Characterizing the Behavior of Sparse Algorithms on Caches

O. Temam, W. Jalby

IRISA/INRIA, Rennes, France

Abstract

Sparse computations constitute one of the most important area of numerical algebra and scientific computing. While there are many studies on the locality of dense codes, few deal with the locality of sparse codes. Because of indirect addressing, sparse codes exhibit irregular patterns of references. In this paper, the behavior on cache of one of the most frequent primitives SpMxV *Sparse Matrix-Vector multiply* is analyzed. A model of its references is built, and then performance bottlenecks of SpMxV are analyzed using model and simulations. Main parameters are identified and their role is explained and quantified. Then, this analysis is used to discuss optimizations of SpMxV. Moreover a blocking technique which takes into account the specifics of sparse codes is proposed.

Keywords: sparse primitives, cache, performance prediction, data locality.

1 Introduction

Due to the increasing difference between memory speed and processor speed, it becomes critical to minimize communications between memory and processor, by addition of caches on the data path. However, a consequence of this worsening difference is that the cost of a cache miss, in terms of processor clock cycles, is becoming quite large, making it critical to improve the hit ratio.

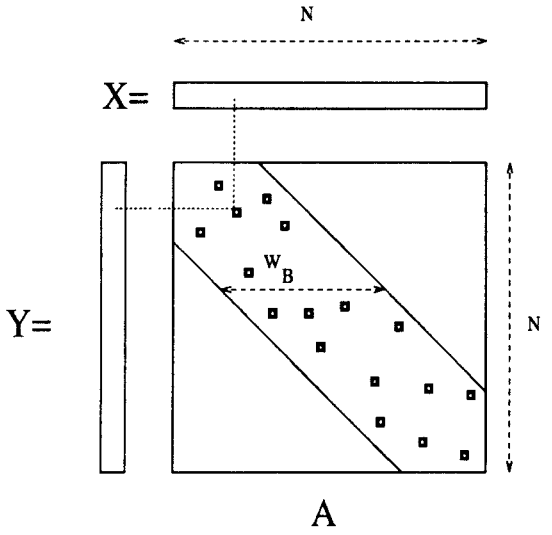
Numerical codes are now some of the most demanding programs in terms of execution time and memory usage. The existing literature related to the study of numerical codes behavior on cache memories focuses on regular do-loops, i.e with linear references to arrays [8, 2]. There is an important set of numerical codes, "sparse codes", which do not belong to this category. Sparse numerical codes like classic numerical codes are made of a collection of simple numerical primitives. We chose to study *Sparse Matrix-Vector multiply* (SpMxV) because it is among the most frequently used ones along with *gaussian elimination* [3], and still it is simple enough to allow a sharp analysis of its workings (cf. figure 5); moreover, a number of sparse primitives exhibit rather similar patterns of references (i.e a few arrays addressed regularly and indirect addressing to another array). Because of indirect addressing, sparse codes have the particularity of breeding *irregular patterns of references to memory and to cache*, and consequently, the behavior of caches under numerical workloads is seemingly non-predictible and hard to analyze. Because of this *apparently random* behavior, caches, which principles rely on locality of programs, are generally said to be inefficient with sparse codes [14].

However, in this paper, we show that this assumption is true only for a restricted domain of main problem parameters (cache size, line size, matrix bandwidth and number of non-zero elements). Even then, in some cases, it is possible and *worthwhile* to exploit unused locality through software techniques. Though classic blocking methods hardly allow the utilization of this locality, it is possible to exploit it through blocking techniques that take into account the specifics of sparse matrices.

In section 2 of this paper, spatial and temporal locality of SpMxV is qualitatively evaluated and potential problems are identified. Then, in section 3, a meaningful share of the paper is devoted to modeling the non-regular references which appear in SpMxV are presented (see [12] for details of the model), because a necessary prerequisite to evaluating and optimizing a primitive is a good understanding of its behavior. Besides, the purpose of this section is to show that it is possible to predict the behavior and performance of sparse codes, and to actually *quantify* their impact on caches. In section 4, using the model and simulations, the behavior of SpMxV is characterized according to the values of the parameters. Finally, in section 5, software optimizations are discussed, and a blocking technique based on the observations of the previous section is presented.

2 A qualitative study of locality within SpMxV

Position of the problem The purpose of the paper is to analyze SpMxV on caches. *Storage-by-row* has been chosen because it is among the most commonly used storage techniques (see page 2 for more details). Otherwise, for sake of simplicity the cache is assumed to be direct-mapped. It can be seen in section 4.4 that this hypothesis is not very restrictive since set-associative and direct-mapped caches exhibit relatively similar behaviors with SpMxV, and that the model built can be extended to set-associative caches (cf. [12]).



N	:	Matrix dimension
N_{nz}	:	Total number of non-zero elements
$n_{nz} = \frac{N_{nz}}{N}$	:	Average number of non-zero elements per row
W_B	:	Matrix bandwidth
Sets	:	1 (Direct-mapped cache)
C_S	:	Cache size
L_S	:	Line size

Figure 1: *Banded sparse matrix; matrix and cache parameters.*

Remark: *All cache dimensions are divided by the size of a single-precision floating-point element (4* bytes*), so that a size S actually corresponds to $S \times 4$* bytes.

Cross-interference misses (or conflict misses) : an array element flushed by an element of another array.
Self-interference misses (or capacity misses) : an array element flushed by an element of the same array.
Intrinsic misses (or compulsory misses) : an array element loaded for the first time.

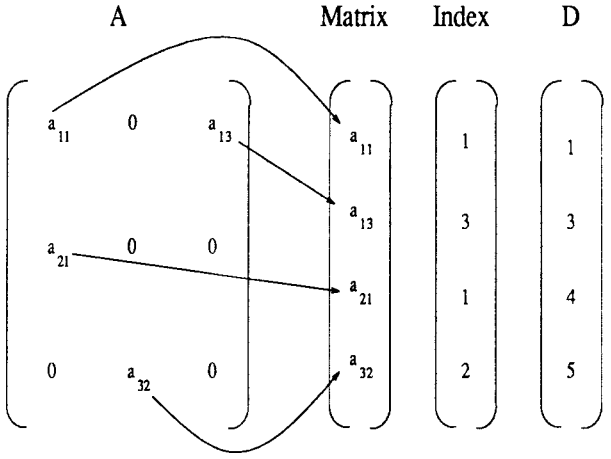


Figure 2: *Example of storage-by-row for a 3×3 sparse matrix*

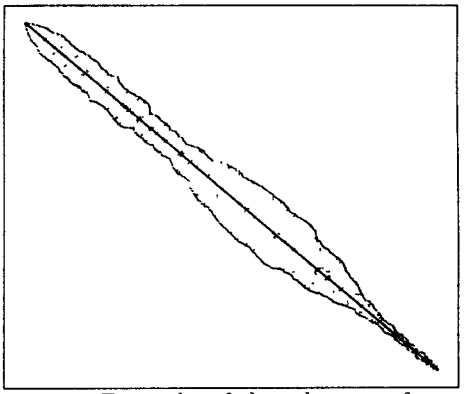


Figure 3: *Example of distribution of non-zero elements within finite-element matrices (matrix* 1138 BUS *of the Harwell-Boeing suite).*

```
DO I=1,N
   DO J=1,N
      Y(I) = Y(I)+A(I,J)*X(J)
   ENDDO
ENDDO
```

```
DO I=1,N
   REG = Y(I)
   DO J=D(I),D(I+1)-1
      REG = REG + Matrix(J)* X(Index(J))
   ENDDO
   Y(I) = REG
ENDDO
```

Figure 4: *Original loop nest; storage by row.*

Figure 5: *Problem parameters*

2.1 Data locality

A first step to understanding the behavior of SpMxV on cache is to study the locality of the data used by this primitive. Since there are N_{nz} references to arrays X, *Index* and *Matrix*, and $2N$ references to arrays Y and D, the total number of references is $3 * N_{nz} + 4 * N$ (cf. figure 5).

Arrays Y and D have very similar behaviors in the sense that they both exhibit flawless spatial and temporal locality. Most probably $D(I)$ and $D(I+1)$ will be stored in registers and therefore should not provoke a reference to memory on each iteration of loop J. So arrays Y and D (of size N) are mostly responsible for *intrinsic misses*, and therefore account for a small share of total cache misses (since, in general $N \ll N_{nz}$).

Arrays *Matrix* and *Index* have no temporal locality and again exhibit flawless spatial locality. These two arrays account for $2N_{nz}$ references, that is, a major part of total references. Since no element is reused, cache misses due to these arrays are only *intrinsic misses*. Because of their size, they may also provoke important *cross-interferences* with other arrays (i.e, flush other arrays from cache).

Array X: Because of the indirect addressing through array *Index*, array X exhibits a complex behavior. If a uniform distribution of non-zero elements on a row within the band (of size W_B) is assumed, then the average distance between two columns with non-zero elements is $\frac{W_B}{n_{nz}}$. Therefore, in most cases there is some spatial locality if $\frac{W_B}{n_{nz}}$ is of the order of L_S. Actually, in usual *finite-element* matrices, the non-zero elements are sometimes grouped along specific diagonals, within the band (cf. [4] and figure 3). In that case, the spatial locality may not be negligeable even if $\frac{W_B}{n_{nz}} \gg L_S$.

Array X is the only array which presents an unexploited temporal locality, and therefore from which significant gains can be expected; however the temporal locality of X is non-trivial and therefore hard to analyze and exploit. That is why our efforts will mainly focus on analyzing the behavior of array X. Due to the properties of sparse matrices (especially *finite-element* ones), if there is approximately n_{nz} non-zero elements per row, there is also about n_{nz} non-zero elements per column. A first consequence of that observation is that each element of X may theoretically be reused n_{nz} times at best. Secondly, the average distance (in terms of iterations of loop I) between two reuses is approximately $\frac{W_B}{n_{nz}}$. Meanwhile, about $\frac{W_B}{n_{nz}} \times 3n_{nz}$ elements (from arrays X, *Matrix* and *Index*) are loaded into cache and may flush the elements to be reused.

The conclusion of the previous observations on X is that whether temporal locality and spatial locality of X are significant and can be exploited highly depends on W_B, C_S, L_S and n_{nz}.

3 Modeling: understanding and quantifying

From previous section, it appears that the main source of cache misses are misses of *Matrix* and *Index* which can be evaluated easily because they are intrinsic misses, and misses of X which are hard to estimate because addressing to array X is indirect and irregular. The misses of X are mainly *cross-interference* or *self-interference* misses. First of all, the simulations done (cf. section 4) show that the role of cross-interference and self-interference phenomenons on X is very similar. Second, both of the two kinds of misses can be modeled using techniques presented thereafter. Third, because only main trends need to be identified, the purpose of our model is to provide a good understanding of the interactions between parameters rather than an accurate formula of the total number of cache misses. Therefore, for sake of simplicity, only self-interference misses are precisely modeled and quantified (cf. section 3.1).

3.1 Modeling self-interferences of array X

References to array X are highly irregular and consequently cannot be investigated through classic *deterministic* methods. Therefore, *probabilistic modeling* is being used. The main problem seems to choose a distribution which matches that of non-zero elements on a row within the band. Though it is for the least possible to find an approximate distribution for *finite-element* matrices, this yields formulas which are too complex to handle (cf. [12]). Therefore, though most computations are conducted for any distribution $p(i,j)$ (probability that element (i,j) of A is non-zero), *uniform distribution* ($p(i,j) = p$) is employed for final calculi.

The object of section 3.1 is to build the model of references to array X and compute the number of cache misses. This part describing model elaboration can be skipped, though it provides an insight on the behavior of SpMxV.

3.1.1 Simplifying expression of the original problem

Let us consider original matrix A. All non-zero elements of A located on column j of this matrix breed a reference to element j of X. Now, let us consider the c^{th} cache location. All elements j of X such that $j \ mod \ C_S \in [c, c + L_S - 1]$ are mapped to the same cache line c. Therefore, all columns j of A such that $j mod C_S \in [c, c + L_S - 1]$ breed references to elements of X which are mapped to the same cache line.

Therefore, it is possible to divide the problem into $\frac{C_S}{L_S}$ sets of elements of X, all elements within a set being mapped to the same cache line. Similarly, A is divided into $\frac{C_S}{L_S}$ sets of columns. all breeding references to elements of X mapped to the same cache line (cf. figure 6). Since, the cache is direct-mapped, none of these sets interact with each other.

So, if the original problem can be formalized as follows

Problem P: *Compute an approximation of* $N_{cm}^{self} = N_{cm}(P)$, *the number of self-interference misses of array* X *in the sparse matrix-vector multiplication. The dimension of* X *is equal to* N. A *is an* $N \times N$ *matrix, and the cache is direct-mapped and of size* C_S.

Then P, is now equivalent to C_S subproblems $P_i, (1 \leq i \leq \frac{C_S}{L_S})$. Through simulations, it is possible to check that, for distributions occuring in *finite-element* matrices (non-zero elements are grouped along three diagonals) and even more for *uniform* distributions, it is a very fair approximation to assume that all subproblems P_i are equivalent to subproblem $P^{'}$.

Problem $P^{'}$: *Compute an approximation of* $N_{cm}(P^{'})$, *the number of self-interference misses of array* $X^{'}$. *The dimension of* $X^{'}$ *is equal to* $N^{'}$ *(where* $N^{'} = \lfloor \frac{N}{C_S} \rfloor \times L_S$ *or* $N^{'} = (\lfloor \frac{N}{C_S} \rfloor + 1) \times L_S$*).* $A^{'}$ *is an* $N \times N^{'}$ *matrix, and the cache is direct-mapped and of size* L_S.

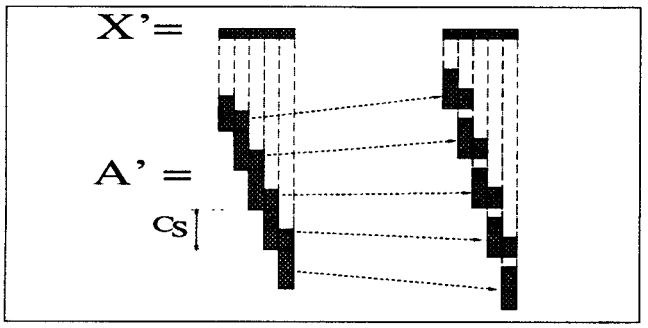


Figure 6: *Decomposition of subproblem* $P^{'}$

Now, $A^{'}$ can be decomposed into sections of C_S rows, which would all have the same shape as shown on figure 6. Since, in general, $C_S < N$, there is a great number of such sections in $A^{'}$ (approximately $\frac{N}{C_S}$). Through experiments, it can be observed that the number of cache misses corresponding to the execution of each section becomes rapidly stable. Therefore, it is possible to restrict the study to only one section $S^{'}$ of $A^{'}$.

Let us do an ultimate simplification. The number of columns in the sections described above is not constant, because of the banded shape of the matrix. In order to ease the computations even more, a section $S^{'}$ can be divided into two parts $S_1^{''}$ and $S_2^{''}$, each with a constant number of columns (cf. figure 6). The characteristics of each subsection are the following ones:

$$S_1^{''} \begin{cases} n_c^1 = \lfloor \frac{W_B}{C_S} \rfloor \\ n_l^1 = C_S - W_B \ mod \ C_S \end{cases}$$
$$S_2^{''} \begin{cases} n_c^2 = \lfloor \frac{W_B}{C_S} \rfloor + 1 \\ n_l^2 = W_B \ mod \ C_S \end{cases}$$

Problem $P^{''}$ can now be defined as follows:

Problem $P^{''}$: *Compute an approximation of* $N_{cm}(P^{''})$, *the number of self-interference misses of array* $X^{''}$. *The dimension of* $X^{''}$ *is equal to* $N^{''}$ *(where* $N^{''} = n_c$*).* $A^{''}$ *is an* $n_l \times n_c$ *matrix, and the cache is direct-mapped and of size* L_S.

3.1.2 Estimating the number of cache misses

Let us now formalize the notion of "cache miss" within the scope of problem $P^{''}$. During execution of SpMxV, elements of a subsection $A^{''}$ of problem $P^{''}$ are referenced row-wise. Let us call $\pi_{out}^k(i,j)$ the probability that element k of $X^{''}$ be out of cache right before element (i,j) of the subsection is referenced. Let us also call $p(i,j)$ the probability that element (i,j) of $A^{''}$ be a non-zero element. Now, the probability that element j of $X^{''}$ is not in cache, right before element (i,j) of $A^{''}$ is being considered, is equal to $\pi_{out}^j(i,j)$. Therefore, the probability for a cache miss to occur at that moment is equal to $p(i,j) \times \pi_{out}^j(i,j)$.

Then, the number of cache misses due to $A^{''}$ is given by the following expression:

$$N_{cm}(P^{''}) \simeq \sum_{i=1}^{n_l} \sum_{j=1}^{n_c} \pi_{out}^j(i,j) \times p(i,j)$$

And, the total number of cache misses is equal to (see [12] for more details):

$$\begin{aligned} N_{cm}(P) &\simeq \frac{C_S}{L_S} N_{cm}(P^{'}) \\ &\simeq \frac{C_S}{L_S} \times \frac{N}{C_S} N_{cm}(P_1^{''} \bigcup P_2^{''}) \\ &\simeq \frac{N}{L_S} \times (N_{cm}(P_1^{''}) + N_{cm}(P_2^{''})) \end{aligned}$$

When distribution is uniform $p(i,j) = p$ and p can be given as a function of problem parameters $p = 1 - (1 - \frac{N_{nz}}{NW_B})^{L_S}$ (see [12] for more details). For this distribution of probability, the total number of self-interference misses on X is given by expression of figure 7.

3.2 Number of cache misses

3.2.1 Self-interferences of array X

$\frac{W_B}{C_S} > 1$: Since $N_{nz} \ll N \times W_B$ when W_B is sufficiently large, it can be assumed that $p \ll 1$ and that $p \simeq \frac{N_{nz}}{NW_B} \times L_S$. Therefore, a first order development of expression in figure 7 gives (cf. [12] for more details)

$$N_{cm}^{self} \simeq N_{nz} - \frac{N_{nz}^2 \times C_S}{2 \times N \times L_S \times W_B} \qquad (1)$$

$\frac{W_B}{C_S} < 1$: In that case, there are no self-interferences of array X because all active elements of X fit in the cache, therefore

$$N_{cm}^{self} = 0$$

$$N_{cm}^{self} \simeq \frac{N}{L_S} \times \left(\frac{pn_c^1}{1-(1-p)^{n_c^1}} \times \left[n_l^1 (1-(1-p)^{n_c^1-1}) + p(1-p)^{n_c^1-1} \frac{1-(1-p)^{n_c^1 n_l^1}}{1-(1-p)^{n_c^1}} \right] + \frac{pn_c^2}{1-(1-p)^{n_c^2}} \times \left[n_l^2 (1-(1-p)^{n_c^2-1}) + p(1-p)^{n_c^2-1} \frac{1-(1-p)^{n_c^2 n_l^2}}{1-(1-p)^{n_c^2}} \right] \right)$$

Figure 7: *Expression of the total number of self-interference misses on array X*

3.2.2 Intrinsic misses

There are N references to D, Y and X, and N_{nz} references to *Matrix* and *Index*. Therefore, the total number of intrinsic misses is given by:

$$N_{cm}^{int} = \frac{3N + 2N_{nz}}{L_S} \quad (2)$$

Since in general $N_{nz} \gg N$, arrays *Matrix* and *Index* represent the main source of intrinsic misses.

4 Highlighting the role of the problem parameters

Of course, all problem parameters have an impact on SpMxV. However, through model analysis and experiments, three coefficients (L_S, $w = \frac{W_B}{C_S}$, $d = \frac{n_{nz}}{W_B}$) proved to have a major impact on the hit ratio, and are sufficient to characterize most phenomenons. L_S is a critical parameter, mainly because of its influence on *intrinsic misses*, but also on *cross-interference misses*. Parameter $w = \frac{W_B}{C_S}$, called *degree of interference*, it indicates how many elements of X conflict for the same cache line (cf. section 3.1.1), and therefore it reflects quite well the degree of self-interferences occuring on X. Finally, parameter $d = \frac{n_{nz}}{W_B}$, called *density*, corresponds to the average distance between two non-zero elements on a row and on a column of original matrix A (cf. section 2). In other terms, it is a measure of the degree of temporal and spatial locality of non-zero elements of matrix A, and consequently, of the references to array X.

Basing our analysis on the model obtained in section 3 and simulations, the role and importance of the above parameters is discussed in the following subsections. A small subsection is also devoted to discussing the difference between direct-mapped and set-associative caches. For sake of simplicity, the simulations, which were used to make the graphs of this section, are mainly based on uniformly distributed matrices, but account quite well for phenomenons occuring in real sparse matrices.

4.1 Line size L_S

Influence of L_S on the intrinsic misses of *Matrix* and *Index* The expression of the number of intrinsic misses (2) shows that this number decreases hyperbolically with L_S (cf. figure 8). Therefore, a small increase of L_S brings important reductions of the number of intrinsic misses.

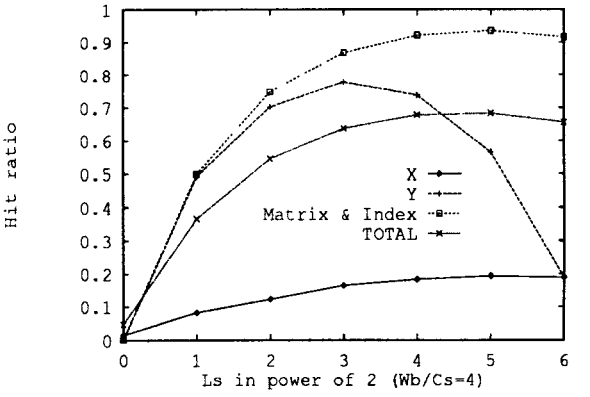


Figure 8: *Influence of L_S on the total hit ratio and the hit ratio of each array.*

Influence of L_S on self and cross-interference misses Let us assume that $\frac{W_B}{C_S} > 1$. In the case where non-zero elements are uniformly distributed across the sparse matrix, the approximate number of self-interference misses is given by (1). This expression is a function of L_S: $N_{cm}^{self} = \alpha - \frac{\beta}{L_S}$.

Therefore, N_{cm}^{self} grows with L_S, though this increase is more or less moderate depending on β. Otherwise, $N_{cm}^{cross} = \frac{2N_{nz}^2 \times L_S}{C_S}$, therefore the number of cross-interference misses grows linearly with L_S. Consequently, an increase on L_S corresponds to an increase on both the number of self-interference and cross-interference misses (cf. figure 9).

The consequence of the previous observations is that the hit ratio of all arrays but X increases very quickly (hyperbolically) with L_S. On the other hand, because of cross and self-interference misses, the hit ratio of X increases much more moderately (or sometimes even decreases) when L_S grows.

Therefore, for small values of L_S the main cause of cache misses are arrays Matrix and Index, while for high values of L_S, X accounts for the major part of cache misses (cf. figure 10).

Consequently, devoting important efforts to benefit from the temporal locality of X should be considered only when L_S is such that X becomes a major cause of cache misses, otherwise little improvements of total hit ratio can be expected (cf. figure 10).

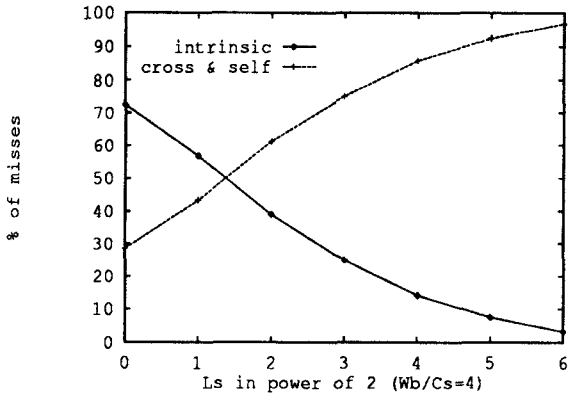


Figure 9: *% of* intrinsic misses *and* self + cross-interference misses *for different values of* L_S.

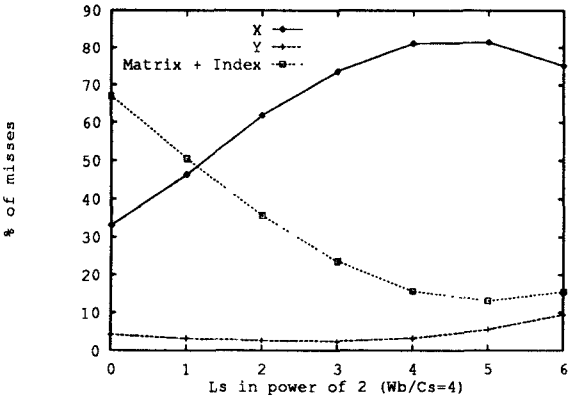


Figure 10: *% of misses of each array for different values of* L_S.

Proper values of L_S for finite-element sparse matrices Due to the properties of mesh structures (where each node has a relatively constant number of neighbors) and the use of renumbering algorithms to minimize bandwidth, finite-element sparse matrices exhibit a non-uniform distribution of non-zero elements. They are grouped by packs of 2, 3, 4 or more elements depending on the mesh type. This spatial locality of non-zero elements induces a spatial locality of references to array X. Therefore, $L_S = 2$ *or* 4 is sufficient to make use of this locality, while little improvement can be expected for higher values of L_S.

As it can be seen on figure 11, the reduction of cache misses between $L_S = 1$ and $L_S = 2$ is impressive, while it steps down after $L_S = 4$. This phenomenon is characteristic of finite-element matrices. It can be noted on figure 11 that an increase of L_S breeds progressive instead of drastic improvements on the hit ratio of X, when the non-zero elements are uniformly distributed within the sparse matrix. The

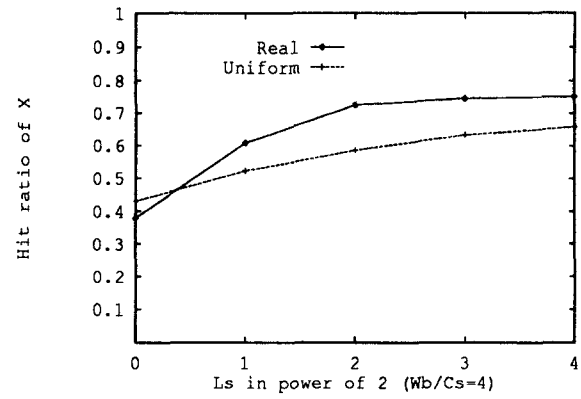


Figure 11: *Influence of the distribution of non-zero elements on the spatial locality of references to X*

shape itself of finite-element sparse matrices (cf. figure 3) suggests a greater spatial and temporal locality than of uniformly distributed matrices. However, both uniform and finite-element distributions tend to behave similarly for large enough cache line sizes, i.e when the locality effect of finite-element sparse matrices does not show anymore.

4.2 Degree of interference $w = \frac{W_B}{C_S}$

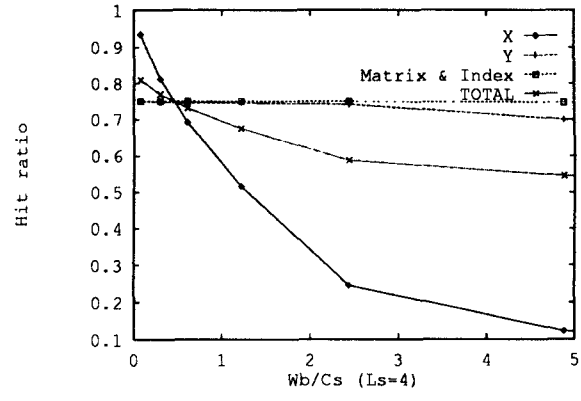


Figure 12: *Influence of* $\frac{W_B}{C_S}$ *on the total hit ratio and the hit ratio of each array.*

Influence of w for self-interference misses The effect of W_B and C_S cannot be dissociated. For $W_B < C_S$, the number of self-interference misses due to X is equal to zero. This appears clearly when considering *simplified problem* P'': the number of columns of A'', i.e the number of interfering columns, is equal to $\lfloor \frac{W_B}{C_S} \rfloor$ or $\lfloor \frac{W_B}{C_S} \rfloor + 1$, that is, 0 *or* 1. Therefore, once an element is loaded into a cache line, it cannot be flushed by another element. So, *for* $W_B < C_S$, *the number of cache misses due to* X *is optimum.* Now,

for $W_B > C_S$, model expression (1) shows that the number of self-interference misses increases hyperbolically with w: $N_{cm}^{self} = \alpha - \frac{\gamma}{w}$.

Now, if the previous expression is considered as a function of w, and is differentiated it, then it appears that the increase of N_{cm}^{self} becomes small (i.e less than 10 %) whenever $w \geq \sqrt{10\alpha}$.

So, three cases can occur. First $w < 1$, the number of self-interference misses is negligeable and it is not useful to reduce W_B. Second $w \simeq 1$, in this interval the number of self-interference misses increases hyperbolically with w, and therefore very significant improvements can be obtained through slight bandwidth reduction. Third $w \gg 1$, and the number of self-interference misses is close to maximum (nearly no element of X is reused), and only a drastic bandwidth reduction may bring improvements.

Cross-interference misses When $w \gg 1$ is sufficiently large, because of self-interferences only, there is little reuse on X. Therefore, the effect of *Matrix* and *Index*, i.e cross-interferences, can only be redundant with that of X.

When $w \leq 1$, there are no self-interference misses. In that case, *alive* (i.e currently used) elements of X are located in an area of size W_B within cache. Now, *Matrix* and *Index* can be considered as two "trains" of references moving across the cache. Therefore, the larger W_B, the higher the probability that these "trains" meet the area of alive elements of X, i.e the higher the probability of cross-interferences. Still, these cross-interference misses account for a relatively small share of total misses, unless L_S is large, i.e there are few *intrinsic misses* (cf. figure 13).

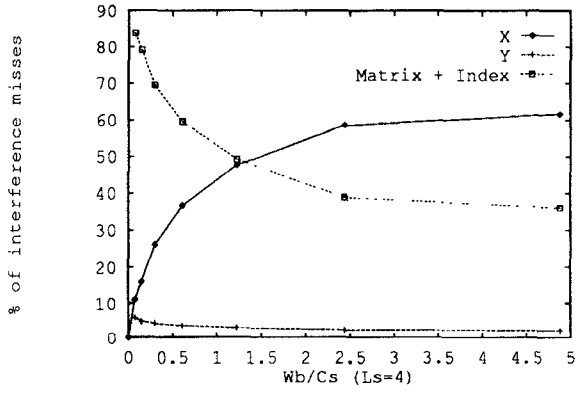


Figure 13: *% of interference misses for different values of* $\frac{W_B}{C_S}$.

Current values of $\frac{W_B}{C_S}$ Let us now try to see what are the values of $\frac{W_B}{C_S}$ currently found.

The size of numerical problems tends to grow, and consequently W_B grows accordingly. When *bandwidth reduction* renumbering algorithms are employed, $W_B \simeq \sqrt{N}$ *or* $W_B \simeq N^{\frac{2}{3}}$ *or* $W_B \simeq \frac{N}{10}$ according to the problem type, while $W_B \simeq N$ when *minimum-degree* renumbering algorithms are used. Large current problem sizes range from $N = 10^5$ to $N = 10^6$, therefore W_B generally varies between $W_B = 300$ and $W_B = 10^6$.

For single-cache machines, C_S clearly tends to grow. C_S values of more than *256 Kbytes* can now be found. So, if cache sizes increase fast enough $\frac{W_B}{C_S}$ will soon be on the "safe zone" (i.e $\frac{W_B}{C_S} \ll 1$; *256 Kbytes* cache, $N = 10^5$, $W_B = N^{\frac{2}{3}}$, $\frac{W_B}{C_S} = 0.01$). On single-cache machines equipped with current-size caches (i.e, $C_S \simeq$ *4192* or *16 Kbytes*), a large third-dimensional problem (i.e, $N \simeq 10^5$, $W_B \simeq N^{\frac{2}{3}}$ to $W_B = N$) exhibits a $\frac{W_B}{C_S}$ ratio of *0.5* to *3.5*.

The increasing popularity of multi-level caches makes small (primary) caches more frequent. The size of such caches is currently of the order of *4 Kbytes*. What is more, N also tends to grow, and *minimum degree* is a rather popular algorithm. So very large ratios $\frac{W_B}{C_S}$ may become more frequent (*4 Kbytes* cache, $N = 10^5$, $W_B = N$, $\frac{W_B}{C_S} = 100$).

4.3 Density $d = \frac{n_{nz}}{W_B}$

Let us now consider parameter $d = \frac{n_{nz}}{W_B}$. d can be considered as the "density" of non-zero elements on one row of matrix A. When d is very high matrix A looks very much like a banded *dense* matrix, while the matrix is "very sparse" when d is relatively small.

Depending on w, the density of non-zero elements induces two different phenomenons.

If $w < 1$, the smaller W_B (i.e the larger d) the less cross-interferences occur (cf. paragraph 4.2). Consequently, it is possible to benefit from the temporal locality of references to X (cf. graph $L_S = 1$ of figure 14). For the same reason, it is also possible to benefit from spatial locality. Indeed, once an element of X is loaded $L_S - 1$ consecutive elements of the same array are also loaded. Though they are not immediately referenced, they are not flushed from cache (as seen above), and therefore they stay into cache until they are referenced. That is why array X also benefits from spatial locality in this case, independently of the distribution of non-zero elements (cf. paragraph 4.1 and figure 14).

Now it can be observed that, for a fixed value of W_B, *when* n_{nz} *is large there exists an important potential reuse on* X. Since it is possible to benefit from temporal locality when $w < 1$, the higher n_{nz} (i.e the larger d) the higher the hit ratio of X (cf. figure 14). Nevertheless, a high value of n_{nz} slightly worsens cache pollution, though n_{nz} must be quite large for this phenomenon to counterbalance the benefits from spatial and temporal locality (cf. figure 14).

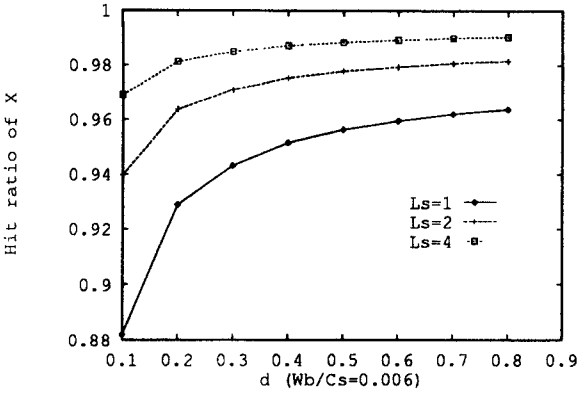


Figure 14: *Influence of the density on the hit ratio of X for* $\frac{W_B}{C_S} < 1$.

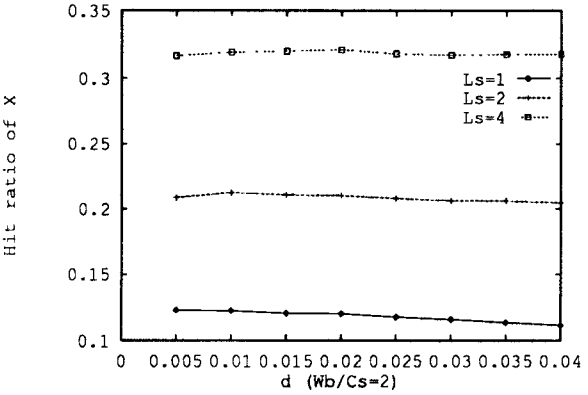


Figure 15: *Influence of the density on the hit ratio of X for* $\frac{W_B}{C_S} > 1$.

When $w > 1$, the behavior of X is not correlated to d anymore because interferences (cross and self) occur so often that benefiting from temporal and spatial locality becomes hypothetical (cf. figure 15).

Usual values of d For 2-dimensional finite-element problems, the average number of non-zero elements per row is of the order of *10*, while it is of the order of *100* for 3-dimensional problems. *So essentially 3-dimensional problems are worth optimizing.* As it has been seen in paragraph 4.2, W_B ranges from $\sqrt{N}$ (2-dimensional) or $N^{\frac{2}{3}}$ (3-dimensional) to N. Typically, the density of a 3-dimensional problem may range from *0.001* to *1*.

4.4 Set-associative caches

Though the model presented in section 3 corresponds to direct-mapped caches, it can be extended to set-associative caches (cf. [12]). Nevertheless, simulations can already show that associativity brings little

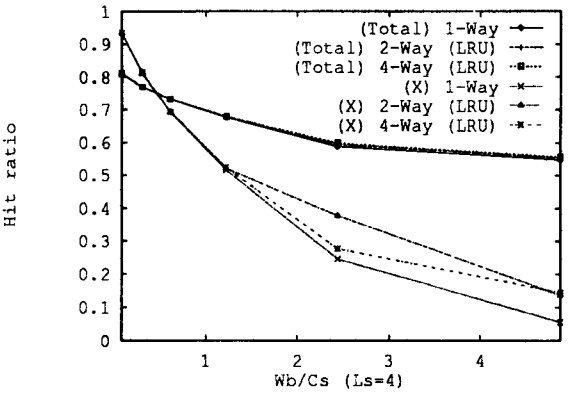


Figure 16: *Performance comparison of set-associative and direct-mapped caches for different values of w*

improvements on total hit ratio, and more particularly on the hit ratio of X. Basically associativity is helpful when interferences occur. Now, when $w < 1$, there are very few interferences, and when $w > 1$, interferences are so numerous (at least w elements of X conflict for the same cache location) that a 2-way or 4-way associativity brings little or no improvement (cf. figure 16).

5 Improving the behavior of SpMxV

In this section, possible software optimizations are discussed. Two different approaches for software optimization of SpMxV are distinguished: an algorithmic approach which aims at reducing bandwidth using renumbering algorithms, and a software approach based on particular blocking techniques.

5.1 Software optimization: *Bandwidth reduction*

Two main kinds of renumbering algorithms are employed: *bandwidth reduction* algorithms which are derivatives of that of Cuthill and McKee, and the *minimum-degree* algorithm. According to George [6], reducing bandwidth is not closely related to minimizing arithmetic operations and storage. On the other hand, *minimum-degree* algorithm is efficient for finding low-fill orderings. Therefore, this second renumbering scheme tends to become popular. However, it must be noted that an effect of *minimum degree* is to scatter non-zero elements across the matrix, while bandwidth reduction algorithms are generally very efficient in grouping non-zero elements. So, if *minimum degree* is more efficient for LU factorization as shown by George, it is far less profitable for SpMxV in terms of locality, because it widens considerably matrix bandwidth.

Therefore, if SpMxV is used in a direct sparse code, a tradeoff exist between optimizing SpMxV or LU factorization. Now, in iterative codes, SpMxV is the most frequently used primitive and matrix renumbering can be chosen for SpMxV to behave most efficiently. Any-

way, it should be kept in mind, that renumbering in order to minimize bandwidth is profitable in our case, only if the degree of interference w can be made smaller than *1*.

5.2 Software optimization: *Blocking*

5.2.1 Classic blocking

The reason why sparse codes do not work on caches when matrix parameters are much larger than cache parameters is the same as for dense codes: because elements to be reused cannot be kept in cache. However, a solution valid for dense codes [2], i.e blocking, is not valid for sparse codes, because each element is not reused a sufficient number of times to override the overhead of blocking.

Moreover, sparse codes exhibit an *irregular locality* which cannot be foreseen at compile-time (i.e *a priori*), while dense codes exhibit *regular locality*, which can be exploited at compile-time.

When classic blocking is used, part of this irregular locality cannot be used because of the average distance between two reuses ($\simeq \frac{n_{nz}}{W_B}$ iterations). Moreover, large overhead data is required in order to locate non-zero elements within each block. Only if the matrix is nearly dense ($d \simeq 1$), classic blocking is profitable due to the important potential reuse per element of X (cf. figure 17).

5.2.2 Blocking by diagonal

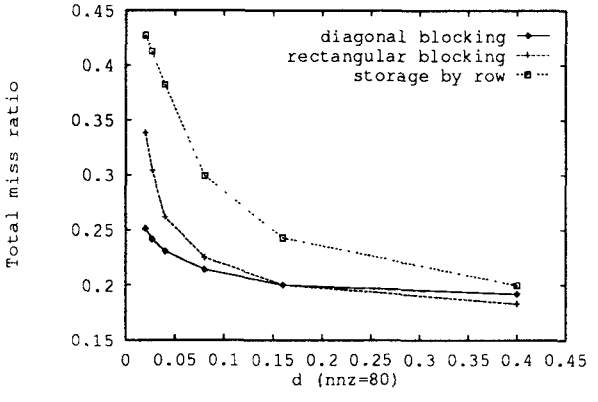


Figure 17: *Effect of diagonal blocking on the total miss ratio.*

It is clear that ai y software optimization techniques should take into account the specifics of SpMxV. First of all, in opposition to most dense codes, it must be assimilated that one computation corresponds to one *non-zero* element and not to any element of the matrix. Therefore, any blocking technique should implicitly deal with *blocks of non-zero elements* rather than *blocks of matrix elements.*

Second of all, in dense matrices, the elements of symmetry are *columns and rows*, which explain why such matrices are blocked using rectangular shapes. The elements of symmetry of banded sparse matrices are *diagonals.* Therefore, it seems natural to block along diagonals, otherwise too many blocks would be half-empty and would degrade the efficiency of the blocking technique.

According to section 4.2, the original large band should be split into several small bands, such that their width is of the order of cache size. Now, these blocks should be based on non-zero elements and not on arbitrary geometric dissection. Moreover, having the number of non-zero elements on each of the different diagonals constant (except for the first and last diagonal blocks) would reduce the overhead data to be kept, as seen in above paragraph.

For that purpose, n_{nz} the average number of non-zero elements per row and the number of non-zero elements on each diagonal of the original matrix must be computed. Since the goal of the method is to obtain a collection of banded matrices which bandwith is smaller than cache size, the original matrix needs to be split into approximately $n_B = \lceil \frac{W_B}{C_S} \rceil$ submatrices. In all these submatrices except for the first and last one, the number of non-zero elements ought to be constant, approximately equal to $\frac{n_{nz}}{n_B}$. Additional data is necessary to store the number of non-zero elements on the rows of the first and last diagonal block.

This blocking method is interesting only when X is the major cause of cache misses, and the potential reuse is high (cf. sections 4.1 and 4.3). Though it increases the number of misses on Y (they are nearly multiplied by n_B), it decreases the miss ratio of X, so that the improvement of the total hit ratio may be quite important (cf. figure 17). The main asset of the method is to be applicable even when the density is low, where other classic blocking methods would fail.

Moreover, the shape of certain types of matrices such as finite-element matrices suggests that diagonal blocking could be further enhanced. Indeed, in such matrices, there are generally three diagonals along which non-zero elements are grouped (cf. figure 3). Therefore a first step to diagonal blocking would be to find the diagonals of "highest density" and block non-zero elements around them. Not only, nearly dense blocks of non-zero elements would be obtained, but the number of blocks itself (and the overhead) would be considerably reduced in many cases, since it would be fixed (and would not depend on W_B and C_S).

However, all blocking techniques introduce bounds of their own. In our case, the number of blocks determine the maximum reuse per element of X. If there are n_B blocks and n_{nz} non-zero elements per row, then *diagonal blocking* authorizes a maximum reuse of $\frac{n_{nz}}{n_B}$ per element of X, while the theoretical maximum is n_{nz}.

5.2.3 Parallelization

When a multiprocessor machine with local caches is considered, it may appear that the most natural and most commonly employed methods for blocking sparse

matrices (e.g. blocking by rows) may not be the best regarding communications between the different levels of memory. For instance, the original *storage-by-row* scheme already allows an easy parallelization of SpMxV (cf. figure 5) on loop I. However, analysis of section 4 shows that a large bandwidth breeds an important miss ratio of X. Now, if SpMxV is blocked on loop I and the bandwith of original matrix is large, then the bandwidth of the resulting submatrices remains the same. Therefore, the miss ratio of X on each local cache may be high.

On the other hand, *blocking by diagonal* has several assets which can be profitable to parallelization *and* locality also, especially by improving data reuse without bringing significant overheads. Since the original problem (a banded sparse matrix) is split diagonally, the submatrices bandwidth is smaller than that of original matrix (actually it is equal to that of original matrix divided by the number of blocks). Therefore, on each local cache the miss ratio of X may be far smaller than when blocking by row. Consequently, parallelizing this way naturally *decreases the burden on local caches* by reducing the bandwidth, and consequently the hit ratio.

6 Conclusions

This paper presents a methodology for modeling the irregular references of sparse codes using probabilistic methods. The model was shown to be very accurate for uniformly distributed matrices, *finite-element* matrices renumbered with *minimum degree algorithm*, and still reflects quite well the behavior of *finite-element* matrices renumbered with *bandwidth reduction algorithm*.

The analysis of the model and simulations allowed to identify three main parameters and their impact on the behavior of SpMxV. First of all, cache size and bandwidth are closely dependent. When bandwidth is smaller than cache, spatial and temporal locality of sparse matrices is well exploited and SpMxV needs not be optimized. On the opposite, when bandwidth is greater than cache size, self and cross-interferences degrade the reuse of vector X which cannot exploit its temporal and spatial locality. Moreover, in that case little optimizations can be expected if the line size is small because *intrinsic misses* are too numerous anyway. However, when line size is sufficiently large, then exploiting the potential locality of array X may yield significant improvements of the total hit ratio, especially in 3-dimensional *finite-element* problems where the potential reuse per element of X is important.

Little hardware or software techniques exist for making use of locality within sparse problems. First, *Bandwidth reduction renumbering algorithms* may significantly improve the exploitation of locality within SpMxV. Second, blocking methods are considered. Classic rectangular blocking proves to be efficient only when the matrix is nearly dense within its band, otherwise the method breeds too much overhead. A blocking technique *blocking by diagonal* that takes into account the specifics of sparse codes has been presented. It is shown to efficiently exploit locality where other blocking methods would fail, i.e when the matrix is "very sparse". Moreover, the parallel version of the diagonally blocked algorithm on a multi-cache system would naturally reduce the burden on local caches.

Acknowledgements We would like to thank J. Erhel and A. Seznec for their helpful comments and many enlightening discussions.

References

[1] S. G. Abraham and T. A. Davis: *Blocking for parallel sparse linear system solvers*, Proc. of Int. Conf. on Parallel Processing, 1989.

[2] F. Bodin, C. Eisenbeis, D. Windheiser, W. Jalby: *A strategy for array management in local memory*, Advances in Languages and Compilers for Parallel Processing, MIT Press, 1991.

[3] I. Duff, A. Erisman and J. K. Reid: *Direct methods for Sparse matrices*, Oxford University Press, Oxford, England, 1987.

[4] I. Duff, R. Grimes, J. Lewis: *Sparse matrix test problems*, ACM TOMS, 15 (1989), pp. 55-64.

[5] C. Fricker, P. Robert: *An analytical cache model*, INRIA Report, INRIA Rocquencourt, France, July 1991.

[6] A. George: *Direct solution of sparse positive definite systems: some basic ideas and open problems*, in Sparse Matrices and Their Uses, edited by I. S. Duff, Academic Press, 1981.

[7] R. N. Ibbet, T. M. Hopkins and K. I. M. McKinnon: *Architectural mechanisms to support sparse vector processing*, Proc. of ISCA, 1989.

[8] Monica S. Lam, Edward E. Rothberg and Michael E. Wolf: *The cache performance and optimizations of blocked algorithms*, Proc. of ASPLOS, 1991.

[9] E. Rothberg and A. Gupta: *Techniques for improving the performance of sparse matrix factorization on multiprocessor workstations*, Proc. of ACM Supercomputing 90.

[10] Y. Saad and H. A. G. Wijshoff: *SPARK: A benchmark package for sparse computations*, Proc. of Int. Conf. on Supercomputing, 1990.

[11] A. Seznec and Y. Jegou: *Synchronizing processors through memory requests in a tightly coupled multiprocessor*, Proc. of Int. Symp. on Computer Architecture, 1988.

[12] O. Temam, W. Jalby: *Characterizing the behavior of sparse algorithms on caches*, INRIA Report No. 1666, April 1992.

[13] H.(Harry) A.(Albert) G.(Geraldine) Wijshoff: *Implementing Sparse BLAS primitives on concurrent/vector processors: a case study* CSRD Report No. 843, CSRD, University of Illinois, 1989.

[14] A. Wolfe, M. Breternitz Jr., C. Stephens, A. L. Ting, D. B. Kirk, R. P. Bianchini Jr., J. P. Shen: *The White Dwarf: A high-performance application-specific processor*, Proc. of ISCA, 1988.