On Optimal Locality of Linear Relaxation

Claudia Leopold
Institut für Informatik
Friedrich-Schiller-Universität Jena
07740 Jena, Germany
email: claudia@informatik.uni-jena.de

ABSTRACT
Tiling is well-known to reduce the number of cache misses in linear relaxation codes. This paper investigates analytically how close to optimum the improvement gets. We consider one time step of the Jacobi and Gauss-Seidel methods on a two-dimensional array of size $(N+2) \times (N+2)$. For cache capacity C and line size L, we prove that at least $0.6N^2/(LC)$ capacity misses are taken, independent on the schedule of operations in the program. Furthermore, we show that tiled codes are off this bound by a factor of $4L$. Finally, we reduce the factor to 7 with a sophisticated data layout scheme.

KEY WORDS
High Performance Computing, Applications, Memory and I/O Systems, Optimization

1. Introduction

Linear relaxation codes such as the Jacobi and Gauss-Seidel kernels are among the most time-consuming routines in many scientific and engineering applications. The kernels form the core of classical iterative solvers for partial differential equations, and are used as building blocks in modern efficient schemes such as multigrid.

Linear relaxation kernels are frequently denoted as *stencil codes* because they update array elements according to some fixed pattern, called stencil. Stencil codes perform a sequence of sweeps through a large array, in each sweep updating all array elements except the boundary. This paper considers a two-dimensional array with five-point stencil, which updates the array elements on the basis of their own values and the values of the four immediate neighbors. The codes are given in Fig. 1.

As has been observed by, for instance, Douglas et al. [1], the performance of stencil codes lags far behind peak performance, chiefly because cache usage is poor. Nevertheless, stencil codes have a significant locality potential since successive accesses refer to neighbored array elements. In our 2D codes, four types of data reuse can be distinguished:

(1) Assuming column-major storage order, the cache line that holds $A(I,J), A(I{+}1,J), A(I{+}2,J)\ldots$ is reused in the I-loop.

(2) $A(I,J)$ is reused in the I-loop, in the updates of $A(I{-}1,J)$, $A(I,J)$, and $A(I{+}1,J)$.

(3) $A(I,J)$ is reused in the J-loop, in the updates of $A(I,J{-}1)$, $A(I,J)$, and $A(I,J{+}1)$.

(4) $A(I,J)$ is reused in the time loop.

Reuse types 2–4 correspond to temporal locality since the same data element is accessed; reuse type 1 corresponds to spatial locality since data from the same cache line are accessed. Reuse types 1 and 2 are exploited by the input codes, but reuse type 3 is not exploited if the array size N exceeds the cache capacity C. Then, $A(I,J)$ must be loaded anew for the updates of $A(I,J)$ and $A(I,J+1)$, despite having been loaded for the update of $A(I,J-1)$. We do not consider reuse type 4. This paper investigates reuse type 3 on the assumption that $N < C$. This assumption may hold for L1 caches. Moreover, the problem can be considered a simpler version of the cache optimization problem for stencil codes on 3D arrays with $N^2 < C$.

The technique for exploiting type 3 reuse is well-known: tiling. Closest related to ours is work by Rivera and Tseng who study tiling experimentally. Figure 2 depicts the tiled Gauss-Seidel code for the input of Fig. 1. The tiled Jacobi code is analogous, but has been omitted for brevity. Whereas previous research has experimentally shown that tiling improves performance, we investigate analytically how close to optimum tiling gets in terms of cache misses.

In particular, we prove a lower bound on the number of capacity misses for one sweep through the array in the 2D Gauss-Seidel and Jacobi kernels. Note that, in the Jacobi kernel, a sweep is different from a time step. Without loss of generality, we consider the first sweep, in which array A is read, and array B is written to.

In line with common notation, cache misses are classified as cold misses (the first access to an element), capacity misses (misses due to limited cache capacity), and conflict misses (misses due to cache thrashing). Obviously, the Gauss-Seidel and Jacobi kernels take about $(N+2)^2$ cold misses since all elements of A (except the corners) must be read. We do not consider conflict misses, but assume that padding [6] is applied after our optimizations to eliminate conflict misses.

We prove a lower bound of $\Omega(N^2/(LC))$, in which L denotes cache line size. For $C \leq N/10$, the constant

a) do T = 1, time ! or repeat until convergence
do J = 1, N
do I = 1, N
$A(I,J) = W1 * A(I,J) + W2 * A(I-1,J)$
$+W3 * A(I+1,J) + W4 * A(I,J-1)$
$+W5 * A(I,J+1)$ (*)
end do
end do
end do

b) do T = 1, time ! or repeat until convergence
do J = 1, N
do I = 1, N
$B(I,J) = W1 * A(I,J) + W2 * A(I-1,J)$
$+W3 * A(I+1,J) + W4 * A(I,J-1)$
$+W5 * A(I,J+1)$ (*)
end do
end do
do J = 1, N
do I = 1, N
$A(I,J) = W1 * B(I,J) + W2 * B(I-1,J)$
$+W3 * B(I+1,J) + W4 * B(I,J-1)$
$+W5 * B(I,J+1)$ (*)
end do
end do
end do

Figure 1. a) Gauss-Seidel and b) Jacobi kernel. A and B are supposed to be $[0 \ldots N+1] \times [0 \ldots N+1]$ arrays

do T = 1, time
do II = 1, N, S
do J = 1, N
do $I = II$, min(II+S-1, N)
$A(I,J) = W1 * A(I,J) + W2 * A(I-1,J)$
$+W3 * A(I+1,J) + W4 * A(I,J-1)$
$+W5 * A(I,J+1)$ (*)
end do
end do
end do
end do

Figure 2. Tiled Gauss-Seidel code

factor is estimated by 0.6. Furthermore, we analyze tiling and show that the tiled codes take $2N^2/C$ capacity misses. Thus the bounds are off by a factor of at most $4L$. In light of the $(N+2)^2 >> 2N^2/C$ cold misses that any Gauss-Seidel or Jacobi code takes, this result indicates very close-to-optimum cache performance for tiling.

In the second part of the paper, we improve the upper bound to $4N^2/(LC)$, using a sophisticated data layout scheme instead of column-major. The scheme's drawback is a high addressing expense. We also compare column-major and row-major storage order and observe that these are equivalent for tiled codes.

The rest of this paper is organized as follows. Section 2 proves the lower bound and estimates the constant factor. Section 3 analyzes tiling and discusses the improvement by L through data layout. Section 4 finishes with related work and conclusions.

2. Lower bound

We consider one sweep through the array, in either the Gauss-Seidel or Jacobi scheme. In a sweep, N^2 operations are carried out, which correspond to instances of statement (*) for $I = 1 \ldots N$, $J = 1 \ldots N$. We allow for redundancy, that is, operations may be repeated, but we do not allow to split statement (*) into subcomputations.

In the Gauss-Seidel scheme, data dependencies exist from iterations $(I-1, J)$ and $(I, J-1)$ to iteration (I, J). The proof ignores these dependencies, which is correct because a lower bound to an unconstrained problem is always a lower bound to a constrained problem, too. Similarly, no particular cache replacement scheme is relied upon. The lower bound refers to user-controlled data placement but, consequently, also holds for any common scheme such as LRU.

Theorem 1 *On the conditions stated above, any schedule of the N^2 operations takes $\Omega(N^2/(LC))$ capacity misses.*

Proof: Let $Sched$ be any schedule. We partition *Sched* into subsequences $S_1, S_2, \ldots S_K$ of successive operations such that $S_1, S_2, \ldots S_{K-1}$ consist of exactly C^2 operations, and S_K consists of up to $2C^2 - 1$ operations. Then,

$$K \geq \lfloor N^2/C^2 \rfloor \ ,$$

where inequality holds in case of redundancy.

Disregarding redundancy, each interior array element is accessed five times in $Sched$, and each boundary element is accessed one time. The following definition captures array elements that are accessed by multiple S_i.

Definition 2 *An array element e is* touched-only *by S_i ($1 \leq i \leq K$) if the number of accesses to e in S_i is at least one, but less than the total number of accesses to e in $Sched$.*

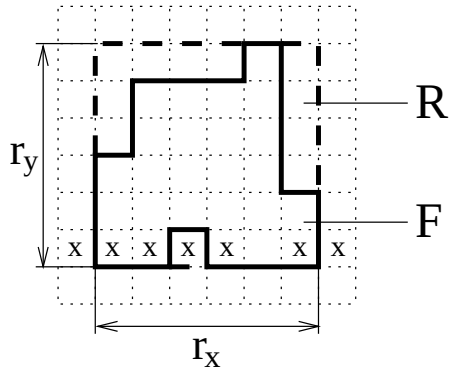

Figure 3. Geometric model of a subsequence

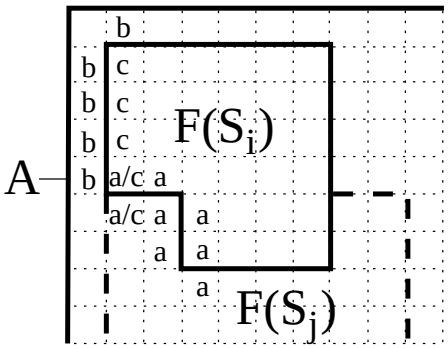

Figure 4. Correspondence between touched and touched-only. At the left boundary, touched squares are marked according to cases a), b) and c)

The rest of this proof uses geometric argumentation. We model A by a large square that is composed of small squares for the $A(I, J)$. Just think of A as being drawn on squared paper such that $A(I, J)$ corresponds to the printed square in the I-th row and J-th column of A.

In the following, we consider any particular S_i. Let $F = F(S_i)$ denote the geometric figure (also called arrangement) that is composed of those small squares whose array elements are updated in S_i. Furthermore, let R denote the smallest axes-parallel rectangle that completely holds F, and let r_x, r_y be the side lengths of R in x- and y-direction, respectively (see Fig. 3).

We next interpret the definition of 'touched-only' in our geometric model. We say that a (small) square is *x-touched* by F if

- the square belongs to F, but its left or right neighbor does not, or
- the square does not belong to F, but its left or right neighbor does.

The definition of *y-touched* is analogous, but refers to the neighbors above and below the square. We say that an array element is *touched* if it is x-touched or y-touched. Since an array element has four neighbours, it can be touched up to fourfold. To give an example, the x-touched squares in the bottom row of Fig. 3 have been marked by x. Most of these squares are y-touched, as well.

The update operation (*) refers to geometric neighbors. Consequently, if the square that corresponds to an array element e is touched by F, then at least one of the following conditions hold (see Fig. 4):

a) e is touched-only by S_i, or

b) e belongs to the boundary of A, or

c) e has a neighbor that belongs to the boundary of A.

Lemma 3 below shows that for any arrangement of C^2 squares, at least $4C$ squares are x-touched, or at least $4C$ squares are y-touched. Consequently, the whole schedule comprises at least

$$Z \geq 4KC$$

pairs (S_i, e) for which the square that corresponds to e is touched by the figure that corresponds to S_i. We distinguish these pairs into five types:

(1) e is touched-only by S_i, and one of the accesses to e in S_i is the first access to e in *Sched*

(2) e is touched-only by S_i, and one of the accesses to e in S_i is the last access to e in *Sched*

(3) e is touched-only by S_i, and none of the accesses to e in S_i is the first or last access to e in *Sched*

(4) e is not touched-only by S_i, and e belongs to the boundary of A

(5) e is not touched-only by S_i, and a neighbor of e belongs to the boundary of A

Altogether, $Sched$ comprises at most $8N$ pairs of types 4 and 5. We denote the total number of type 1, 2, and 3 pairs by Z^{in}, Z^{out}, and Z^{inOut}, respectively. From

$$Z^{in} + Z^{out} + Z^{inOut} = Z - 8N$$

and $Z^{in} = Z^{out}$ follows

$$Z^{out} + Z^{inOut} \geq Z/2 - 4N \ .$$

At most C data are kept in cache in-between successive S_i's, summing up to $W \leq (K-1) \cdot C$ elements for the whole schedule. Consequently, at least

$$\begin{aligned} Z^{out} + Z^{inOut} - W &\geq 2KC - 4N - kC + C \\ &\geq \lfloor N^2/C^2 \rfloor \cdot C - 4N + C \end{aligned}$$

elements must be reloaded after having been replaced from cache. Since at most L elements are loaded per memory access, there are at least

$$(1/L) \cdot \left(\lfloor N^2/C^2 \rfloor \cdot C - 4N + C \right) = \Omega(N^2/(LC))$$

capacity misses. ■

For $C \leq N/10$, the bound can more accurately be estimated by:

$$(1/L) \cdot \left(\lfloor N^2/C^2 \rfloor C - 4N + C\right)$$
$$\geq (1/L) \cdot (N^2/C - 4N) \geq 0.6N^2/(LC)$$

The proof of Lemma 3 is still open:

Lemma 3 *For any arrangement of Q squares, at least $4\sqrt{Q}$ squares are x-touched, or at least $4\sqrt{Q}$ squares are y-touched.*

Proof: The proof is by contradiction. Assume that F_0 arranges Q squares such that

$$\#x\text{-touched}(F_0) < 4\sqrt{Q} \quad \text{and}$$
$$\#y\text{-touched}(F_0) < 4\sqrt{Q}. \qquad \text{(P1)}$$

Then, an arrangement F_1 can be constructed that fulfills (P1) and has no empty rows between the lowermost and uppermost rows that contain squares. F_1 is constructed by removing all empty rows from F_0, which implies

$$\#x\text{-touched}(F_1) = \#x\text{-touched}(F_0)$$

and

$$\#y\text{-touched}(F_1) \leq \#y\text{-touched}(F_0) \ .$$

Analogously, an arrangement F_2 can be constructed that fulfills (P1) and has neither empty rows nor empty columns. We call such an arrangement *compact*.

Rotating F_2 if needed, we obtain a compact arrangement F_3 that fulfills (P1) and $r_x(F_3) \leq r_y(F_3)$. Simplifying (P1), we finally get an arrangement F with

- F is compact and consists of Q squares, and (P2)
- $r_x(F) \leq r_y(F)$, and (P3)
- $\#x\text{-touched}(F) < 4\sqrt{Q}$, and (P4)
- among the arrangements that fulfill (P2)–(P4), F minimizes $\#x$-touched (P5)

From now on, we refer to F, except where otherwise mentioned. Let w_i denote the number of squares in row i. Then, the number of squares in row i that are x-touched is at least

$$\#x\text{-touched}(i) \geq \begin{cases} 3 & w_i = 1 \\ 4 & w_i \geq 2 \end{cases} \qquad \text{(Q1)}$$

Let r_s denote the number of rows that contain only a single square, r_f the number of rows that contain r_y squares, and r_h the number of rows that contain more than one but less than r_y squares. Since $r_x \leq r_y$, the r_f-rows are 'full', and the r_h-rows are 'half-full'.

In the following, we distinguish seven cases, and derive a contradiction in each of them. For brevity, only three cases are presented in full, the rest is similar.

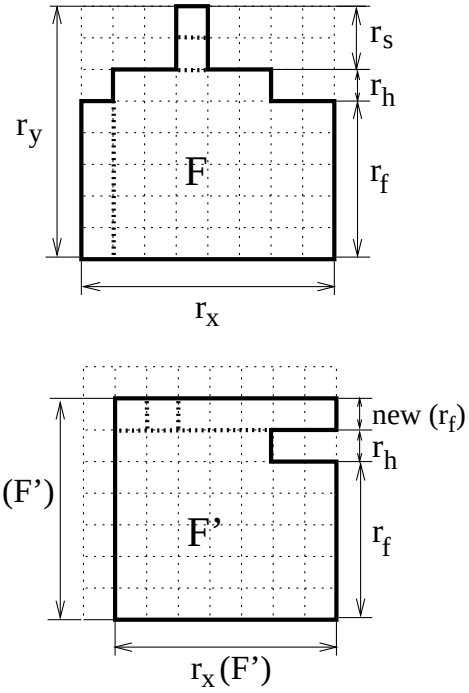

Figure 5. Rearrangement in Case 2

Case 1: $\mathbf{r_s = 0}$. Since $r_y^2 \geq r_x r_y \geq Q$, observation (Q1) implies

$$\#x\text{-touched} \geq 4r_y \geq 4\sqrt{Q} \ ,$$

in contradiction to (P4).

Case 2: $\mathbf{r_h > 0}$, $\mathbf{r_s \geq 2}$. We construct an arrangement F' by combining two r_s-rows into an r_h-row, and moving one square from each r_f-row to that row as well (see Fig. 5). The rearrangement must not insert any new gaps. Then, F' fulfills (P2)–(P4). For (P3), observe that the former r_f-rows fulfill

$$r_x(F') = r_x(F) - 1 \leq r_y(F) - 1 = r_y(F')$$

by construction, and the filled-up row contains at most

$$r_f(F) + 2 = r_y(F) - r_h(F) - r_s(F) + 2 \leq r_y(F) - 1$$

squares. On the other hand,

$$\begin{aligned} \#x\text{-touched}(F') &= 4(r_f(F) + r_h(F) + 1) + 3(r_s(F) - 2) \\ &= \#x\text{-touched}(F) - 2 \\ &< \#x\text{-touched}(F) \ , \end{aligned}$$

in contradiction to (P5).

Case 3: $\mathbf{r_h = 0}$, $\mathbf{r_s \geq 3}$. Similar to case 2.

Case 4: $\mathbf{r_h = 0}$, $\mathbf{r_s = 1}$. This case is depicted in

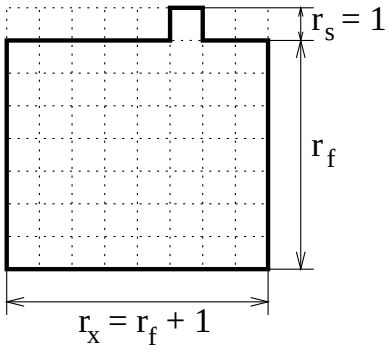

Figure 6. Shape of fi gure in Case 4

Fig. 6. Since

$$\begin{aligned} Q &= r_f(r_f+1)+1 \\ &= r_f^2 + r_f + 1 \\ &= (r_f+1)^2 - r_f \ , \end{aligned}$$

the value $\sqrt{Q}$ is about in the middle between r_f and r_{f+1}. For $r_f \geq 2$, a more accurate calculation yields $\sqrt{Q} \leq r_f + 0.7$. Thus,

$$\#x\text{-touched} \geq 4r_f + 3 \geq 4\sqrt{Q} - 2.8 + 3 > 4\sqrt{Q} \ ,$$

in contradiction to (P4).

Case 5: $\mathbf{r_h = 0}$, $\mathbf{r_s = 2}$. Similar to case 4.

Case 6: $\mathbf{r_h > 0}$, $\mathbf{r_s = 1}$, and

$$\mathbf{r_h(r_y - 1) - \sum_{r_h\text{-rows}} w_i \geq r_f + 1} \ .$$

The last inequality can be interpreted as that there are enough gaps in the r_h-rows to absorb a square from each of the r_f rows and from the r_s row. Similar to case 2.

Case 7: $\mathbf{r_h > 0}$, $\mathbf{r_s = 1}$, and

$$\mathbf{r_h(r_y - 1) - \sum_{r_h\text{-rows}} w_i < r_f + 1} \ .$$

Similar to case 4. ■

3. Upper bound

Figure 7 illustrates the execution order of the tiled code. Each horizontal stripe of S rows is called a tile. For exploitation of type 3 reuse, the cache must hold two columns plus one element of a tile. Hence, we choose $S = C/2$ (ignoring the one element for simplicity of analysis).

The tiled code exploits all cases of type 3 reuse, but it does not exploit type 1 reuse nor type 2 reuse at tile boundaries. Cache lines that are cut by tile boundaries must be

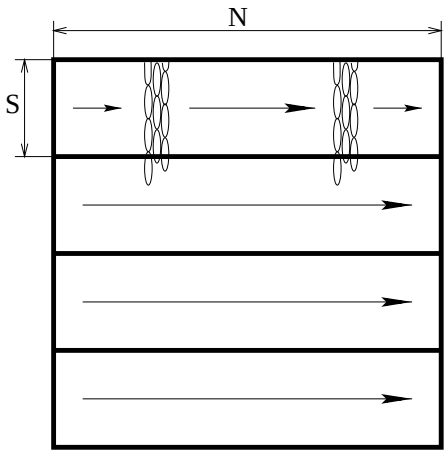

Figure 7. Execution order of tiled code

loaded twice; of the two loads, one is a cold miss, and the other is a capacity miss. It is of no help to let cache lines end at tile boundaries; in contrary, this induces four misses. If cache lines are cut, the total number of capacity misses equals the total length of tile boundaries, which can be estimated by

$$N \cdot (N/S) \leq 2N^2/C \ .$$

Note that this bound holds for both Jacobi (since the tiled code respects data dependencies) and Gauss-Seidel (assuming a write-around cache).

Hence, the upper and lower bounds differ by a factor of less than $4L$. As noted in the introduction, this difference is very low as compared to the number of cold misses. Moreover, it is partly due to simplifi cations in the lower bound proof. Thus, from a practical point of view, tiling is almost optimal.

Nevertheless, the L factor hurts if we think of possible future caches with a large line size. The factor can be eliminated by using a mixed column-row layout as sketched in Fig. 8. In this layout, data are stored column-wise except at tile boundaries, where two rows of data are stored row-wise. If line size does not evenly divide tile size, lines continue from the bottom of a tile's column J to the top of the same tile's column $J+1$.

In mixed column-row layout, the cache must hold two columns of a tile plus four horizontal blocks, that is $C \geq 2S + 4L$. A sophisticated padding scheme can guarantee that the data that are stored row-wise do not interfere with the data that are stored column-wise. Unfortunately, such a scheme has a high addressing expense which is likely to outweigh any gain in locality. Ignoring this problem for now, the layout incurs about

$$(N/S) \cdot N \cdot (2/L) = 4N^2/(LC)$$

capacity misses because, at tile boundaries, we have two cold misses and two capacity misses every L columns.

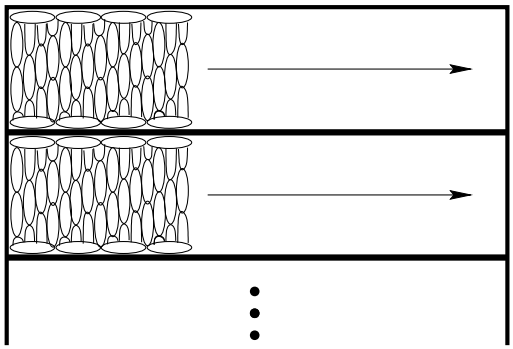

Figure 8. Mixed column-row layout

Hence, the layout reduces the distinction between upper and lower bounds to a factor of less than 7. Again, this is very low as cold misses have not been accounted for.

The advantage of row-wise layout at tile boundaries may suggest that a pure row-major order outperforms standard column-major order as well. However, this is not the case. With row-major order, the cache must hold at least $C \geq SL$ elements, and thus the number of capacity misses is $(N/S) \cdot N \cdot (2/L) = 2N^2/C$. Consequently, row-major order and column-major order are about equivalent.

4. Related work

The compiler optimization community has conducted much research on tiling [5, 9]. Nevertheless, Douglas et al. [1] observe that current production compilers are not able to tile even elementary stencil codes. Part of the reason may be differences in the way tiling works for stencil codes as opposed to dense linear algebra codes (the focus of compiler research). There, the data of a tile are moved into cache together, and then stay until the processing of the tile is complete. Tiled stencil codes, in contrast, keep only a few columns in cache to be efficient.

Several papers deal with tiling specifically for stencil codes. Rivera and Tseng [6] present tiled 3D codes that are analogous to our 2D codes. They also discuss data placement schemes that avoid conflict misses. Like us, they refer to a single sweep and consider updates as a unit. The usefulness of tiling is shown experimentally, lower bounds are not given. In other papers, tiling is used to exploit type 4 reuse [1–3, 7, 8].

The present paper extends our previous work [4], in which we have analyzed cache misses on the assumption $L = 1$. In [4], modifications of tiling were studied, such as a snake-like order of tile execution. These modifications can be applied to $L \neq 1$, as well. The present paper has given emphasis to data layout. Moreover, we have improved the constant factor in the lower bound. In particular, Lemma 3 replaces the weaker estimation of $3\sqrt{Q}$ touched squares that has been used in [4].

5. Conclusions

In summary, this paper has shown that tiled codes achieve close-to-optimal cache performance. Our results complement previous experimental studies, in which tiling was shown to speed up programs, but no comparison with optimal performance was given. We have proven upper and lower bounds that are off by a factor of $4L$, which is very low since the bounds refer to capacity misses, but cold misses dominate the overall missrate. For the case of large L, we have also suggested a mixed column-row data layout that reduces the factor to 7. In ongoing work, we are generalizing our results to 3D codes and tiling schemes for the time loop.

References

[1] C. C. Douglas, J. Hu, M. Kowarschik, U. Rüde, and C. Weiß. Cache optimization for structured and unstructured grid multigrid. *Electronic Transaction on Numerical Analysis*, 10:21–40, 2000.

[2] J.-W. Hong and H. T. Kung. I/O complexity : The red-blue pebble game. In *Proc. ACM Symposium on Theory of Computing*, pages 326–333, 1981.

[3] C. E. Leiserson, S. Rao, and S. Toledo. Efficient out-of-core algorithms for linear relaxation using blocking covers. *Journal of Computer and System Sciences*, 54(2):332–344, Apr. 1997.

[4] C. Leopold. On optimal temporal locality of stencil codes. To appear in *Proc. ACM Symp. on Applied Computing*, 2002.

[5] G. Rivera and C.-W. Tseng. A comparison of compiler tiling algorithms. In *8th Int. Conf. on Compiler Construction*, pages 168–182. LNCS 1575, 1999.

[6] G. Rivera and C.-W. Tseng. Tiling optimizations for 3D scientific computations. In *Proc. Supercomputing*. IEEE, 2000. Available at http://www.supercomp.org/sc2000/Proceedings/start.htm.

[7] Y. Song and Z. Li. New tiling techniques to improve cache temporal locality. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 215–228, 1999.

[8] C. Weiß, W. Karl, M. Kowarschik, and U. Rüde. Memory characteristics of iterative methods. In *Proc. Supercomputing SC'99*. Available in ACM Digital Library.

[9] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. *SIGPLAN Notices*, 26(6):30–44, 1991.