On Improving Data Locality in Sparse Matrix Computations

Peter M.W. Knijnenburg Harry A.G. Wijshoff

High Performance Computing Division, Dept. of Computer Science,
Leiden University,
P.O. Box 9512, 2300 RA Leiden, the Netherlands

peterk@cs.leidenuniv.nl and harryw@cs.leidenuniv.nl

Abstract

Sparse matrix computations and irregular type computations show poor data locality behavior. Recently compiler optimizations techniques have been proposed to improve the data locality for regular type loop structures. Sparse matrix computations do not fall into this categorie of computations and the issue of compiler optimizations for sparse computations is merely understood. In this paper we describe how compiler optimizations based on pattern matching techniques can be used to improve the data locality behavior for sparse computations.

1 Introduction

The effective use of caches is becoming more and more crucial for obtaining efficient implementations of large scale computations on parallel computing platforms. Fortunately, many applications show good cache behavior on sequential architectures [HP90]. However, in the context of parallel computing cache coherency can deteriorate this behavior. This is due to the fact that data residing in caches is invalidated by write actions of other processors. Also the distribution of the workload over multiple processor might decrease the possible data reuse per processor. All this requires the role of caches for large scale application to be re-examined.

A substantial part of large scale applications is governed by sparse matrix computations. These sparse matrix computations exhibit next to poor data locality [SW90] also chal-

lenges for compiler optimizations in general. This is caused by the fact that most of the sparse codes are obscured due to the presence of indirect addressing and the dependence on specific problem instances. Recently runtime solutions, i.e. the CHAOS/PARTI library, have been proposed to facilitate parallelization of sparse matrix computations and in particular mesh type applications [vHKK+92, SBW91]. This runtime approach seems to be very promising to handle the parallelization and data distribution problem of sparse computations. Another approach is formed by requiring that sparse computations are defined on the dense envelop of the sparse data structures followed by an automatic conversion to optimized sparse codes [BW93a, BW93b]. In both approaches the exploitation of data locality has not been addressed yet.

For regular type of loop nests work based on the window concept [BEJW92, EJWB93, GJG87] can be effectively used to obtain automatic compiler optimizations increasing the exploitation of data locality. This technique is very promising whenever an accurate estimate of the window size can be obtained. Due to the fact that sparse computations depend on specific problem instances this approach might breakdown for these types of computations, though.

In this paper we describe how compiler optimizations together with at runtime evaluation of specific library routines can be used to improve the data locality behavior of sparse computations. These techniques rely on pattern matching as a basis to efficiently characterize the sparsity structure of the sparse systems. In order to facilitate the description of the techniques proposed we had to adopt certain constraints on the presentation which are given in the next section. Then some simple methods for improving data locality are described. In section 4 we discuss the need for pattern matching in order to characterize possible data locality improvement. This is followed by a description how compiler support can be used together with runtime evaluation to realize better cache behavior. In section 6 specific techniques are described which rely on pattern matching and finally conclusions are given.

2 Framework

In this paper we assume that a sparse matrix A is stored rowwise. This means that there exists two arrays LO and HI of length N, containing pointers in an array IND in which the indices of entries are stored. The length of this array is the number of non-zero entries in the matrix, NZ. The entries for row i are to be found in IND(LO(i)) through IND(HI(i)). For each i, the sequence of indices in row i is denoted by α_i. The entries of A are stored in an array A of length NZ. In this paper we assume that each sequence α_i is ordered. If the α_i's are not ordered, we can order them locally. Since, on average, every sequence α is small, this can be done efficiently.

The program fragment we use to illustrate the techniques is the simple code for *matrix-vector multiplication* $y = Ax$, given below. The code consists of a double loop, the outermost of which ranges over rows, and the innermost over the entries in a row.

```
DO i = 1,N
  DO j = LO(i),HI(i)
    y(i) = y(i) + A(j) * x( IND(j) )
  ENDDO
ENDDO
```

If a compiler wants to use the techniques described in this paper, it has to know that the collection of arrays LO, HI, IND and A together constitute a representation of the sparse matrix A. Hence we assume that the compiler incorporates a directive to communicate this fact to it.

Next, the transformations we consider consist of traversing the iteration space in a different order than in the original code, for both loops. Hence we assume that the loops to which the transformations have to be employed do not contain loop carried dependences, except possibly a dependence (like the output dependence on y in the fragment above) that can be ignored or broken by other techniques.

In the fragment for matrix-vector multiplication, the entire matrix A is accessed. Hence the fragment serves as a good test case for studying the potential of the proposed techniques. That is, if a technique does not perform well on this code, then it is unlikely that it will perform good on other codes. The transformations we propose can be extended to other

codes, like sparse triangular solve, in a straightforward way.

3 Some simple techniques

Observe that the code for matrix-vector multiplication already exhibits strong data locality on A and y. If an element $A(n)$ is used in one iteration of the innermost loop, then the element $A(n+1)$ is used in the next (spatial locality). Also, during the execution of the inner loop, the element $y(i)$ is referenced `HI(`i`)` $-$ `LO(`i`)` $+1$ times. The element $y(i+1)$ is referenced in the next iteration of the outer loop (temporal locality). The only place where the code does not exhibit locality is on references to the vector x. In this section we give some techniques for improving the locality on x. We compare the hit ratios for the original loop, and the transformed loops. The techniques we discuss are two obvious strategies for improving the locality on x. First we *strip* the matrix into sets of consecutive columns; then we *block* the matrix. These techniques for improving data locality have been studied intensively for the dense case [GJM86].

First we give an approximation of the hit ratio of matrix-vector multiplication. We assume that the dimension of A is N, and that it is randomly filled with γ entries on average in each row (or column). Hence A contains γN entries in total. The length of a cache line is c. We furthermore assume that every reference to x generates a cache miss. In order to facilitate the analysis of the hit ratio we have chosen in this paper to ignore misses caused by cache conflicts. For analytic models taking into account also these kind of conflict the reader is referred to [TFJ93]. Under these assumptions we obtain for matrix-vector multiplication

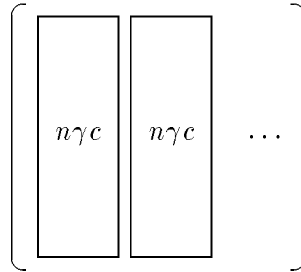
- N/c misses and $2\gamma N - N/c$ hits on y;
- $\gamma N/c$ misses and $\gamma N - \gamma N/c$ hits on A.

From this it easily follows that the hit ratio HR_0 is given by

$$HR_0 = \frac{3}{4} - \frac{1}{4}\frac{1}{\gamma c} - \frac{1}{4c} \qquad (1)$$

3.1 Stripping columns

In this section we consider the possibility to access nc columns of A at the same time, where c is the length of a cache line and n is some constant. We divide A up in $\lceil N/nc \rceil$ collections of consecutive columns, or *strips*. Each strip contains $n\gamma c$ entries.



We now present the code fragment into which the matrix-vector multiplication can be transformed. We use an auxiliary array `MARK` to keep track of the first position in `IND` where the current collection of columns may start. We initialize `MARK`(i) to `LO`(i), for each i. The code for matrix-vector multiplication can be transformed into the following fragment. In this fragment, NS denotes the number of strips $NS = N/nc$.

```
   DO k = 1,NS
     DO i = 1,N
       DO j = MARK(i),HI(i)
         IF ( IND(j) > k*c ) THEN
           MARK(i) = j
           GOTO 1
         ELSE
           y(i) = y(i) + A(j) * x( IND(j) )
         ENDIF
       ENDDO
       MARK(i) = HI(i)+1
1      CONTINUE
     ENDDO
   ENDDO
```

We can compute the hit ratio for the transformed loop as follows. For each strip, we have

- n misses and $n\gamma c - n$ hits on x;
- N/c misses and $2n\gamma c - N/c$ hits on y;

- $n\gamma$ misses and $n\gamma c - n\gamma$ hits on A.

From this it follows that the hit ratio HR_1 is given by

$$HR_1 = 1 - \frac{1}{4}\frac{1}{\gamma c}(1+\gamma) - \frac{1}{4c}\frac{N}{n\gamma c} \tag{2}$$

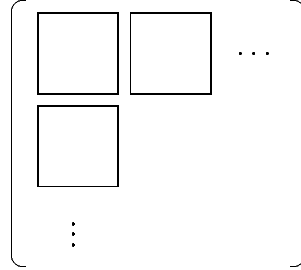
For comparing the expression for HR_0 in (1) with the above expression for HR_1, we compute $HR_1 - HR_0$.

$$HR_1 - HR_0 = \frac{1}{4} - \frac{1}{4}\frac{1}{\gamma c}\left(\frac{3}{4} + \gamma\right) - \frac{1}{4c}\left(\frac{N}{nc\gamma} - 1\right)$$

Instantiating this expression with $\gamma = 10$ and $c = 8$, we see that the hit ratio improves ($HR_1 > HR_0$) if n is roughly larger than $N/300$.

3.2 Blocking

In this section we discuss the idea of accessing A in a block-wise fashion. We consider blocks of size $nc \times nc$, where c is the length of a cache line and n is some constant.



The code into which we may transform the matrix-vector multiplication fragment is a straightforward extension of the code in the previous subsection, and is therefore omitted. We present an analysis of the hit ratio in this case. Under the same assumptions as in the previous subsection, we have that each block contains $\left(\frac{nc}{N}\right)^2 N\gamma$ entries. We have $(N/nc)^2$ blocks. Then for each block there are

- n misses and $\left(\frac{nc}{N}\right)^2 N\gamma - n$ hits on x;
- n misses and $2\left(\frac{nc}{N}\right)^2 N\gamma - n$ hits on y;
- $\left(\frac{nc}{N}\right)^2 \frac{1}{c} N\gamma$ misses and $\left(\frac{nc}{N}\right)^2 N\gamma \left(1 - \frac{1}{c}\right)$ hits on A.

The hit ration HR_2 now follows

$$HR_2 = 1 - \frac{1}{2}\frac{1}{\gamma c}\frac{N}{nc} - \frac{1}{4c} \tag{3}$$

Computing $HR_2 - HR_0$ we obtain

$$HR_2 - HR_0 = \frac{1}{4} - \frac{1}{4}\frac{1}{\gamma c}\left(2\frac{N}{nc} - 1\right)$$

With $\gamma = 10$ and $c = 8$, this expression is larger than zero, if n is roughly larger than $N/100$.

4 Pattern matching

In this section we propose a different technique for improving data locality based on *pattern matching*. This technique is complementary to the techniques from the previous section. That is, both techniques can be used at the same time resulting in better locality exploitation than before. We want to analyze the structure of a sparse matrix, and extract information that allows us to improve data locality. Our approach can be summarized as follows. Sparse matrices exhibit in many cases "dense" regions, blocks or lines. It seems reasonable to try and exploit the structure of these regions. Since these regions are dense and usually not too large, we expect to achieve high temporal and spatial locality. This observation suggests that it may be profitable to isolate these regions and process them separately. Hence we search for *patterns* in the matrix, having certain favorable properties like high density. Depending on the code at hand, there may be other properties that are interesting as well. The crucial observation to make now is that these patterns are mirrored in the compact representation of the matrix, in particular, in the array `IND` used to index the rows of A.

Consider the following example. Suppose our matrix A has a (large) number of submatrices of the form

$$\begin{matrix} * & * & * & * \\ * & & * & \\ * & * & * & \\ * & & & * \end{matrix}$$

Assume that the left upper corner is located at position $\langle i, j \rangle$ and that the pattern is permuted to the front of each row. Then we may unroll the loop accessing this pattern as follows. Note that we have $j =$ `IND`(i).

```
ii = LO(i)
y(i) = y(i) + A(ii)*x(j) + A(ii+1)*x(j+1) + A(ii+2)*x(j+2) + A(ii+3)*x(j+3)
ii = LO(i+1)
y(i+1) = y(i+1) + A(ii)*x(j) + A(ii+1)*x(j+2)
ii = LO(i+2)
y(i+2) = y(i+2) + A(ii)*x(j) + A(ii+1)*x(j+1) + A(ii+2)*x(j+2)
ii = LO(i+3)
y(i+3) = y(i+3) + A(ii)*x(j) + A(ii+11)*x(j+3)
```

The execution of this fragment is more efficient than the executing the corresponding statements in the original loop. First, we have removed (part of) the loop, and thus need not update loop counters and execute branches. More importantly, we achieve high locality by processing this pattern in isolation. Assuming that the architecture on which we want to run the program has a cache line size of 4, we obtain one cache miss for the references to x, one cache miss for the references to y, and four cache misses for the references to A. The total number of references is $4 \times 11 = 44$. The total number of hits is $44 - 6 = 38$. Hence the hit ratio is 19/22 which is optimal for this case. Moreover, if we extract the pattern from a large matrix, and process the rest of the matrix as it stands, we expect not to destroy the locality that is implicitly present in A too much. Furthermore, we may use other techniques, like stripping or blocking as described in the previous section, to improve locality in the remainder of A. This is completely independent of the processing of extracted regions.

The point is that we may recognize a pattern as above using the indirection array `IND`. The pattern is present at position $\langle i, j \rangle$ if α_i contains the indices $j, j+1, j+2, j+3$; α_{i+1} contains $j, j+2$, etc. We can sweep once through A and collect all pairs $\langle i, j \rangle$ such that this pattern is present at this position.

An easy way of keeping track of the entries in A that have been handled as part of a pattern is the following. We permute the pattern to the front of the row (or immediately to the right of patterns already permuted to the front). Likewise for the actual entries of the matrix in `A`. We then know where to find the pattern, and after consumption of the pattern we may set `LO(`i`)` to `LO(`i`)` plus the length of the pattern in row i. Thus we maintain the invariant that j values between `LO(`i`)` and `HI(`i`)` still have to be accessed.

Recall that we assume that for each i, the sequence α_i, which is the sequence of indices

`IND( LO(`i`) )` through `IND( HI(`i`) )`, is ordered. Given two ordered sequences α and β, we can decide whether they are equal, whether they contain an identical subsequence, or what their intersection is, in time $\mathcal{O}(|\alpha| + |\beta|)$. In most cases, the number of entries in a row in a sparse matrix is a small constant on average. Hence, on average, we can compare two patterns in constant time.

5 Compiler support

As we mentioned in a previous section, the compiler should be communicated the data structures that represent a sparse matrix. Since it is very expensive to go and search for arbitrary patterns, the compiler should be told the kind of pattern that the programmer expects to encounter in the sparse matrices on which the program is intended to run. Although it is possible to define a language for specifying patterns, together with special code for such a pattern, and let the compiler generate code for recognizing these patterns, we do not pursue this issue further here. At present we assume that the compiler has access to library calls which can recognize blocks and diagonal lines.

A library call to a function for recognizing a certain kind of pattern delivers a list of descriptions of every such pattern present in the matrix. A description will consists of a starting position and a specification of its size. For instance, a block may be described by the position of its left upper corner $\langle i, j \rangle$, its length k, and its depth ℓ.

Since the function that recognizes a pattern will permute the pattern to the front or the back of the row, it is important that the patterns will be accessed in the same order as in which they have been detected. So the list of patterns will be maintained as a queue.

For an easy example, suppose we have a sparse matrix A represented by the arrays `LO`, `HI`, `IND` and `A`, and suppose that the patterns present in A are blocks. Then we have the original code

```
C$SPARSE LO,HI,IND,A
C$PATTERN BLOCK

   DO i = 1,N
     DO j = LO(i),HI(i)
```

```
      ......
    ENDDO
  ENDDO
```

The compiler transforms this into

```
  CALL find_blocks( LO,HI,IND,A,BL )

  WHILE BL not empty DO
    CALL get_block( BL,i,j,k,l )
    ..... ! consume block using the same code as the original loop
          ! but also adapt LO(i) or HI(i) to consume a pattern
  ENDDO

C blocks have been processed, LO and HI have been adapted

  DO i = 1,N
    DO j = LO(i),HI(i)
      .....
    ENDDO
  ENDDO
```

At runtime, this code will be executed and the structure of the sparse matrix given by a specific problem instance will be characterized.

6 Specific techniques

In this section we discuss some examples of pattern matching in a sparse matrix and the transformations that can be applied.

6.1 A fixed pattern

Given some pattern α, we can extract all rows i such that $\alpha_i = \alpha$. We denote this collection by `ALPHA`. In this case, the transformed loop takes on a very special appearance. First we execute the rows that are equal to α, and then the rows that are not equal to α. Code for doing this is straightforward an therefore omitted. Another possibility we get in this case, is that we may permute the loops, as follows. Let n be the length of α.

```
DO j = 1,n
  DO i in ALPHA
    k = LO(i) + j - 1
    y(i) = y(i) + A( k ) * x( IND(k) )
  ENDDO
ENDDO
```

An analogous transformation may be employed if we collect rows which contain identical subsequences. In order to keep track of the position of the subsequence we permute this subsequence to the front of the row. The array A containing the entries of the matrix is adapted analogously.

6.2 Blocks

We now describe how one can exploit and detect some patterns automatically. The first pattern we consider is a block of entries in A. In this case, α_i contains $n, n+1, \ldots, n+m$; α_{i+1} contains $n, n+1, \ldots, n+m$, etc. A block can be described by giving the coordinates of the left upper corner $\langle i, j\rangle$, its length k and its depth ℓ. Note that, if we have permuted the block to the front of the row, the j coordinate is not necessary. Given such a block, we may execute the following code fragment for it.

```
DO ii = i,i+l
  DO jj = 1,k
    y(i) = y(i) + A( LO(ii)+jj ) * x( IND( LO(ii)+jj ) )
  ENDDO
  LO(ii) = LO(ii) + k  ! front is done
ENDDO
```

We now discuss how to recognize blocks in a sparse matrix. The algorithm we propose has $\mathcal{O}(NZ)$ running time, where NZ is the number of non-zeroes in the matrix. We sweep through the matrix row by row. We use a queue for maintaining a worklist containing triples $\langle j, k, \ell\rangle$, where j is the starting coordinate of the block in the previous row, k is its length, and ℓ is the depth seen so far.

1. Initialize the queue to empty.

2. For each row i do

- If i contains a consecutive sequence $n, \ldots, n+m$, and the worklist has an entry $\langle n, m, \ell \rangle$, remove this entry and enqueue $\langle n, m, \ell+1 \rangle$. If the worklist does not contain such an entry, enqueue $\langle n, m, 1 \rangle$.
- Permute the sequence to the front of the row.
- If the worklist contains an entry $\langle j, k, \ell \rangle$ such that this row does not have a consecutive sequence starting at position n, then the preceding row contained the bottom of the block. Store the block $\langle i-\ell-1, j, k, \ell \rangle$ in a set of recognized blocks.

3. For each stored block, execute a code fragment as given above for it.

Note that we may adapt the preceding algorithm in order to store only those blocks that have length and/or depth larger than some predefined constant. Too many very small blocks may cause unacceptable overhead.

We may analyze the performance of this technique as follows. Suppose we have recognized b blocks, with average length k and depth ℓ. It is easy to see that each such blocks has

$$4k\ell - \frac{k}{c} - \frac{\ell}{c} - \frac{k\ell}{c}$$

hits. The remainder of the matrix contains $\gamma N - bk\ell$ entries. Assuming that there are only misses on x, we can compute the total number of hits in this remainder as follows.

- N/c misses and $2(\gamma N - kb\ell) - N/c$ hits on y;
- $(\gamma N - kb\ell)/c$ misses and $(\gamma N - kb\ell) - (\gamma N - kb\ell)/c$ hits on A.

From this it follows that the hit ratio HR_3 is given by

$$HR_3 = \frac{3}{4} + \frac{bk\ell}{4\gamma N} - \frac{1}{4\gamma N}\left(\frac{kb}{c} + \frac{\ell b}{c}\right) - \frac{1}{4}\frac{1}{\gamma c} - \frac{1}{4c} \tag{4}$$

We obtain

$$HR_3 - HR_0 = \frac{b}{4\gamma N}\left(k\ell - \frac{k}{c} - \frac{\ell}{c}\right)$$

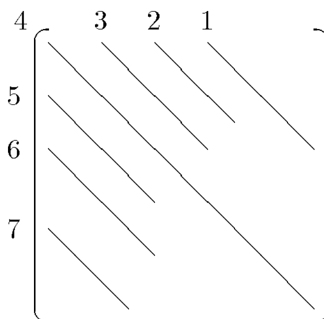
which is always larger than zero. We also see that if the number of blocks b, or the size of these blocks $k\ell$ increases, then the improvement increases also.

A variation on the above algorithm consists of recognizing not only dense blocks, but also blocks with "holes" in them. In this case, a sequence $n_1, n_2, \ldots, n_m$ is also considered consecutive if, for all j, $n_{j+1} - n_j \leq a$ for some small constant a. In practice, a will be typically 2 or 3. In this case, however, the length of a sequence is no longer constant. We can mark the end of such a sequence by replacing `IND(`j`)` by `IND(`j`) + N`. The code fragment we have to execute now is analogous to the previous code fragment and is therefore omitted.

6.3 Diagonal lines

Next we discuss how to deal with diagonal lines in a matrix. If a matrix contains a lot of diagonals, we may want to access these line by line. We can view this as *skewing* the access to the matrix. In this section we assume, for technical convenience, that the matrix *only* contains diagonal lines. Below we indicate how the general case can be handled.

The first problem we have to solve is in which order to access the lines. Consider the following matrix.



We store the coordinates of the left (top) end point of the diagonal $\langle i, j \rangle$ and its length ℓ. If we access the lines in increasing numbering, that is, from the upper-right to the lower-left corner, then we access every row from right to left. We order the descriptions of the diagonals in this way. That is, for any two diagonals $d_1 = \langle i, j, \ell \rangle$ and $d_2 = \langle i', j', \ell' \rangle$, d_1 is accessed before d_2 iff either $j > j'$ and $i < i' + \ell'$, or $i + \ell < i'$. This scheme of accessing diagonals enables us to find for each row the relevant j coordinate without extra computation or indirection.

Given a diagonal $\langle i, j, \ell \rangle$, we may execute the following code fragment for it. Note that we only need to use one loop.

```
DO ii = i,i+1
  y(i) = y(i) + A( HI(i) ) * x( IND( HI(i) )
  HI(i) = HI(i) - 1
ENDDO
```

The last statement in the loop removes the current j coordinate from the list of yet to access j coordinates. So an invariant of this techniques is that for each row i, the j coordinates in IND from LO(i) to HI(i) still have to be accessed.

We can recognize diagonal lines in a matrix in much the same way as we did for blocks. We maintain a working list of partially recognized lines. An entry in this list is given by a pair $\langle j, \ell \rangle$, where j is the j coordinate of a point on a line in the previous row, and ℓ is the length of the line seen so far. Then the line extends to the current row, if this row has an entry with coordinate $j+1$. So we enqueue $\langle j+1, \ell+1 \rangle$ if this is the case. Otherwise, we have reached the end of a line, and store $\langle i-\ell-1, j-\ell, \ell \rangle$ as a description of the line.

In case the matrix contains diagonal lines, and moreover other entries not on these lines, we can still use the above transformation, if we permute the j coordinates lying on a line to the right-hand side of the row. It is of course also possible to search for lines with a different slope than diagonal lines.

When we analyze the performance of this technique, we see that every reference to A will be a cache miss. The references to x, on the other hand, exhibit good locality. We can show that the hit ratio on x is $\lambda\left(1-\frac{1}{c}\right)$, where λ is the fraction of entries located on diagonal lines. However, if this fraction λ is large, we may decide to reorder the matrix and store it diagonalwise. If we do this, we can compute the hit ratio as follows. Suppose we have detected ℓ lines, with average length k. We obtain $4k\ell - 3\frac{k\ell}{c}$ hits in these lines. Analogously to the deduction in the previous subsection we can compute the hit ratio HR_4 as

$$HR_4 = \frac{3}{4} + \frac{k\ell}{4\gamma N} - \frac{k\ell}{2\gamma N c} - \frac{1}{4}\frac{1}{\gamma c} - \frac{1}{4c} \tag{5}$$

From this it follows that $HR_4 - HR_0$ is given by

$$HR_4 - HR_0 = \frac{k\ell}{4\gamma N}\left(1-\frac{2}{c}\right)$$

This means that the the hit ratio always increases, and if the number of lines and average increase, then so does the hit ratio.

7 Conclusions

In this paper we gave an initial description of how the difficult problem of compiler optimizations for the exploitation of data locality for sparse computation can be handled by techniques based on pattern matching. The simplified analytic analysis used in the paper seems to indicate that there is a big potential for these techniques to be useful. In order to come to more conclusive evidence for this, the techniques will have to be implemented and cache modeling should be used to prove the exact effectiveness. We plan to address this in more detail in forth-coming work.

References

[BEJW92] F. Bodin, C. Eisenbeis, W. Jalby, and D. Windheiser. *A quantitative algorithm for data locality optimazation.* Springer Verlag, Berlin, 1992.

[BW93a] A.J.C. Bik and H.A.G. Wijshoff. Compilation techniques for sparse matrix computations. In *Proc. Int. Conference on Supercomputing*, pages 416–424, 1993.

[BW93b] A.J.C. Bik and H.A.G. Wijshoff. On automatic data structure selection and code generation for sparse computations. In *Proc. 6th Int. Workshop on Languages and Compilers for Parallel Computing*, volume 768 of *Lecture Notes in Computer Science*, pages 57–75. Springer Verlag, Berlin, 1993.

[EJWB93] C. Eisenbeis, W. Jalby, D. Windheider, and F. Bodin. A strategy for array management in local memory. Special Issue od Matj. Programming B on Applications of Discrete Optimization, 1993.

[GJG87] D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformation. In *Proc. International Conference on Supercomputing.* Springer Verlag, Berlin, 1987.

[GJM86] K. Gallivan, W. Jalby, and U. Meier. The use of BLAS3 in linear algebra on a parallel processor with hierarchical memeory. *SIAM J. on Scientific and Statistical Computing*, 8(6), 1986.

[HP90] J.L. Hennessy and D.A. Patterson. *Computer Architecture, A Quantitative Approach.* Morgan Kaufmann Publishers, San Mateo, 1990.

[SBW91] J. Saltz, H. Berryman, and J. Wu. Multiprocessors and run-time compilation. *Concurrency: Practice and Experience*, 3(6):573–592, 1991.

[SW90] Y. Saad and H.A.G. Wijshoff. Spark: A benchmark package for sparse computations. In *Proc. on International Supercomputing*, pages 239–253, 1990.

[TFJ93] O. Temam, C. Fricker, and W. Jalby. Impact of cache interference on numerical codes cache behavior: Predicting and estimating, 1993. To appear in *Proceedings of the IEEE.*

[vHKK+92] R. von Hanxleden, K. Kennedy, C. Koelbel, R. Das, and J. Saltz. Compiler analysis for irregular problems in Fortran D. In *Proc. 5th Workshop on Languages and Compilers for Parallel Computing*, New Haven, CT, 1992.