Performance Optimization and Modeling of Blocked Sparse Kernels

Alfredo Buttari[*], Victor Eijkhout[†], Julien Langou[†], and Salvatore Filippone[*]

Report ICL-UT-04-05

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 peak.

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 significant results but poses a serious problem on portability because the software becomes tightly coupled with the underlying architecture.

The Self Adaptive Numerical Software efforts [4, 10] 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.

---

[*] Tor Vergata University, Rome, Italy

[†] University of Tennessee, Knoxville TN 37996, USA. This research was partly support by SciDAC: TeraScale Optimal PDE Simulations, DE-FC02-01ER25480

We remark that the state of kernel optimization in numerical linear algebra is furthest advanced in dense linear algebra. The ATLAS software [10] gives near optimal performance on the BLAS kernels. Factorizations of sparse matrices (MUMPS [1, 12], SuperLU [6], UMFPACK [3, 14]) 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 parameter optimized is the choice of the block dimensions, which is a function of the particular matrix and the machine.

An approach along these lines has already been studied in [9]. We employ essentially the same optimizations, but relax one restriction in that research. However, we have developed a more accurate performance model, which leads to better predictions of the block size, and consequently higher performance. We will compare the accuracy of the models and the resulting performance numbers.

Other authors have proposed similar and different techniques for accelerating the sparse matrix-vector product. For instance, Toledo ([8] 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 [7] also consider reordering the matrix; they use it explicitly to find larger blocks, which leads to a Traveling Salesman Problem.

While the reordering approach gives an undoubted 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 [8] and Vuduc [9] propose a solution where a matrix is stored as a sum of differently blocked matrices, for instance on 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); it will also be part 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 issue related to the sparse linear algebra. In Section 3, we present a storage format that is appropriate for block sparse operations, and provide the implementations for the matrix-vector product and the sparse triangular solve. We then give performance analysis and results for the matrix-vector product. Because of the very similar structure of the operations, this discussion carries over to the ILU solve.

2 Optimization of sparse matrix-vector operations

Matrix-vector multiplication and triangular system solving are very common yet expensive operations in sparse algebra computations. These two operations typically account for more than 50% of the total time spent in the solution of a linear sparse system using an iterative method and, 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 spatial locality:** sparse matrices are stored in data structures where in addition to the values of the entries the row indices and 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]. 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.
- **Low temporal locality:** In order to minimize memory access, it is important to maximize the number a data 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, 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. Unfortunately, the elements of the initial vector are accessed according to the column indices of the elements of the active row of A. The elements of x are reused during the matrix-vector product when their row indices belongs to two (or more) consecutive rows of the matrix A where there are elements on the corresponding column. Using the CSR storage format for the matrix implies that all the computations are performed row by row, thus, while moving from one row to the next the cache is in general overwritten. This leads to poor temporal locality of the source vector.
- **Low ratio between floating-point operations and memory operations:** 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. The comparison with the dense linear algebra is even starker if we consider that there is one write operation per row. Since there typically are far fewer elements per row in the sparse case, this type of overhead is relatively higher in the sparse case. Moreover retrieving and manipulating the column/row indices informations implies an amount of integer operations that is not negligible.
- **High loop overhead:** Connected to the low number of elements per row in sparse systems, the loop overhead is correspondingly higher. Furthermore, since the loop length is not constant throughout the matrix, there is more indexing computation involved, and because of the non-uniformity several compiler techniques such as loop interchange are not possible in straightforward manner.

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. This causes an improvement in scalar performance due to reduced indexing and greater data locality of the dense blocks. Unfortunately the number of operations increases due to the operations performed on the zeros arising in the dense blocks (this phenomenon will be referred to as *fill-in*). There is clearly a trade-off, which we propose to analyze.

Optimizing the sparse matrix-vector product kernels has two components:

1. Assessing the performance for blocks of different sizes. This performance is a non-trivial function of various architectural features;
2. Finding the best tiling for a given matrix. Each different block size results in a different number of stored nonzeros, and therefore a different number of operations performed. This needs to be balanced with the performance of the dense blocks, in a way that will be explained below.

Previously, the Atlas project [10] has been singularly successful in optimizing dense linear algebra kernels. The ATLAS strategy consists in optimizing the different parameters to the architecture in a installation phase.

In the sparse case, the structure of the matrix has a great influence on the optimal parameters and the resulting performance. A static approach like this one is not possible. Since the structure of the matrix is only known at runtime, the choice of the parameters for the sparse matrix-vector product is performed at runtime. Consequently the block size selection is on the basis of information that is gathered in two distinct phases:

1. **Installation-time phase:** in this phase we analyze the impact of the architecture characteristics on the performance of the block operations.
2. **Run-time phase:** in this phase we examine how the sparsity structure of the matrix influences the fill-in ratio.

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.

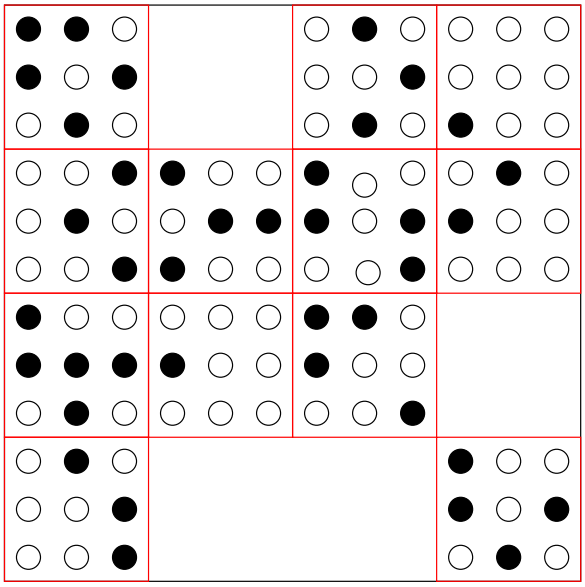

Figure 1: Fill-in for 3×3 row and column aligned blocks.

Figure 1 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 with 3×3 block size the fill-in ratio is 2.8. Performing the matrix-vector product with the matrix in Figure 1 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 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 2 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.

The data structure used to store a matrix in BCSR format is the same as the one used inside PSBLAS to store matrices in CSR, COO or JAD formats. All of these storage formats need two integer arrays to store information about the indices and a double precision real array to store the nonzero elements of the matrix. In the case of the BCSR format, we decided to store the blocks row size r and column size c inside the `INFOA(:)` array that is meant to contain informations relative to the structure of the matrix.

```
TYPE D_SPMAT
  ! Numbers of rows and columns
  INTEGER     :: M, K
```

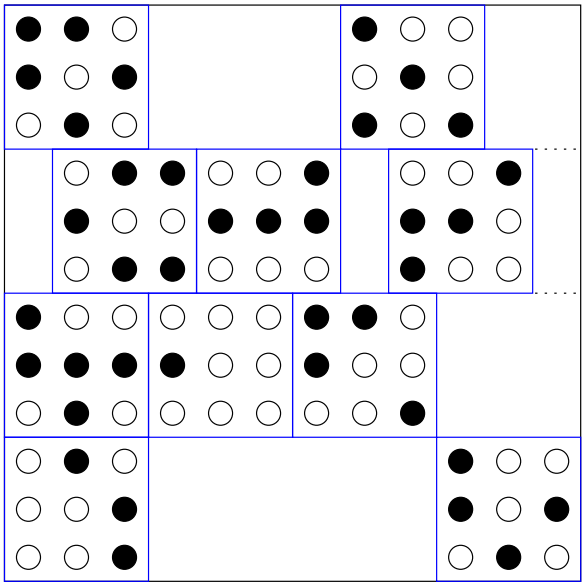

Figure 2: Fill-in for 3×3 row aligned blocks.

```
    ! Identify the representation method. Ex: CSR, JAD, ...
    CHARACTER(LEN=5) :: FIDA
    ! Describe some characteristics of the sparse matrix
    CHARACTER(LEN=11) :: DESCRA
    ! Contains some additional information on the sparse matrix
    INTEGER     :: INFOA(10)
    ! Contains the coefficients of the sparse matrix
    REAL(KIND(1.D0)), POINTER :: ASPK(:)
    ! Contains indices that describes sparse matrix structure
    INTEGER, POINTER :: IA1(:), IA2(:)
    ! Permutations matrix
    INTEGER, POINTER :: PL(:), PR(:)
  END TYPE D_SPMAT
```

The array `ASPK(:)` contains the elements of the matrix (nonzeros plus fill-in zeros), the array `IA1(:)` contains the columns indices of each block while the array `IA2(:)` contains the pointers for each block row inside `ASPK(:)` and `IA1(:)`.

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 the following:

```
    ...
    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;
}
...
```

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 an accumulator `yi` and the code relative to 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 squared 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:

```
...
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];
}
...
```

The code is very similar to the one for the 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 MFlops 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 method we use for estimating the fill-in caused by a block size is the one proposed in [9]: we sample a number of matrix rows and compute their individual fill-in. The fill-in of the whole matrix is estimated the fill-in of this sample. This method relies on the assumption that the matrix has a regular pattern.

This approach works fine with most matrices. However, the estimate can be inaccurate when applied to matrices that have an highly irregular structure. There is a trade-off in this strategy between the time we want to spent at run-time to obtain a good estimation of the fill-in and the inaccuracy of this fill-in. Following [9], a parameter acc ($0 \leq acc \leq 1$) is added to give the user a control on the size of the sub-matrix used in the fill-in estimation. Given $m = \lceil n/r \rceil$ the total number of block rows of the matrix for a given value of r (the block-row dimension), the fill-in is computed with the $m \cdot acc$ block rows of the matrix. The selection of the rows is made with a divide-and-conquer method based on random numbers generation: the set of m block rows is split in $m \cdot acc$ subsets and inside each of these subsets a block row is selected randomly. If A' is the submatrix composed by the block rows $m \cdot acc$ samples, the operation performed at this phase can be formalized as

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

In the case of an irregular matrix, where an accurate fill-in value is needed, the choice $acc = 1$ (the fill-in to be computed exactly) might be necessary. 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 PHiPAC) is $acc = 0.2$.

We already pointed out a difference between our package AcCELS and PHiPAC [9], in that AcCELS does not require blocks to be column aligned, which implies a lower fill-in. Regarding the time spent in the matrix-vector product, this is an advantage. However this feature implies that the estimation of the fill-in (preprocessing phase) becomes more complex and thus more expensive (up to three times more expensive). We add a parameter to control whether the user wants aligned blocks or unaligned blocks. The default in the AcCELS package is to have unaligned blocks.

4.2 Modeling block matrix performance

As is apparent from the code examples above, use of the BCSR storage format improves the performance rate 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 dimension becomes too big and register spilling happens. In this Section, we show that this expected behavior is almost never clearly followed on the different architectures tested (except in the case of the Itanium2 architecture). In practice, it is not possible to predict the performance of a certain block size emperically, so (as we will see in the experiments in section 4.2.1 and 4.2.2), we have to perform an exhaustive search through all the possible dimensions of the blocks 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 performance gain to be expected. In Figure 3 we plot the speedup obtained for the matrix-vector product, (over the reference implementation) for all the possible $r \times c$ block sizes on an Itanium2 machine. The matrix used is a 1500×1500 dense matrix stored in BCSR format. The highest speedup is obtained for the 8×1 block size, with a value of 3.55. The effect of the register spilling is visible on the 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. However, the performance is actually only increasing until the value $r = 8$, after which it drops to a lower level. On the other architectures tested we do not observe this behaviour. For instance, in Figure 4 the speedup is plotted for the same matrix on an SGI Octane. Here the numbers can not so easily be interpreted, even though we still have reasonable speedups (the highest is 1.97 for 5×7 blocks dimension).

However, these measurements on a dense matrix are likely to give only an upper bound on the possible speedup. In an actual sparse matrix some operations will be on stored fill-in elements, bringing down the speedup over the reference implementation, and the loop overhead will be larger too. Figures 5 and 6 describe the actual speedup and fill-in ratio respectively, using the 'venkat01' matrix on an Itanium2.

This matrix has a block structure and, even if the fill-in ratio is very high for bigger values of r and c (6.14 for 10×10 blocks dimension), for the 4×4 blocks dimension and the sub-multiples, the fill-in ratio is 1. The speedup over the time spent in a matrix-vector operation is the result of the effect of register reuse and the overhead due to the fill-in and is plotted in Figure 7 (matrix venkat01 on an Itanium2 machine).

Comparing Figures 5, 6 and 7 we can see that even if the highest MFlops rate is for block dimension 8×1 (4.09), the fastest matrix-vector product is for block dimension 4×2 because of lower fill-in.

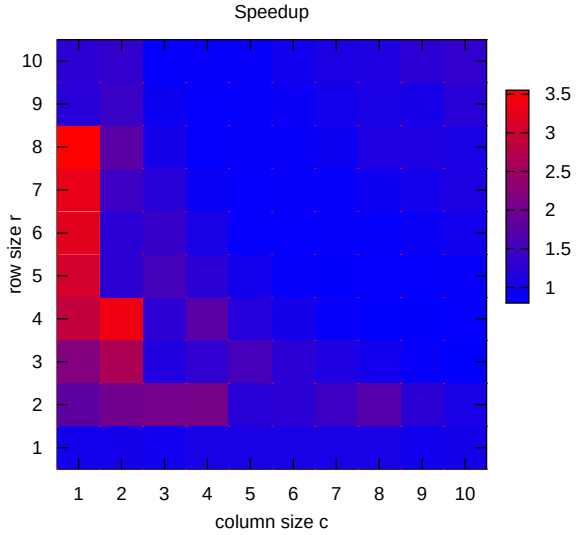

Figure 3: Speedup for the matrix-vector product of a 1500×1500 dense matrix stored in BCSR format on an Itanium2 machine.

4.2.1 Performance modeling by dense matrix

In this section we present the implementation of the performance prediction method that is used in [9]. The impact of the register blocking on the performance is estimated at installation-time by performing the matrix-vector product of a dense matrix stored in BCSR format for all the possible combinations of c and r. Thus, this model implicitly makes the assumption that

$$perf'_A(r,c) = perf_{Dense}(r,c).$$

This is justified if the sparsity structure of a matrix has a negligible impact on the effect that the register blocking has on the performance.

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

We add an optimization to the strategy in [9]. 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 lower 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 that 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. For example, in Figure 3: the values on the first column (those relatives to $c = 1$) often are the highest for their corresponding each row.

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

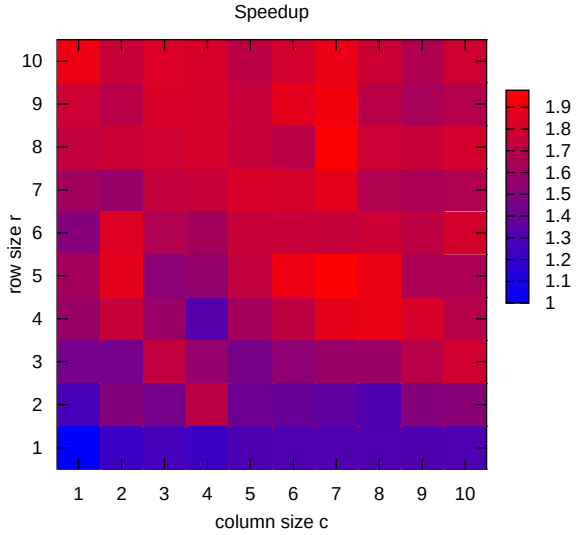

Figure 4: Speedup for the matrix-vector product of a 1500×1500 dense matrix stored in BCSR format on a SGI Octane machine.

1. Reading the file built at installation-time phase that contains the performance information $perf'_A(r, c)$ 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 $\frac{fill'_A(r,c)}{perf'_A(r,c)}$ has the minimum value.

4.2.2 Improved performance model

The main reason why the performance prediction method described above might be inaccurate is that often the performance rate 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 operation: 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):

```
...
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]=y[0];
}
...
```

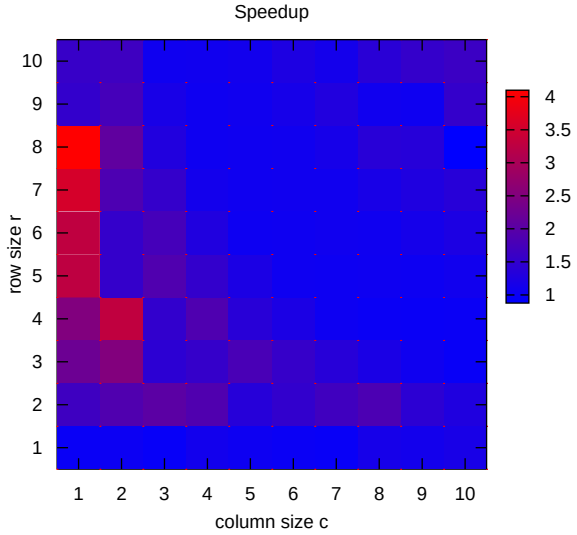

Figure 5: Speedup of blocked algorithm over the reference CRS method, for matrix venkat01 stored in BCSR format on an Itanium2 machine.

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 much more expensive than a floating-point operation, if we have a high ratio between the number of floating-point operations and write memory accesses, we have better performances. The number of write memory accesses depends on the size of the matrix thus the matrix-vector product has better performance for the matrices which have a higher 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 red line describes the performance for the matrix-vector product of sparse hand built matrices with all the elements close to the diagonal while the green one describes the flop rate for the matrix-vector product of a set of sparse matrices from real-world applications. (The set corresponds to the matrices that we present in Section 5). The difference between the two curves is due to the fact that for the first set of matrices the number of elements per row is constant among all the rows while for the second set of matrices the number of elements per row may be different from one row to another.

- **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 to have elements of the source

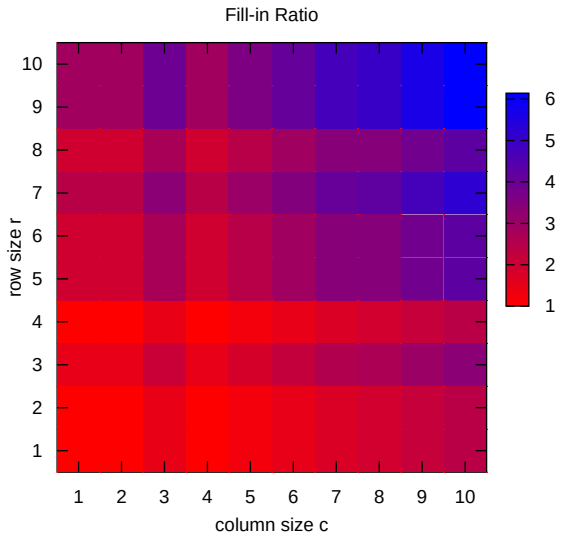

Figure 6: Fill-in ratio for matrix venkat01 stored in BCSR format.

vector that are brought inside one cache line. If elements in consecutive rows are close to each other it means that there is an higher probability to have elements on the same column and thus an higher reuse of the elements of the source vector when making the computations relative to those rows.
The curves in Figure 9 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 an interval around the diagonal. The bandwidth is defined as: $\max_{j=1,nrows}(max_j - min_j)$ where $nrows$ is the number of rows of the matrix and max_j and min_j are the maximum and minimum column indices over each row. Thus the bandwidth of a matrix is defined as the width of the interval. The column indices are randomly generated.
In Figure 9 we observe that the matrices with a lower bandwidth (red curve) have higher performance than those with a large bandwidth.

Even if it is possible to exhibit the role played by the distribution of the elements in a row (Figure 9), Figure 8 indicates that the performance of matrix-vector products for matrices coming from real-world applications is very close to the performance of banded matrices. Thus taking into account how the elements are distributed in a row, is only of minor practical interest. For this reason, we only design a model that takes into account the number of nonzero per row of the matrix.

This model (as opposed to the first one) takes the sparsity characteristic of the matrix into account to have a better evaluation of the performance. Thus it is aimed to have a better prediction 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 [11] (resp. the University of Florida matrix collection [13]) have

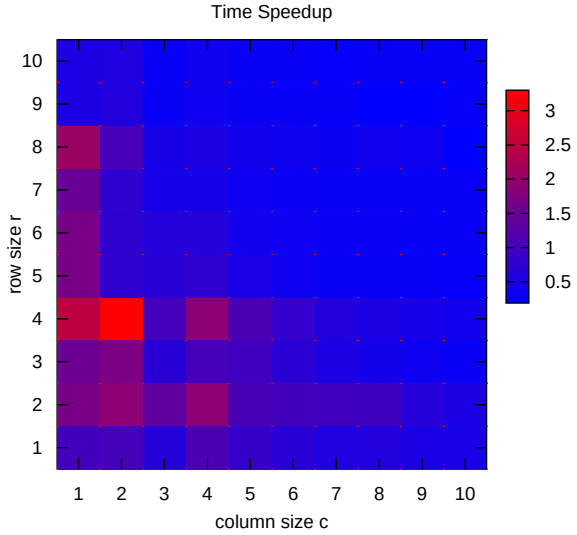

Figure 7: Time reduction for the matrix-vector product of matrix “venkat01” on an Itanium2 machine.

less than 7 (resp. 8) elements per row (as of this writing). Figure 8 shows that using a dense matrix to model the performance rate of the sparse matrix leads to a misprediction of factor 3 for more than half of the sparse matrix available in those two standards collection.

We expect that this 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 case.

A simple implementation of this strategy consists in computing the curve in Figure 8 for each block size, and store 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 such approach is prone to spurious timings resulting in unreliable values of MFlops 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 relative to each (block) row can be modeled as $c_1 + c_2 elem\_row$ where $elem\_row$ is the number of (block) nonzeros, the number of operations is for itself proportional to $elem\_row$. Finally the corresponding flop rate (number of operations divided by time), $\mathrm{perf}\text{”}_A(r,c)$, is expected to follow a hyperbola that we prefer to write with three parameters:

$$\underset{A}{\mathrm{perf}}\text{”}(r,c) = a + \frac{b}{elem\_row + c}. \tag{3}$$

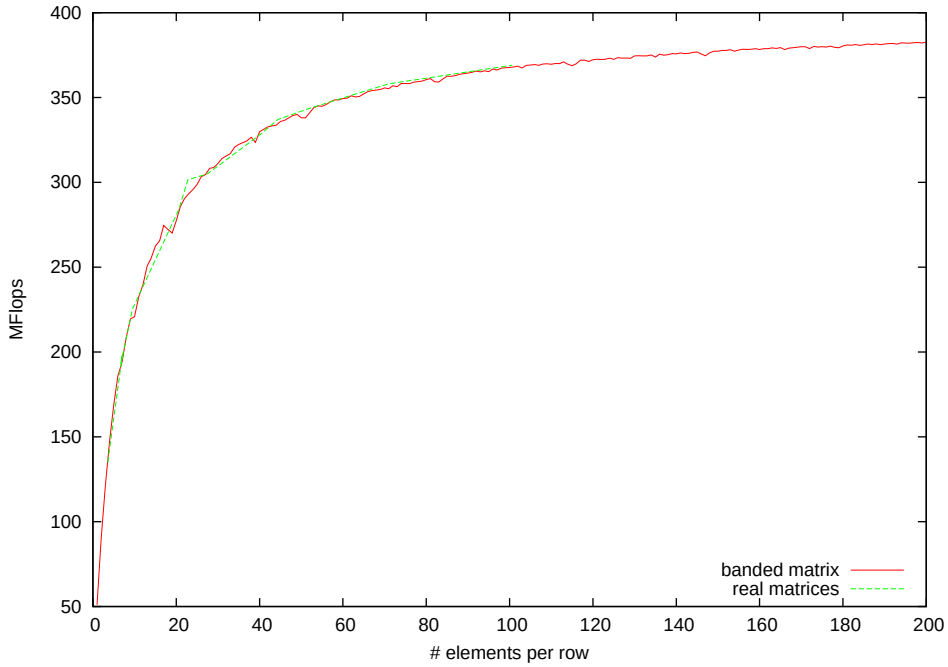

Figure 8: Performance rate for matrices with different number of elements per row. The red line plots the performance of banded sparse matrices while the green one plots the performance rate for a set of sparse matrices from real-world applications.

(We add a third parameter for flexibility purpose.) a is equal to $\mathrm{perf}'_A(r,c)$, the performance rate for the dense matrix. $b/(elem\_row + c)$ is the correction we proposed to add in order to have a more accurate model. b is negative, c is positive, so that the negative correction term gets larger for smaller $elem\_row$.

Figure 10 reports the curve that is measured for the 1×1 block size case (the red one) and the curve that is built for the same block size case with the regression model (3): the distance between the two is negligible for our purposes. In a general manner, on the Itanium2 machine, we observe that the curves plotted for the possible block sizes are all within a few percent of the model (3).

We have observed that $ac = -b$, to within a few percents. This constraints sets $\mathrm{perf}''_{nnz=0}(r,c) = 0$. (As we can see on Figure 10, the curve goes through the origin.) This constraint fits the two-parameter model (c_1, c_2) explained previously ($\mathrm{perf}''_A(r,c) \propto elem\_row/(c_2 elem\_row + c_1)$); however we prefer to work with a three-parameter model (a,b,c) as exposed in Equation 3. It is not much more expensive and might fit experimental data better.

4.3 Performance modeling and optimization procedure

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

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 (a, b and c) are then computed using a least-square fitting method and finally the parameters a, b and c are tabulated for all the block sizes.

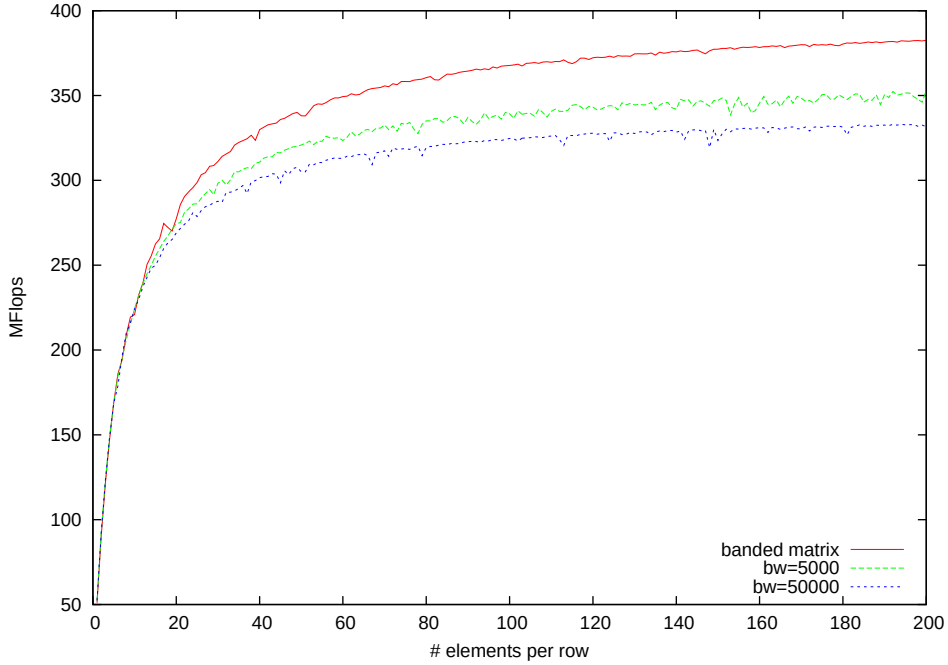

Figure 9: Flop rate versus number of elements per row. The red line plots the performance on banded matrices (i.e. the bandwidth is as low as possible), the green one plots the performance on sparse matrices with bandwidth=5000 while the blue one plots the performance rate sparse matrices with bandwidth=50000.

The matrices used during this process are automatically generated banded matrices, and the least square 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 $b \leq 0 \leq c$; if the data is very messy, the regression might violate this condition (this has happened on some architectures for some block sizes). In such a case, we set $b = c = 0$ and a equal to the mean value of the computed performance rates. This reduces our strategy to the one used in [9] 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 a, b, and c of the rational function 3 are read from the file built at installation-time. phase.
- the number of elements per row is computed as:

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

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

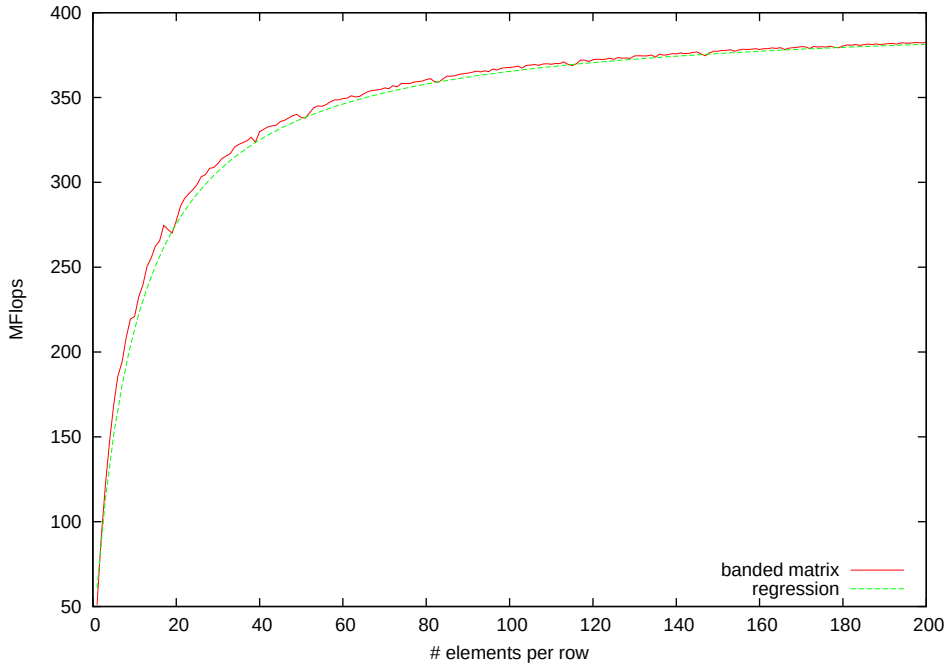

Figure 10: Comparison between the measured performance vs number of elements per row curve (red) and the one built with the regression method (green).

- the performance estimate is computed as:

$$perf'_A(r,c) = a + \frac{b}{elem\_row + c} \tag{5}$$

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

5 Numerical tests

In this section we report the results of our block-size selection strategy compared with results obtained using the PHiPAC software described in [9].

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 can not 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 with a high probability. 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 time 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 behaviour, 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 green line reports the case where cache effect is not avoided (i.e. data set that includes only one matrix) while the red one 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 is

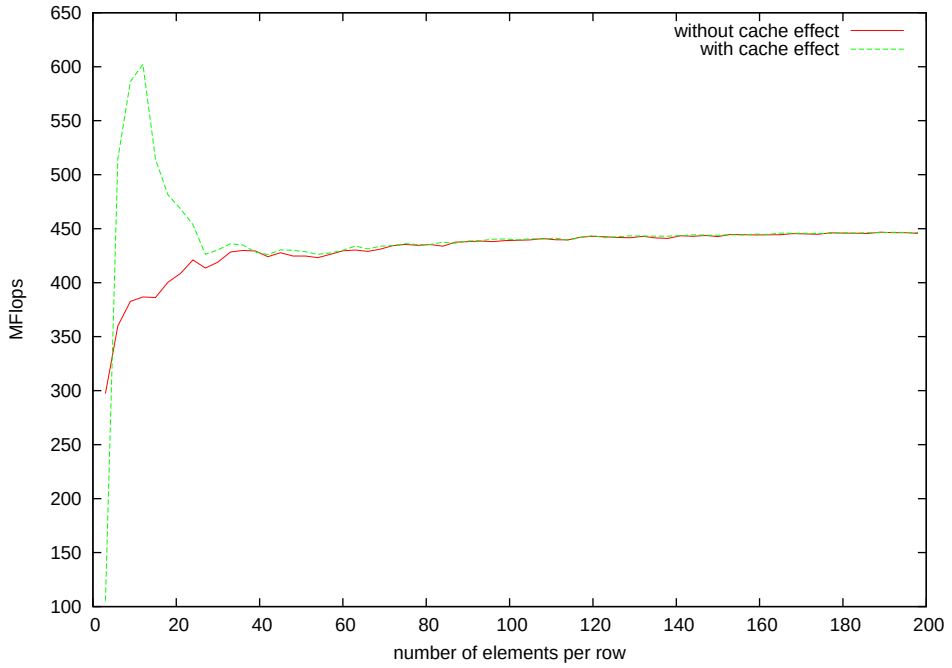

Figure 11: Comparison between timings with (green) and without (red) cache effects. This data is computed using a banded sparse matrix on an Itanium2 machine.

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

In [9], the matrices studied are large enough to automatically flush the cache. For this reason, there is no cache flush. Given that some matrices in our test set have smaller dimension, it is necessary to adopt our timing method to obtain reliable measures. Thus even when comparing with performance obtained with PHiPAC we will refer to the timings measured with the proposed technique.

matrix	Predicted perf. (MFlops)	measured perf. (MFlops)	actual perf. MFlops	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 1: Predicted versus measured versus actual performance with code in [9]

5.2 Results

5.2.1 Timing Routine and Quality of the Models

Table 1 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 [9]; 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 use 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 [9] 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 red curve in Figure 10: 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 2 reports predicted versus measured performance for the same matrices with our improved selection strategy. The last column of this table contains the error of the performance prediction the is considerably lower than the error that affects the selection strategy that is based on the dense matrix performance. .

5.2.2 Comparison of the two selection strategies

Tables 3, 4 and 5 report the timing for the matrix-vector products for both AcCELS and PHiPAC software respectively on Itanium2, AMD K6 and Power3 architectures. For both the packages we report the time with the block size that is selected by the selection strategy

matrix	predicted perf. (MFlops)	actual perf. (MFlops)	matrix size (MegaBytes)	elem. per row	Pred. error
raefsky3	1196	1275	11.35	70.2	6%
shyy161	725	815	2.51	4.3	11%
mcfe	954	964	0.186	31.9	1%
jpwh_911	402	411	0.045	6.1	2%

Table 2: Predicted versus actual performance with the improved selection strategy.

matrix	Time AcCELS selection (sec)	Time AcCELS best-case (sec)	Time PHiPAC selection (sec)	Time PHiPAC best-case (sec)
raefsky3	2.33e-3	=	2.29e-3	=
shyy161	2.47e-3	=	3.01e-3	2.65e-3
mcfe	1.05e-4	=	1.05e-4	=
jpwh_991	5.77e-5	=	6.59e-5	5.79e-5
bayer02	5.43e-4	5.40e-4	5.91e-4	5.38e-4
saylr4	1.58e-4	=	1.89e-4	1.83e-4
ex11	2.61e-3	=	2.75e-3	=
memplus	8.03e-4	=	8.70e-4	8.08e-4
wang3	1.19e-3	=	1.44e-3	1.32e-3

Table 3: Time spent for a matrix-vector product with the selected block size and with the best-case block size for AcCELS and PHiPAC.

(respectively the improved and the one based on the dense matrix performance) 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 gives the lower time.

Note that the matrix-vector product operations have different performance 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 PHiPAC (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.

To finish comparing the two strategies, we shall say that in what concern the time spent at installation-time, the basic selection strategy is much faster than the improved one. Regarding the preprocessing phase (made at running time), it is about the same for both strategies.

5.2.3 Speedup obtained over the standard matrix-vector product

Table 6 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 enables nice speedup for this class of matrix on the Itanium2.

matrix	Time AcCELS selection (sec)	Time AcCELS best-case (sec)	Time PHiPAC selection (sec)	Time PHiPAC best-case (sec)
crystk03	2.16e-2	=	2.39e-2	=
orani_678	1.78e-3	=	2.82e-3	1.97e-3
rdist	1.84e-3	=	2.10e-3	2.04e-3
goodwin	7.62e-3	=	8.42e-3	=
coater2	6.33e-3	6.19e-3	6.73e-3	=
lhr10	5.56e-3	=	6.51e-3	5.76e-4
ex11	1.73e-2	=	2.26e-2	2.14e-2

Table 4: Time spent for a matrix-vector product with the selected block size and with the best-case block size for AcCELS and PHiPAC. AMD K6 architecture.

matrix	Time AcCELS selection (sec)	Time AcCELS best-case (sec)	Time PHiPAC selection (sec)	Time PHiPAC best-case (sec)
bayer02	1.90e-3	=	2.06e-3	1.84e-3
orani_67	1.78e-3	=	2.82e-3	1.97e-3
saylr4	5.19e-4	=	7.07e-4	5.88e-4
shyy161	8.65e-3	=	1.09e-2	8.92e-3
ex11	1.39e-2	=	1.52e-2	=
lhr10	4.35e-3	=	5.38e-3	4.77e-3
onetone2	6.68e-3	=	7.09e-e	6.70e-3

Table 5: Time spent for a matrix-vector product with the selected block size and with the best-case block size for AcCELS and PHiPAC. Power3 architecture.

matrix	Time AcCELS selection (sec)	Time AcCELS reference (sec)	Speedup
raefsky3	2.33e-3	8.39e-3	3.63
shyy161	2.47e-3	4.23e-3	1.71
mcfe	1.05e-4	1.57e-4	1.49
venkat01	3.38e-3	1.12e-2	3.31
bayer02	5.43e-4	8.21e-4	1.51
saylr4	1.58e-4	2.38e-4	1.50
ex11	2.61e-3	6.22e-3	2.38
memplus	8.03e-4	1.29e-3	1.60
crystk03	4.09e-3	9.85e-3	2.40

Table 6: Time spent for a matrix-vector product with the selected block size and with the best-case block size for AcCELS and PHiPAC. Itanium2

6 Conclusions

There are several issues in using a blocking strategy in sparse matrix-vector. First of all, there is a strong matrix/architecture dependence. According to equation (1), to have an improvement over the 1×1 product (the reference implementation), we definitely need to have a fill-in (matrix dependent) smaller than the increase in the performance rate (machine dependent). It is also application dependent. There is a non-negligible preprocessing overhead at run-time due to (a) the selection of the block-size (performance and fill-in prediction), and (b) the change of data-format of the matrix (the matrix comes with its original format and needs to be changed in BCSR format with the chosen block size). These factors are outside the scope of this paper; for a detailed discussion we refer the reader to [9].

In this paper we have presented implementations of the sparse matrix-vector product and LU solve kernels using a block matrix format that is more general than earlier proposed formats (unaligned blocks). A performance model is given that accurately predicts the performance of the sparse kernels. Compared to previous strategy, prediction and selection are improved. Finally numerical tests bear out the accuracy of our model, and the improvements attainable.

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

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

[8] 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.

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

[10] 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.

[11] http://math.nist.gov/MatrixMarket/.
[12] http://www.enseeiht.fr/lima/apo/MUMPS/.
[13] http://www.cise.ufl.edu/research/sparse/matrices/.
[14] http://www.cise.ufl.edu/research/sparse/umfpack/.