically account for more than half of the total time spent in the solution of a linear sparse system using an iterative method; moreover, they tend to perform very poorly on modern architectures. There are several reasons for the low performance of these two operations:

- **Indirect addressing / Low ratio between floating-point operations and memory operations:** Sparse matrices are stored in data structures where, in addition to the values of the entries, the row indices or the column indices have to be explicitly stored. The most common formats are Compressed Sparse Row (CSR) and Compressed Sparse Column (CSC) storage [2, §4.3]. This means that, apart from the elements of the matrix, the indices also have to be explicitly read from memory which leads to a high consumption of the CPU-memory bandwidth. Basically, there are two reads per floating-point multiply-add operation. The ratio is one in the dense case. Moreover retrieving and manipulating the column/row indices informations implies an amount of integer operations that is not negligible.
- **High per row overhead:** The sparse matrix-vector product compares unfavourably with the dense case when we consider loop overhead. Since there typically are far fewer elements per row in the sparse case, any existing overhead is relatively more important in the sparse case. This includes both the loop overhead, and the cost of the write-back operation. Furthermore, the inner loop has dynamically computed bounds, preventing the compiler from applying several optimizations.
- **Low spatial locality:** During the matrix-vector product, in the case of CSR storage of the matrix (resp. CSC) the discontinuous way the elements of the source vector (resp. destination vector) are accessed is a bottleneck that causes low spatial locality. Typically, we do not expect any loaded cache lines to be fully utilized.
- **Low temporal locality:** In order to minimize memory access, it is important to maximize the number of times a data item is reused. During a sparse matrix-vector product with a matrix stored in Compressed Sparse Row (CSR) format, the elements of the matrix are accessed sequentially in row order and are used once, while the elements of the destination vector are accessed sequentially and each of them is reused as many times as the number of elements in the corresponding row of the sparse matrix which is optimal with respect to the temporal locality.
  On the other hand, the elements of the source vector x would be reused during the matrix-vector product when their row indices belongs to two (or more) nearby rows of the matrix A where there are elements on the corresponding column. Such rows in general need not exist, which implies that reuse of x is not guaranteed.

The optimization of the sparse matrix-vector operations presented in this paper consists in *tiling* the matrix with small dense blocks that are chosen to cover the nonzero structure of the matrix.

Below, we will discuss in detail the way in which this affects performance. For now we note two considerations that need to be balanced:

- Use of small tiles causes an improvement in scalar performance due to reduced indexing and consequent reduction of data traffic, and improved spatial and temporal

locality. Since this is strictly a function of the architecture, albeit a nontrivial one, we evaluate this factor in the **installation phase** of the AcCELS software.
While increased block size leads to diminished overhead in a regular manner, it also exhausts processor resources in a less predictable way, so the installation phase will be an empirical evaluation of the performance of different block sizes.

- Unfortunately, the number of operations increases due to the operations performed on the zeros stored in the dense tile blocks (this phenomenon will be referred to as *fill-in*). There is then a trade-off with the theoretically optimal tile size, and this can only be decided in a **runtime phase**, when the actual matrix structure is known.

We will discuss both factors in considerable detail in the remainder of this paper.

Previously, the ATLAS project [13] has been singularly successful in optimizing dense linear algebra kernels. The ATLAS strategy consists of optimizing the different algorithmic parameters to the architecture in a installation phase. This optimization can be done completely at installation time, since performance is a function only of architecture parameters, and not of the actual matrix. In the sparse case, the structure of the matrix has a great influence on the optimal parameters and the resulting performance, so a dynamic phase is needed where the part of the analysis that depends on the matrix sparsity structure is performed.

3 The block sparse matrix format

In this section we present the block sparse matrix storage format, and the implementation of the matrix-vector multiply and the triangular solve kernels.

3.1 The BCSR storage format

The Block Compressed Sparse Row storage format for sparse matrices exploits the benefits of data blocking in numerical computations. This format is similar to the CSR format except that single value elements are replaced by dense blocks of general dimensions $r \times c$. Thus a BCSR format with parameters $r = 1$ and $c = 1$ is equivalent to the CSR format. All the blocks are row-aligned which implies that the first element of each block (i.e., the upper leftmost element) has a global row index that is a multiple of the block row dimension r. We can choose whether or not to let the blocks also be column-aligned.

A matrix in BCSR format is thus stored as three vectors: one that contains the dense blocks (whose elements can be stored by row or by column); one that contains the column index of each block (namely the column index of the first element of each block); and one which contains the pointers to the beginning of each block-row inside the other two vectors (a block row is a row formed by blocks, i.e. an aligned set of r consecutive rows).

Formally (in Fortran 1-based indexing),

```
for j=ptr[i]...ptr[i+1]-1:
    for k=1...(r*c):
        elem[(j-1)*r*c+k] contains
         $A((i-1)*r+(k-1)/c+1, \texttt{col\_ind}[j]+\mathrm{mod}(k-1,c)+1)$
```

All elements of the matrix A belong to a small dense block; this means that when the number of nonzero elements is not enough to build up a block, we explicitly store zero values to fill the empty spaces left in the blocks. These added zero values are called fill-in elements.

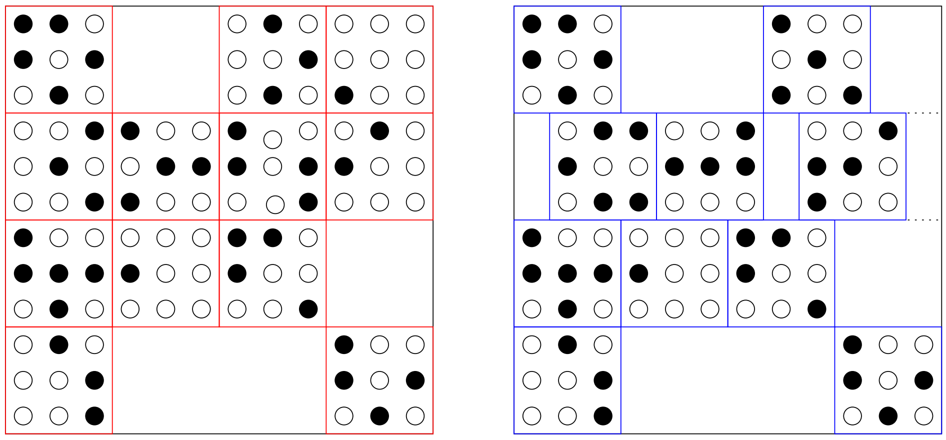

Figure 1: Fill-in for 3×3 row and column aligned blocks (left) and row aligned but column unaligned blocks (right).

Figure 1 (*left*) shows the tiling of a 12×12 matrix with 3×3 row and column aligned blocks. The black filled circles are the nonzero elements of the matrix while the empty circles are zero elements added. The fill-in ratio is computed as the ratio between the total number of elements (original nonzeros plus fill-in zeros) and the nonzero elements; for the matrix in Figure 1 (*left*) with 3×3 block size the fill-in ratio is 2.8. Performing the matrix-vector product with the matrix in Figure 1 (*left*) stored in BCSR format with 3×3 block size, 2.8 times as many floating point operations as in the case of the CSR format have to be executed.

Fortunately, in most sparse matrices the elements are not randomly distributed, so such a block tiling often makes sense. Either the matrices have an intrinsic block structure (in which case the fill-in is zero), or elements are sufficiently clustered so that it is possible to find a block size for which the fill-in is low.

We can often get a lower fill-in ratio by relaxing the limitation that the blocks be column aligned. Each block inside a block row begins at a column index that is not necessarily a multiple of the column size c. While this choice increases the time spent during the matrix building phase since more possibilities have to be evaluated, it has no extra overhead during the matrix-vector product operation. Figure 1 (*right*) shows the tiling of the same matrix with 3×3 row aligned but column unaligned blocks. In this case the fill-in ratio is reduced to 2.36.

3.2 BCSR kernels

In this section we describe the implementation of the matrix-vector product and the triangular system solve for a matrix stored in BCSR format.

3.2.1 The matrix-vector product

The source code for the matrix-vector product $y \leftarrow y + Ax$ with A with a tiling block size of 2×3 is given in Figure 2. The code consists of two loops: the outer is over the number of block-rows, while the inner loop is over the number of blocks in each row. The partial result of the product of each row is held in accumulators `y0,y1` and the code for

```
for(i=0;i<*m;i++,y+=2){
    int j;
    register double y0=y[0];
    register double y1=y[1];
    for(j=ia2[i];j<ia2[i+1];j++,ia1++,aspk+=6){
        y0 += aspk[0]*x[*ia1 +0];
        y1 += aspk[3]*x[*ia1 +0];
        y0 += aspk[1]*x[*ia1 +1];
        y1 += aspk[4]*x[*ia1 +1];
        y0 += aspk[2]*x[*ia1 +2];
        y1 += aspk[5]*x[*ia1 +2];
            }
    y[0]=y0;
    y[1]=y1;
}
```

Figure 2: Source code that implements the sparse matrix vector product for matrices stored in BCSR format for blocks of size 2×3.

the product of the small dense block with a piece of x is completely unrolled. Each dense block is stored in the array `aspk` in a row-wise order.

3.2.2 The triangular system solve

The triangular system solve operation can be performed on a triangular matrix that possibly has a unit diagonal. In the case of a unit diagonal we use the same data structure that we use for a general sparse matrix; in the general case we force the blocks on the diagonal to be squares of dimension $r \times r$, thus we need an additional array `D(:)` to store them. The code for the lower triangular system solve $Lx = b$ in the case of non unitary diagonal matrix with 2×3 blocks is given in Figure 3. The code is very similar to the one for the

```
...
double *xp=x;
for(i=0; i<*m; i++, xp+=2, b+=2, d+=4){
  register double x0=b[0];
  register double x1=b[1];
  for(j=ia2[i]; j<ia2[i+1]; j++, aspk+=6){
    x0-=aspk[0]*x[*ia1+0];
    x1-=aspk[3]*x[*ia1+0];
    x0-=aspk[1]*x[*ia1+1];
    x1-=aspk[4]*x[*ia1+1];
    x0-=aspk[2]*x[*ia1+2];
    x1-=aspk[5]*x[*ia1+2];
  }
  //Solve small system on the diagonal
  x1-=d[0][1]*x0;
  xp[1]=x1/d[1][1];
  xp[0]=(x0-d[0][1]*x1)/d[0][0];
}
...
```

Figure 3: Source code that implements the sparse triangular system solve for matrices stored in BCSR format for blocks of size 2×3.

matrix-vector product except for the fact that at the end of each block-row there is a small triangular system solution.

4 Performance optimization and modeling

In this section, we present a model for the performance of the block sparse matrix-vector product. The time spent for a matrix-vector product of a matrix A can be computed as the ratio between the flop rate at which it is performed and the number of floating-point operations executed. Since the number of floating-point operations performed is proportional to the fill-in ratio, we have:

$$\text{time} \propto \frac{\text{fill}_A(r,c)}{\text{perf}_A(r,c)} \tag{1}$$

where $\text{fill}_A(r,c)$ and $\text{perf}_A(r,c)$ are respectively the fill-in ratio and the matrix-vector product performance rate for a given $r \times c$ block size. Thus the best choice for the block size (i.e., the one that results in the lowest time spent for the matrix-vector product operation) is the one that minimizes the ratio in equation (1). The exact knowledge of the numerator and denominator in equation (1) requires performing the matrix-vector product itself. An exhaustive search through r, c space is thus possible, but also quite expensive. We therefore limit ourselves to computing some estimates for these two values instead. We compute $\text{fill}'_A(r,c)$ and $\text{perf}'_A(r,c)$ for every relevant block size and minimize the quantity

$$\frac{\text{fill}'_A(r,c)}{\text{perf}'_A(r,c)} \tag{2}$$

Section 4.1 explains how the fill-in is estimated; Section 4.2 deals with how the performance optimization is automated.

4.1 Estimating the fill-in

The first step in predicting the performance of the matrix vector product of a matrix A, with a $r \times c$ tiling, is estimating the fill ratio $\text{fill}'(r,c)$.

We use the method proposed in [12]: we sample a number of matrix rows and compute their individual fill-in. The fill-in of the whole matrix is assumed to be the same as the fill-in of this sample. Specifically, we introduce a parameter acc ($0 \leq acc \leq 1$) for the user to control the number of rows used to estimate the fill-in. Given a total number of block rows $m = \lceil n/r \rceil$ for a given value of r (the block-row dimension), the fill-in is computed for $m \cdot acc$ block rows of the matrix.

Since a matrix need not be uniform in structure, we use the following strategy to ensure that we sample fairly. First we divide the matrix in $m \cdot acc$ parts; then in each of these parts a block row is selected randomly. If A' is the submatrix composed of the selected $m \cdot acc$ block rows, the operation performed at this phase can be formalized as

$$\text{fill}'_A(r,c) = \text{fill}_{A'}(r,c).$$

A value of $acc = 1$ causes the whole matrix to be evaluated, which is the most accurate choice, but it may be too expensive. If the matrix has a regular pattern, or if setup time is at a premium, a small value of acc can be taken. The default value for acc used in AcCELS (and SPARSITY) is $acc = 0.2$.

4.2 Modeling block matrix performance

The second step in predicting the performance of the matrix vector product of a matrix A, with a $r \times c$ tiling, is estimating the expected performance $\text{perf}'_A(r,c)$. We will first discuss abstractly the influence of the block size parameters r, c, and then discuss two strategies for estimating the performance of a full matrix-vector product.

4.2.1 Influence of the block size on performance

As is apparent from the code examples above, use of the BCSR storage format improves the performance of the matrix-vector product since $r + c$ registers store elements of the source vector x for reuse, and elements of the destination vector y to minimize writes back to main memory. What one would expect is that performance grows with r and c until blocks size becomes too big and register spilling happens. However, in practice, it is not possible to predict the performance of using a certain block size theoretically, so (as we will see in the experiments in section 4.2.3 and 4.2.4), we have to perform an exhaustive search through all the possible block sizes where r ranges from 1 to r_{max} and c ranges from 1 to c_{max}. The default value for values r_{max} and c_{max} is 10. On all the matrices used in our tests, block sizes greater than 10 gives unreasonably large fill-in, so there is no overall performance gain to be expected.

In Figure 4 we plot the speed of the matrix-vector product operation measured in Mflop/s obtained for all the possible $r \times c$ block sizes from 1×1 to 10×10 on an Itanium2 machine. The matrix used is a 1500×1500 dense matrix stored in BCSR format. The highest speedup with respect to the reference CSR implementation (or the 1×1 BCSR) is obtained for the 8×8 block size, with a value of 4.32. The effect of the register spilling is visible on the

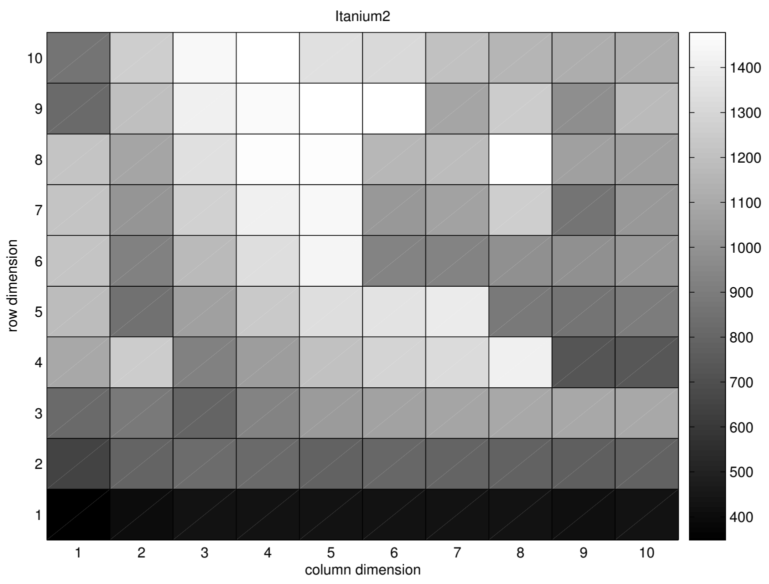

Figure 4: Matrix-vector product flop rate for a 1500×1500 dense matrix stored in BCSR format on an Itanium2 architecture.

upper rightmost part of the graph. With increasing r there is increasing reuse of the source vector x, so we expect an increase in performance.

The behaviour observed here can be explained qualitatively to a degree, but is not easily modeled and predicted quantitatively, hence the need for an exhaustive test. Figure 5 shows the same information as the previous image for a SGI octane machine with a R12000 processor; also in this case considerable speedup over the reference case can be observed (specifically, 2.13 for 10×10 blocks).

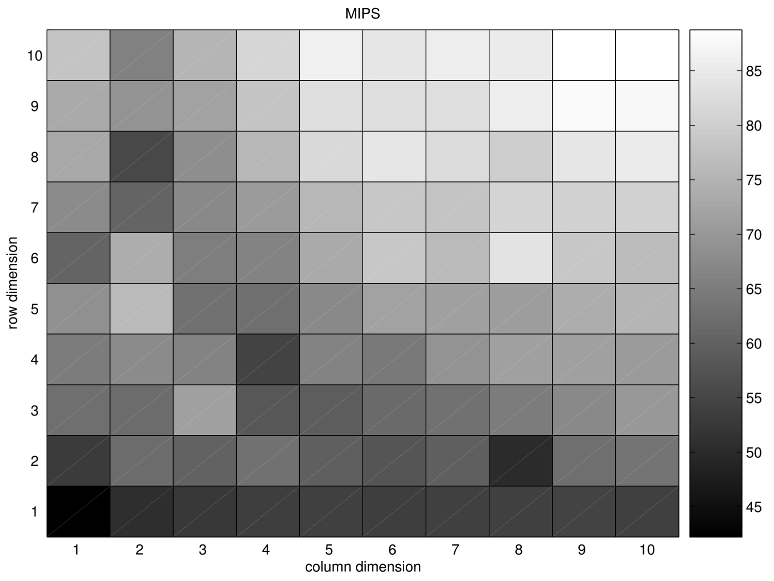

Figure 5: Matrix-vector product flop rate for a 1500×1500 dense matrix stored in BCSR format on a MIPS architecture.

4.2.2 Interaction between block performance and fill-in

These results above do not by themselves determine the reduction of the time required to perform a matrix-vector product operation on a real sparse matrix. Sparse matrices, in fact, are affected by the presence of fill-in elements when stored in the BCSR storage format. This means that the amount of floating point operations needed to perform the matrix-vector product operation increases by a factor that is equal to the fill-in ratio. Figure 6 reports the flop rate (*top left*), fill-in ratio (*top right*) and the matrix-vector product execution time (*bottom left*) for a sparse matrix from a real world application (matrix *venkat01*) on an Itanium2 machine.

Comparing the graphs in figure 6 we can see that even if the highest flop rate is for block dimension 8×8 (4.09 Mflop/s), the fastest matrix-vector product is for block dimension 4×2 because of lower fill-in.

4.2.3 Performance modeling by dense matrix

In this section we present the implementation of the performance prediction method that is used in [12]. In that research the performance of a matrix-vector product with a $r \times c$ tiled matrix is estimated to be that of a dense matrix tiled with those values of r and c:

$$\mathrm{perf}'_A(r,c) = \mathrm{perf}_{Dense}(r,c).$$

In effect, this ignores the influence of the sparsity structure, an assumption that we will argue below is unwarranted.

Once $\mathrm{perf}_{Dense}(r,c)$ has been evaluated for the different block sizes, the resulting flop rates of these tests are stored in a file and then accessed during the preprocessing phase of the matrix-vector products.

The block size selection (performed at run-time) for this strategy consists of:

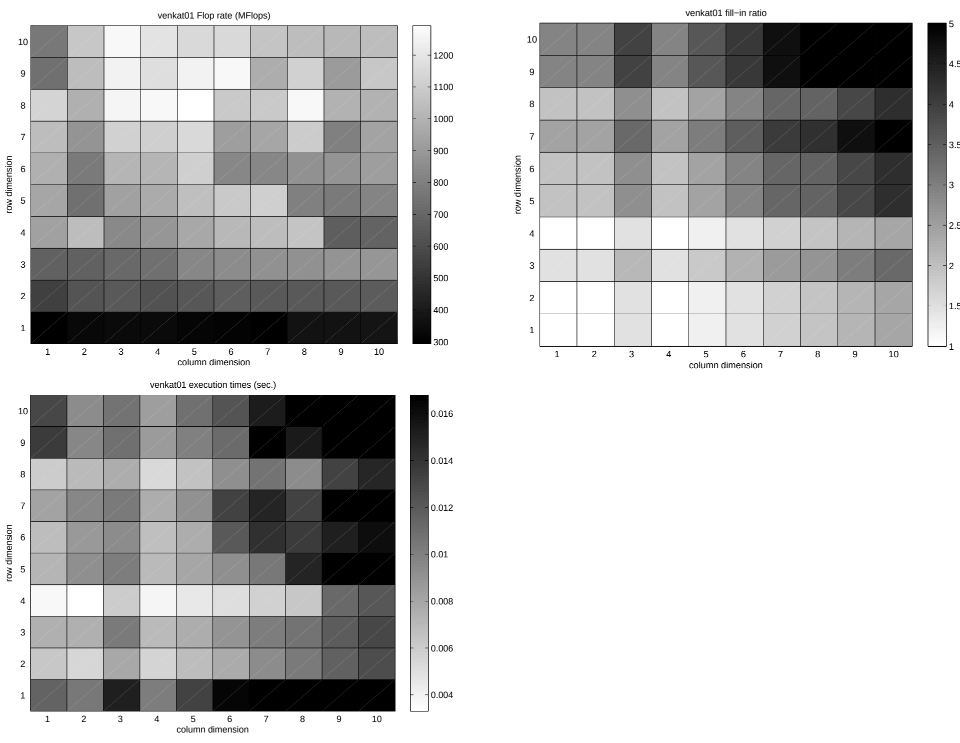

Figure 6: This figure shows how different choices for r and c affect execution times through flop rates and fill-in ratios. In the each part of the figure assume that “white is better” (i.e. higher flop rates, lower fill-in ratios and lower execution times). On the top-left part is depicted how flop rate changes with different values of r and c, on the top-right part is depicted how fill-in changes with r and c and on the bottom-left part is depicted how the execution times change with r and c.

1. Reading the file built at installation-time phase that contains the performance information $\mathrm{perf}'_A(r, c)$ for each r and c.
2. Estimating the fill-in $fill'_A(r, c)$ for each r and c, as described in 4.1.
3. Selecting the block size for which $\frac{fill'_A(r,c)}{\mathrm{perf}'_A(r,c)}$ has the minimum value.

We add an optimization to the strategy in [12]. Considering two block sizes $r \times c$ and $r \times c'$ such that (a) c' is a sub-multiple of c and (b) the performance obtained for the $r \times c'$ block is higher than the one for the $r \times c$ block, then there is no use to consider the block size $r \times c$. If, for example, the 4×2 blocks size gives better performance than the 4×4, it is not worth considering this last block size because each small 4×4 block is the same as two 4×2 blocks and then we would have exactly the same fill-in but lower performance. The gain of applying this tuning can be considerable.

4.2.4 AcCELS performance model

The main reason why the performance prediction method described above might be inaccurate is that the performance of the matrix-vector product is affected by the sparsity structure of the matrix. Tests we have done show the influence of two different parameters on the performance of the matrix-vector product: the number of elements per row and the spread of elements in each row.

Number of elements per row To understand the impact that this parameter has on the performance of the matrix-vector product let us consider the code of the matrix-vector product for the 1×1 block size case (that is the CSR case) reported in figure 7.

```
...
for(i=0;i<*m;i++,y+=1){
  register double y0=y[0];
  for(j=ia2[i];j<ia2[i+1];j++,ia1++,aspk+=1){
    y0 += aspk[0]*x[*ia1+0];
  }
  y[0]=y0;
}
...
```

Figure 7: Source code that implements the sparse matrix vector product for matrices stored in CSR format.

The product is performed row-wise and for each row the partial result is held in an accumulator `y0`. At the end of the loop for a given row, the value in the accumulator is written back to memory. Thus for each row we have $2 \times elem\_row$ floating point operations, where $elem\_row$ is the number of elements per row, and a write memory access. Given that a write memory access is more expensive than a floating-point operation, we expect a higher performance for matrices with a large (average) number of elements per row. This is confirmed by the data plotted in Figure 8 which describes the flop rate of the matrix-vector product for matrices with different numbers of elements per row in the case of 1×1 block size.

The irregularities in the plot data for the real world matrices can be attributed to the fact that individual rows can have any number of nonzeros, perturbing the performance with respect to a banded matrix. Also, the test matrices can have arbitrary bandwidth, which influences spatial locality in the matrix-vector product.

The AcCELS performance model takes the sparsity characteristics of the matrix into account to have a better estimate of the performance. The main aim is to better predict performance for matrices with low number of elements per row. Matrices with a low number of elements per row are very common in practice: more than 50% of the matrices in the Matrix Market collection [14] (resp. the University of Florida matrix collection [16]) have less than 7 (resp. 8) elements per row (as of this writing).

In figure 8, using a dense matrix to model the performance rate for a sparse matrix is equivalent to using the asymptotic flop rate value. This is seen to lead to a misprediction by a factor of 3 for more than half of the sparse matrix available in those two standard collections. We expect that our improved model leads not only to a better prediction of the performance for a given block size but also enables us to have a better selection strategy in practical cases.

A simple implementation of this strategy consists of computing the curve in Figure 8 for each block size, and storing it for reference. The main drawback of this approach is that it need considerable data storage that need to be accessed during the setup phase. Moreover

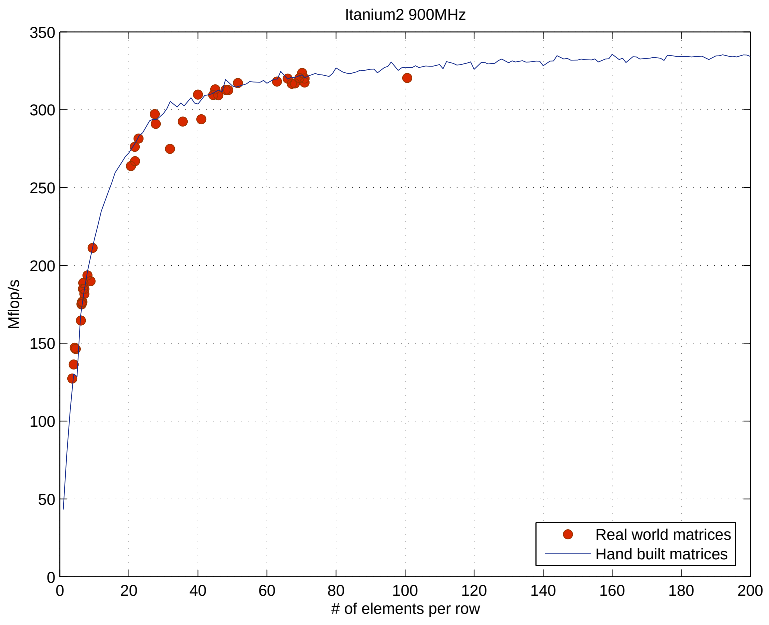

Figure 8: Flop rate for matrices with different number of elements per row. The curve plots the performance of banded sparse matrices while the circles plot the performance rate for a set of sparse matrices from real-world applications. An Itanium2 architecture was used.

such an approach is prone to spurious timings resulting in unreliable values of the flop rate. Instead we use a parametric model for these curves.

For each row in the sparse matrix-vector multiply, the following operations are involved:

- Loop overhead and index/bound calculations;
- One update of the result vector;
- A number of additions and multiplications proportional to the number of nonzeros in the row.

This means that the time spent in the computations performed on each (block) row can be modeled as $c_1 + c_2 \cdot elem\_row$ where $elem\_row$ is the number of (block) nonzeros, the number of operations is itself proportional to $elem\_row$. Finally the corresponding flop rate (number of operations divided by time), perf"$_A(r, c)$, is expected to follow a hyperbola which we model as follows[1]:

$$\underset{A}{\text{perf"}}(r, c) = \alpha + \frac{\beta}{elem\_row + \gamma}. \tag{3}$$

where α is equal to perf$'_A(r, c)$, the performance rate for the dense matrix. $\beta/(elem\_row + \gamma)$ is the correction we proposed to add in order to have a more accurate model. β is negative, γ is positive, so that the negative correction term gets larger for smaller $elem\_row$.

Figure 9 shows that the curve that is measured for the 1×1 block size case, and the curve that is built for the same block size case with the regression model (3), are identical for our purposes. Similarly, we observe that the curves plotted for the possible block sizes are all within a few percent of the model (3).

1. Strictly speaking, one of the three parameters can be eliminated, since $\alpha \cdot \gamma = -\beta$ This models the constraint perf"$_{nnz=0}(r, c) = 0$. However, we keep the third parameter to better deal with noisy or irregular data.

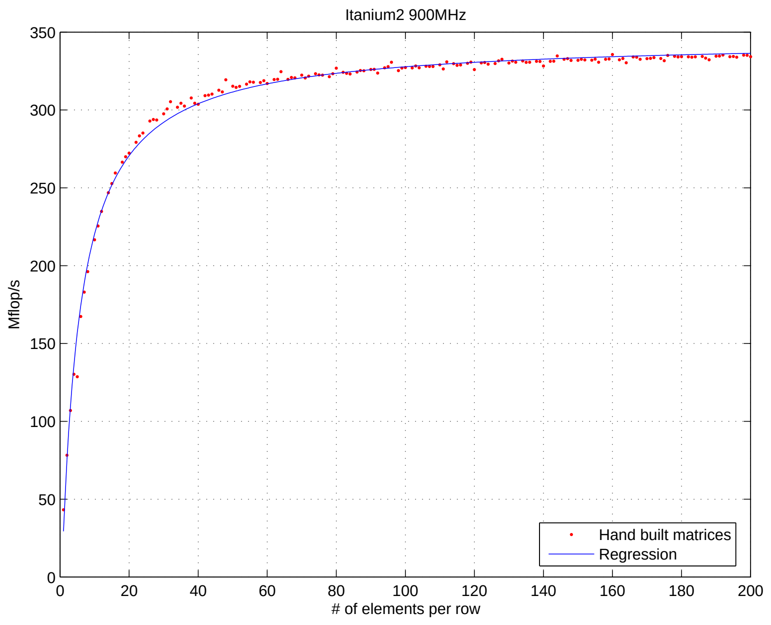

Figure 9: Comparison between the measured performance vs number of elements per row for banded matrices (dots) and the curve built with the regression method (curve). Itanium2 architecture.

Distance between the elements The distance between the elements of a matrix influences both the spatial and temporal locality in the accesses of the source vector. If the elements in a row are close to each other spatial locality is improved: depending on the cache line length there is an higher probability of having elements of the source vector that are brought inside one cache line. The likelihood of a source element being reused for a next matrix row is also higher.

Conversely, column indices spread far apart are more likely to lead to TLB conflicts.

The curves in Figure 10 plot flop rate versus number of elements per row of matrices with different bandwidth. The matrices are hand built and on each row the column indices are randomly generated inside a band around the diagonal.

In Figure 10 we observe that the matrices with the elements confined to a more narrow bandwidth (curve with ○ markers) have higher performance than those with a large bandwidth.

While clearly the distribution of the elements in a row can affect the performance of the matrix-vector product, Figure 8 indicates that the such cases may be exceptional. Also, while we were able to model the behaviour induced by varying numbers of elements per row (see below), modeling the distribution proved elusive. For these reasons, our model limits itself to the influence of the number of nonzeros per row of the matrix.

4.3 Performance modeling and optimization procedure

We summarize the above by giving a step-by-step description of the optimization process.

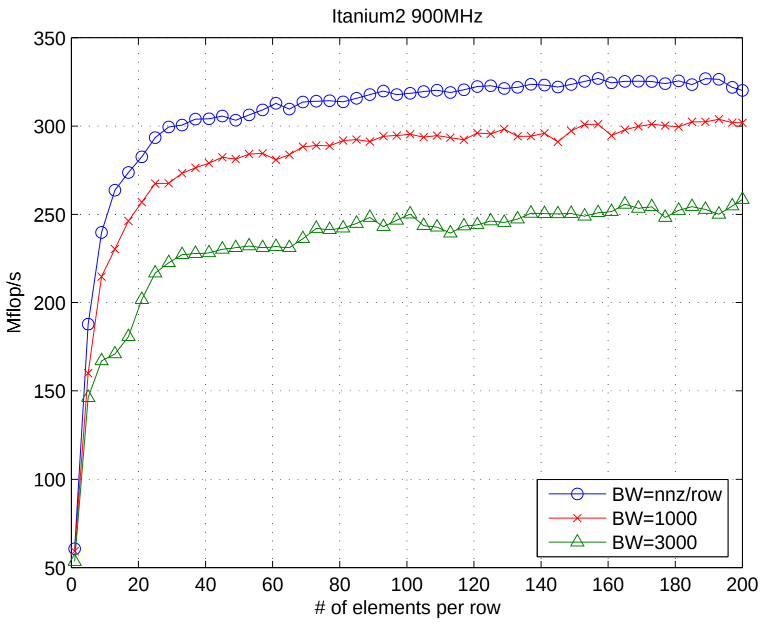

Figure 10: Flop rate versus number of elements per row for different bandwidths. The line with $\circ$ markers plots the performance on banded matrices (i.e. the bandwidth is as low as possible), the one with $\times$ markers plots the performance on sparse matrices with bandwidth=5000 while the one with $\triangle$ markers plots the performance rate sparse matrices with bandwidth=50000. An Itanium2 architecture was used.

At installation-time, for each block size, the matrix-vector product is performed for a small number of different numbers of elements per row; the curve parameters (α, β and γ) are then computed using a least-squares fitting method and finally the parameters α, β and γ are tabulated for all the block sizes.

The matrices used during this process are automatically generated banded matrices, and the least squares fitting method is composed by a linear regression phase and a non-linear one: the linear regression phase is used to build an initial guess for the non-linear one, then the iterative non-linear technique is used to optimize the fitting. The variables of the correction needs to satisfy $\beta \leq 0 \leq \gamma$; if the data are very messy, the regression might violate this condition (this has happened on some architectures for some block sizes). In such a case, we set $\beta = \gamma = 0$ and α equal to the mean value of the computed performance rates. This reduces our strategy to the one used in [12] for these problematic cases.

With the information gathered at installation-time, we use our performance model at run-time to predict the performance of a matrix-vector operation as follows. For each (r, c) pair we evaluate the following steps:

- Let the fill-in ratio $fill'_A(r, c)$ be calculated as described in 4.1.
- the parameters α, β and γ of the rational function equation (3) are read from the file built at installation-time. phase.
- the number of elements per row is computed as:

$$elem\_row = \frac{nnz}{m} \times fill'_A(r, c) \qquad (4)$$

where nnz is the number of nonzero elements in the matrix and m is the size of the matrix.

- the performance estimate is computed as:

$$\mathrm{perf}'_A(r,c) = \alpha + \frac{\beta}{elem\_row + \gamma} \tag{5}$$

Now the block size $r \times c$ is chosen such that the quantity (1) is minimized.

4.4 Cost of the dynamic setup

Estimating the amount of fill-in for a given block size, and subsequent conversion of a matrix to block storage, is a relatively costly operation. While the exact cost depends on the matrix, the architecture, and the value of acc used, in our experiments on a collection of test matrices it rarely exceeded the cost of 10 matrix-vector products using the reference implementation. The difference between the conversion cost for aligned and unaligned blocks (see section 3) was in general between factor of 2 and 3. Our AcCELS software has a parameter for the user to disable unaligned blocks.

5 Numerical tests

In this section we report the results of our block-size selection strategy compared with results obtained using the SPARSITY software described in [12, 7]. Table 1 shows the most relevant characteristics of the architectures we used.

	AMD Athlon 1200	MIPS	Power3	Itanium2
Proc. type	AMD Athlon k6	MIPS R12000	IBM Power3	Genuine Intel IA-64 Itanium2
Proc. freq.	1200 MHz	270 MHz	375 MHz	900 MHz
Cache size	64 KB L1 256 KB L2	32 KB L1 2 MB L2	64 KB L1 8 MB L2	32 KB L1 256 KB L2 1.5 MB L3
Memory size	256 MB	256 MB	1 GB	8 GB
OS	GNU-Linux	IRIX64 6.5	AIX 5.1	Red Hat Linux 3.2.3
Compilers	Intel compilers v9.0	MIPSpro Compilers v7.41	IBM xlc and IBM xlf v6.0	Intel Compilers v9.0

Table 1: Details of the architectures used to test and tune the performance model presented.

We start by devoting some attention to the proper construction of a timer for the sparse operations.

5.1 Implementation of the timing routine

As a general principle, a timing routine should reflect the conditions in which the code is used. In our case, we cannot expect the matrix to stay resident in cache: even if the matrix is small enough to fit inside the cache, the fact that it is in general used in conjunction with other computational routines (e.g., in an iterative solver) means that the matrix is likely to be flushed from the cache between applications of the product routine. Thus, a tester that repeatedly applies a small matrix to an input vector will give an unrealistically high flop rate since the matrix stays resident in the cache.

We prevent artificially high flop rates by allocating a data set larger than the largest cache size – in fact, to account for cache associativity and random-replacement strategies we allocate several times the cache size – and filling this with multiple copies of the matrix-vector problem. All the matrices and vectors in the data set are the same but a different memory area is used for each of them, so that any two consecutive matrix-vector products will be identical in behavior, but operating on different data. The time for a single matrix-vector product is computed as the average time for the matrix-vector product of all the matrices in the data set. Figure 11 shows how data cache influences the measure of performance. The curves plot the performance of the matrix-vector product versus the number of elements per row: the ○-line reports the case where cache effect is not avoided (i.e., data set that includes only one matrix) while the ×-line reports the case where timings are performed on a data set bigger than the data cache size. As can be expected, the impact of the cache

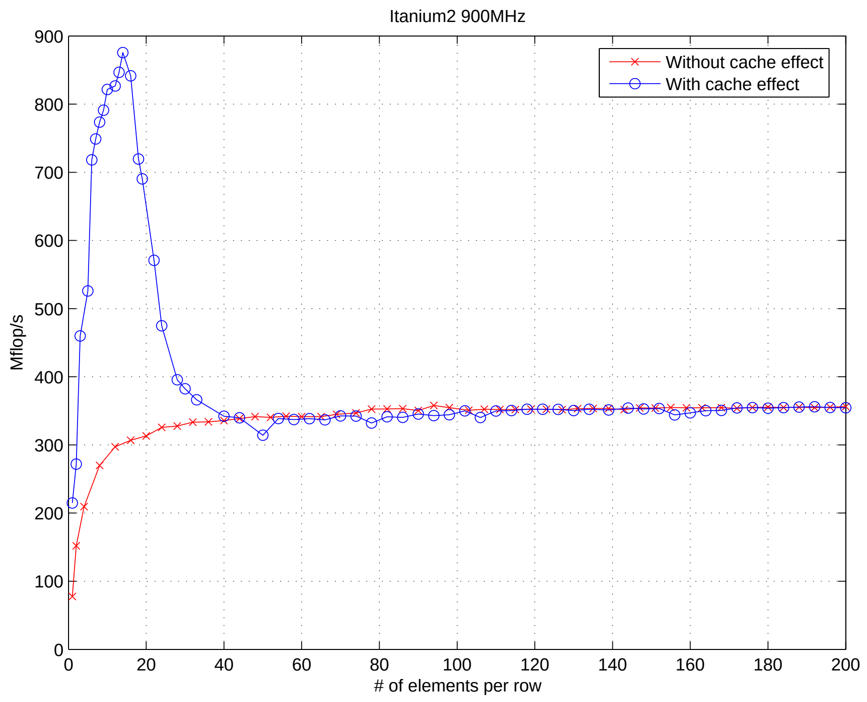

Figure 11: Comparison between timings with (line with ○ markers) and without (line with × markes) cache effects. This data is computed using a banded sparse matrix on an Itanium2 machine.

is only visible for matrices small enough to fit in cache. Note that since the dimension of the matrix is fixed, a bigger number of elements per row means a higher density and thus requires a larger memory.

In [12, 7], the matrices studied are large enough to automatically flush the cache. Given that some matrices in our test set have smaller dimensions, and that newly released processors have increasingly large caches, it is necessary to adopt a timer that is guaranteed to obtain reliable measurements for a truly portable and 'future-proof' package. Thus even when comparing with performance obtained with SPARSITY we will refer to the timings measured with our proposed timer.

matrix	Predicted perf. (Mflop/s)	measured perf. (Mflop/s)	actual perf. Mflop/s	matrix size (MegaBytes)	elem. per row
raefsky3	1409	1315	1298	11.35	70.2
shyy161	720	386	370	2.51	4.3
mcfe	1152	1300	964	0.186	31.9
jpwh_911	397	308	182	0.045	6.1

Table 2: Predicted versus measured versus actual performance with SPARSITY code [12, 7]. Itanium2 architecture.

Table 2 illustrates the importance of a well designed timer, as well as our performance model. This table gives the predicted performance $perf'_A(r, c)$ of the dense matrix model; the measured performance relates to the timing method used in [12, 7]; the actual performance is the performance measured with our improved timer. In each case, the block size selected by the dense matrix model is used.

Numbers reported in this table are measured on an Itanium2 architecture and are collected using four different matrices whose characteristics try to capture the cases where the timing method is inaccurate, or the model is inaccurate or both:

- **raefsky:** this is a large matrix (much larger than the data cache size) with a high number of elements per row. This means that both the timing method and the block size selection strategy presented in [12, 7] should be accurate. The error in the performance prediction is just 8% while the error in the performance measure is 1%.
- **shyy161:** this matrix is larger than the data cache size so the performance measure is accurate enough (error is 4%) while it has a low number of elements per row and thus we expect the performance prediction based on the dense matrix model to be wrong (error is 94%). Such a large error in the performance prediction can be explained taking a look at the curve in Figure 9: the basic selection strategy always predicts a performance that has the value of the asymptote of the rational curve whether the right value (in the leftmost part of the curve) is much lower.
- **mcfe:** this is a small matrix with a relatively high number of elements per row. This means that the timing method will be almost inaccurate (measured error is 34%) while the error in performance prediction is enough low (19%) to result in a successful optimal block size selection.
- **jpwh_991:** this is a small matrix with a low number of elements per row. The timing measure has an error of 69% and the performance prediction has an error of 118%.

Table 3 reports predicted versus measured performance for the same matrices with the AcCELS selection strategy. The last column of this table contains the error of the performance prediction which is considerably lower than the error that affects the selection strategy that is based solely on the dense matrix performance.

5.2 Comparison of the two selection strategies

Tables 5, 6 and 7 report the timing for the matrix-vector products for both AcCELS and SPARSITY software respectively on Itanium2, AMD K6 and Power3 architectures. For both packages we report the time with the block size chosen by the selection strategy (respectively the AcCELS and the SPARSITY ones) and the time with the best-case block size. When there is an "=" sign it means that the selection strategy hits the block size that

Performance prediction error				
matrix	Itanium2	MIPS	Power3	AMD Athl. 1200 MHz
raefsky3	1%	2%	3%	1%
shyy161	7%	4%	9%	3%
mcfe	3%	2%	3%	4%
jpwh_911	2%	2%	4%	2%

Table 3: Predicted versus actual performance with the AcCELS selection strategy. Itanium2 architecture.

SPMV time reduction.				
matrix	Itanium2	MIPS	Power3	AMD Athl. 1200 MHz
s3rmt3m1	2.77	1.56	1.86	2.46
gemat11	2.13	1.19	1.64	1.28
pwt	1.90	1.00	1.23	1.09
bcsstm27	2.68	1.48	1.68	2.39
crystk02	2.66	1.46	1.76	2.71
olafu	2.80	1.49	1.60	2.81
raefsky3	4.09	1.74	1.69	3.64
goodwin	1.86	1.00	1.04	1.35
bai	2.36	1.13	1.33	1.56
bcsstk35	2.97	1.62	1.70	2.80

Table 4: The reduction on time required to performe the matrix-vector product operation using BCSR with the AcCELS automatic block size selection wrt the reference CSR storage format.

gives the best case time. On the last column we show the speedup that can be obtainer over the SPARSITY software package using the AcCELS block size selection method. Roughly speaking the last column reports the ratio between the data in the fourth and second columns.

Note that the matrix-vector product operations have a different performance depending on whether the matrix is stored with aligned or unaligned blocks. Thus the best-case block size (and thus the best time) is often different between SPARSITY (column-aligned) and AcCELS (column-unaligned).

These tables show that our performance model (Equation (3)) gives both a better performance estimation at a given block size (see previous section), and a better block-size selection. Table 4 reports the time spent for a matrix-vector product with the block size that is selected by the selection strategy and the reference time (i.e. the time with the 1×1 block size) for the Itanium2 architecture. We can see that blocking gives a considerable speedup for this class of matrix.

We note that in the installation phase, AcCELS performs substantially more work than SPARSITY, because of our more accurate model. The runtime selection of the blocksize is slower by a factor or 2 or 3, though this is largely the result of our using unaligned blocks, a feature that can be deselected by the user.

matrix	Time AcCELS selection (sec)	Time AcCELS best-case (sec)	Time SPARSITY selection (sec)	Time SPARSITY best-case (sec)	Speedup
raefsky3	2.25e-3	=	2.29e-3	=	1.01
shyy161	2.32e-3	=	3.01e-3	2.65e-3	1.30
mcfe	1.11e-4	=	1.05e-4	=	0.94
jpwh_991	5.77e-5	=	6.59e-5	5.79e-5	1.14
bayer02	5.97e-4	5.82e-4	5.91e-4	5.38e-4	0.98
saylr4	1.52e-4	=	1.89e-4	1.83e-4	1.24
ex11	2.70e-3	=	2.75e-3	=	1.01
memplus	7.92e-4	=	8.70e-4	8.08e-4	1.10
wang3	1.13e-3	=	1.44e-3	1.32e-3	1.27

Table 5: Time spent for a matrix-vector product with the selected block size and with the best-case block size for AcCELS and SPARSITY. Itanium2 architecture

matrix	Time AcCELS selection (sec)	Time AcCELS best-case (sec)	Time SPARSITY selection (sec)	Time SPARSITY best-case (sec)	Speedup
crystk03	2.03e-2	=	2.39e-2	=	1.17
orani_678	1.78e-3	=	2.82e-3	1.97e-3	1.58
rdist	1.98e-3	=	2.10e-3	2.04e-3	1.06
goodwin	7.68e-3	7.66e-3	8.42e-3	=	1.09
coater2	6.12e-3	=	6.73e-3	=	1.10
lhr10	5.96e-3	5.73e-3	6.51e-3	5.76e-4	1.09
ex11	1.81e-2	=	2.26e-2	2.14e-2	1.24

Table 6: Time spent for a matrix-vector product with the selected block size and with the best-case block size for AcCELS and SPARSITY. AMD K6 architecture.

matrix	Time AcCELS selection (sec)	Time AcCELS best-case (sec)	Time SPARSITY selection (sec)	Time SPARSITY best-case (sec)	Speedup
bayer02	1.90e-3	=	2.06e-3	1.84e-3	1.08
orani_67	1.38e-3	1.29e-3	2.82e-3	1.97e-3	2.04
saylr4	6.01e-4	=	7.07e-4	5.88e-4	1.17
shyy161	8.65e-3	=	1.09e-2	8.92e-3	1.26
ex11	1.51e-2	=	1.52e-2	=	1.00
lhr10	4.95e-3	4.91e-3	5.38e-3	4.77e-3	1.08

Table 7: Time spent for a matrix-vector product with the selected block size and with the best-case block size for AcCELS and SPARSITY. Power3 architecture.

6 Conclusions

The sparse matrix-vector product is among the most performance critical elements of many applications. One approach to increasing its flop rate is to tile the sparse matrix with small dense blocks, since these can be handled more efficiently than the general compressed row storage format. This approach, already proposed in [12, 7], requires a static setup phase as in ATLAS [13], but in addition a runtime analysis and conversion of the sparse matrix. However, this latter phase can be amortized over the many iterations of an iterative method, and, in the case of nonlinear method or time-stepping method, over many iterative solves.

We gave a detailed analysis of the spatial and temporal locality of blocked algorithms, relating it to processor elements such as cache lines, memory bandwidth and write-back behaviour, and TLB effects. We presented a performance model for the blocked algorithms that is a great improvement in accuracy over earlier models. As a result, our software also is more accurate in picking the optimal blocksize: in nearly all cases the model predicts the actually optimal blocksize.

Numerical tests given attest to the accuracy of our model, and to the resulting higher performance.

References

[1] Patrick R. Amestoy, Iain S. Duff, Jean-Yves L'Excellent, and Jacko Koster. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM Journal on Matrix Analysis and Applications*, 23(1):15–41, 2001. also ENSEEIHT-IRIT Technical Report RT/APO/99/2.

[2] Richard Barrett, Michael Berry, Tony F. Chan, James Demmel, June M. Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine, and Henk A. van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. Philadalphia: Society for Industrial and Applied Mathematics. Also available as postscript file on http://www.netlib.org/templates/Templates.html, 1994.

[3] Tim A. Davis and Iain S. Duff. A combined unifrontal/multifrontal method for unsymmetric sparse matrices. *ACM Trans. Math. Software*, 25:1–19, 1999.

[4] Jack Dongarra and Victor Eijkhout. Self-adapting numerical software for next generation applications. *Int. J. High Perf. Comput. Appl.*, 17:125–131, 2003. also Lapack Working Note 157, ICL-UT-02-07.

[5] Salvatorre Filippone and Michele Colajanni. PSBLAS: a library for parallel linear algebra computations on sparse matrices. *ACM Trans. on Math Software*, 26:527–550, 2000.

[6] Eun-Jin Im and Katherine Yelick. Model-based memory hierarchy optimizations for sparse matrices. In *Workshop on Profile and Feedback-Directed Compilation, Paris, France.*, 1998.

[7] Eun-Jin Im, Katherine Yelick, and Richard Vuduc. Sparsity: Optimization framework for sparse matrix kernels. *Int. J. High Perform. Comput. Appl.*, 18(1):135–158, 2004.

[8] Xiaoye S. Li. *Sparse Gaussian Eliminiation on High Performance Computers*. PhD thesis, University of California at Berkeley, 1996.

[9] Ali Pinar and Michael T. Heath. Improving performance of sparse matrix-vector multiplication. In *Proceedings of SuperComputing 99*, 1999.

[10] Sivan Toledo. Improving memory-system performance of sparse matrix-vector multiplication. In *Proceedings of the 8th SIAM Conference on Parallel Processing for Scientific Computing*, 1997.

[11] R. Vuduc, J. Demmel, and K. Yelick. Oski: A library of automatically tuned sparse matrix kernels. In *(Proceedings of SciDAC 2005, Journal of Physics: Conference Series **16**:521–530*, 2005.

[12] Richard W. Vuduc. *Automatic Performance Tuning of Sparse Matrix Kernels*. PhD thesis, University of California Berkeley, 2003.

[13] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001.

[14] `http://math.nist.gov/MatrixMarket/`.

[15] `http://www.enseeiht.fr/lima/apo/MUMPS/`.

[16] `http://www.cise.ufl.edu/research/sparse/matrices/`.

[17] `http://www.cise.ufl.edu/research/sparse/umfpack/`.