Tight Bounds on Capacity Misses for 3D Stencil Codes

Claudia Leopold

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

Abstract. The performance of linear relaxation codes strongly depends on an efficient usage of caches. This paper considers one time step of the Jacobi and Gauß-Seidel kernels on a 3D array, and shows that tiling reduces the number of capacity misses to almost optimum. In particular, we prove that $\Omega(N^3/(L\sqrt{C}))$ capacity misses are needed for array size $N \times N \times N$, cache size C, and line size L. If cold misses are taken into account, tiling is off the lower bound by a factor of about $1+5/\sqrt{LC}$. The exact value depends on tile size and data layout. We show analytically that rectangular tiles of shape $(N-2) \times s \times (sL/2)$ outperform square tiles, for row-major storage order.

1 Introduction

Stencil codes such as the Jacobi and Gauß-Seidel kernels are used in many scientific and engineering applications, in particular in multigrid-based solvers for partial differential equations. Stencil codes own their name to the fact that they update array elements according to some fixed pattern, called stencil. The codes perform a sequence of sweeps through a given array, and in each sweep update all array elements except the boundary. This paper considers the Gauß-Seidel and Jacobi kernels on a 3D array, which are depicted in Fig. 1. Since each update operation (*) accesses seven array elements, we speak of a 7-point stencil.

In Fig. 1b), the copy operation can alternatively be replaced by a second loop nest in which the roles of arrays A and B are reversed. Throughout the paper, we consider a single iteration of the time loop. Copy operation or second loop nest are not taken into account. Obviously, the Gauß-Seidel scheme incurs data dependencies from iterations $(i-1,j,k)$, $(i,j-1,k)$, and $(i,j,k-1)$ to iteration (i,j,k), whereas the Jacobi scheme does not incur data dependencies.

When implemented in a straightforward way, the performance of stencil codes lags far behind peak performance, chiefly because cache usage is poor [1]. On the other hand, stencil codes have a significant cache optimization potential since successive accesses refer to neighbored array elements. In 3D codes, five types of data reuse can be distinguished:

(1) Assuming row-major storage order, the cache line that holds elements $A[i,j,k]$, $A[i,j,k+1]\ldots$ is reused in the k-loop,

```
a) for (t=0; t<time; t++)     /* or: while (!converged) */
     for (i=1; i<N-1; i++)
       for (j=1; j<N-1; j++)
         for (k=1; k<N-1; k++)
           A[i,j,k] += A[i-1,j,k] + A[i+1,j,k]
                     + A[i,j-1,k] + A[i,j+1,k]
                     + A[i,j,k-1] + A[i,j,k+1];        (*)

b) for (t=0; t<time; t++) {   /* or: while (!converged) */
     for (i=1; i<N-1; i++)
       for (j=1; j<N-1; j++)
         for (k=1; k<N-1; k++)
           B[i,j,k] = A[i,j,k] + A[i-1,j,k] + A[i+1,j,k]
                    + A[i,j-1,k] + A[i,j+1,k]
                    + A[i,j,k-1] + A[i,j,k+1];        (*)
     Copy(B->A);
   }
```

Fig. 1. a) Gauß-Seidel and b) Jacobi code for $N \times N \times N$ arrays A, B

(2) $A[i,j,k]$ is reused in the k-loop, in the updates of $A[i,j,k-1]$, $A[i,j,k]$, and $A[i,j,k+1]$,
(3) $A[i,j,k]$ is reused in the j-loop,
(4) $A[i,j,k]$ is reused in the i-loop, and
(5) $A[i,j,k]$ is reused in the time loop.

Reuse is exploited if data are kept in cache in-between successive accesses; exploited reuse is denoted as *locality*. Reuse types 1 and 2 are always exploited by the input codes; whether reuse types 3-5 are exploited depends on array size N and cache capacity C. As explained by Rivera and Tseng [7], reuse type 3 is exploited for $C > 2N$. This condition often holds in practice, and thus many 2D codes perform well without cache optimization. The exploitation of type 4 reuse requires $C > 2N^2$, which does not hold for typical cache and array sizes. Tiling can transform programs such that type 4 reuse is exploited, too. We will study this technique in the paper. Type 5 reuse is out of our scope.

Tiling rearranges the updates such that successive type 4 reuses are moved closer together. Fig. 2 depicts a tiled Gauß-Seidel code; tiled Jacobi code is analogous and has been omitted for brevity. Tiling groups the updates into rectangular blocks (called tiles) that are processed one after another. Note that only two loops are blocked, and the third is not. Why it is better to block the j- and k-loops instead of the i-loop will be explained in Sect. 3.

Tiling speeds up 3D stencil codes, as has been shown experimentally by Rivera and Tseng [7]. The present paper complements their work by analytically investigating the question as to how close to optimum tiling gets in terms of cache misses. In line with common notation, we distinguish cache misses into cold misses (the first access to an element), capacity misses (misses due to limited

```
for (t=0; t<time; t++)
  for (jj=1; jj<N-1; jj+=sj)
    for (kk=1; kk<N-1; kk+=sk)
      for (i=1; i<N-1; i++)
        for (j=jj; j<min(jj+sj,N-1); j++)
          for (k=kk; k<min(kk+sk,N-1); k++)
            A[i,j,k] += A[i-1,j,k] + A[i+1,j,k]
                      + A[i,j-1,k] + A[i,j+1,k]
                      + A[i,j,k-1] + A[i,j,k+1];    (*)
```

Fig. 2. Tiled Gauß-Seidel code

cache capacity), and conflict misses (misses due to limited cache associativity). Conflict misses can be eliminated through tile size selection, array padding, and copying [6, 7], and are not considered in this paper.

This paper makes three contributions. First, we prove that $\Omega(N^3/(L\sqrt{C}))$ capacity misses are needed for any ordering of the update operations (*), where L denotes cache line size. Second, we estimate the constant factor in this bound by 0.35 under the assumptions $N \geq 1000$, $C \geq 10000$ and $N^2 \geq 100C$. Third, we analyze the tiled codes for different data layouts and tile sizes. While Rivera and Tseng [7] use square tiles, we argue that rectangular tiles are superior. For these, we prove that the number of cold and capacity misses is off the lower bound by a factor of at most $1 + 4.6/\sqrt{LC}$.

The paper is organized as follows. Section 2 proves the lower bound and estimates the constant factor. Section 3 analyzes the tiled codes. Section 4 surveys related work, and Section 5 finishes with conclusions.

2 Lower Bound

The number of cold misses is trivially equal to $N^3 - 8$ since all elements of A, except the corners, must be read. This number is independent on the order of updates, and is, thus, valid for both the input and tiled codes. In the following, we consider capacity misses only.

We do not rely on a particular cache replacement scheme, but assume user-controlled data placement. The so-derived lower bound applies to any common scheme such as LRU. Similarly, we do not take data dependencies into account, which is correct because a lower bound to an unconstrained problem is always a lower bound to a constrained problem, too. Accesses to B are ignored as they can only increase the number of cache misses. Consequently, the lower bound holds for both the Jacobi and Gauß-Seidel schemes.

An ordering of the updates is denoted as *schedule*. We allow for redundancy, that is, updates may be carried out repeatedly with same indices i, j, k. Updates must not be split into subcomputations, however, to guarantee bitwise-identical results as compared to the input codes.

Theorem 1. *Any schedule of the update operations (*) takes* $\Omega(N^3/(L\sqrt{C}))$ *capacity misses.*

Proof: Let *Sched* be any schedule. We partition *Sched* into subsequences $S_1, S_2, \ldots, S_r$ of successive updates such that $S_1, S_2, \ldots S_{r-1}$ contain exactly $C\sqrt{C}$ updates, and S_r contains up to $2C\sqrt{C}$ updates. Note that

$$r \geq \lfloor (N-2)^3/(C\sqrt{C}) \rfloor \ ,$$

where inequality indicates redundancy.

We define an array element e to be *touched-only* by S_l $(1 \leq l \leq r)$ if the number of accesses to e in S_l is at least one, but less than the total number of accesses to e in *Sched*. In other words, an array element is touched-only if it is accessed by multiple S_l.

The rest of this proof uses geometric argumentation. We model A by a large cube that is composed of small unit cubes for the $A[i,j,k]$. Hence, $A[i,j,k]$ corresponds to a cube of side length 1×1×1 at position (i,j,k) of the large cube.

In the following, we consider any particular S_l. Let F denote the geometric figure (also called arrangement) that corresponds to S_l, that is, the figure that is composed of those unit cubes whose array elements are updated in S_l. Furthermore, let Q be the smallest axes-parallel cuboid that completely holds F. Obviously, the dimensions q_i, q_j, and q_k of Q (in i-, j-, and k-direction, respectively) fulfill $q_i \cdot q_j \cdot q_k \geq C\sqrt{C}$. We define a unit cube to be *i-touched* by F if

- the cube belongs to F, but its left or right neighbor in i-direction does not, or
- the cube does not belong to F, but its left or right neighbor in i-direction does.

Analogously, we define *j-touched* and *k-touched*. We say that a cube is *touched* if it is touched in at least one of the three directions. Since an array element has up to 6 neighbors, a cube can be touched up to sixfold.

The concepts of *touched* and *touched-only* are related. Consider, for instance, a cube inside F that is i-touched by F at the left boundary. Then the corresponding array element $A[i,j,k]$ is updated in S_l, but the left neighbor $A[i-1,j,k]$ is accessed and not updated. Consequently, $A[i-1,j,k]$ is either updated in another subsequence $S_{l'}$ with $l' \neq l$, or $A[i-1,j,k]$ is not updated at all. In the former case, both S_l and $S_{l'}$ access $A[i,j,k]$, and thus $A[i,j,k]$ is touched-only by S_l. In the latter case, $A[i-1,j,k]$ belongs to the boundary of A.

Lemma 1 below shows that any arrangement of $C\sqrt{C}$ cubes induces at least $4C - 6\sqrt{C}$ touched cubes. Consequently, within the whole schedule, the number of pairs (S_l, e) for which the cube that corresponds to e is touched by the figure that corresponds to S_l is at least

$$Z \geq r \cdot (4C - 6\sqrt{C}) \ .$$

We distinguish these pairs into five types:

(1) e is touched-only by S_l, and S_l is the first subsequence that accesses e.
(2) e is touched-only by S_l, and S_l is the last subsequence that accesses e.
(3) e is touched-only by S_l, and S_l is neither the first nor the last subsequence that accesses e.
(4) e is not touched-only by S_l, and e belongs to the boundary of A.
(5) e is not touched-only by S_l, and a neighbor of e belongs to the boundary of A.

The whole schedule comprises at most $6N^2+6(N-2)^2 = 12N^2-24N+24$ pairs of types 4 and 5. Let Z^{in}, Z^{out}, and Z^{inOut} denote the total number of type 1, 2, and 3 pairs, respectively. Then, $Z^{in} + Z^{out} + Z^{inOut} = Z - 12N^2 + 24N - 24$ and $Z^{in} = Z^{out}$ imply $Z^{out} + Z^{inOut} \geq Z/2 - 6N^2 + 12N - 12$.

In-between successive S_l's, at most C data are kept in cache, summing up to $W \leq (r-1) \cdot C$ data in the whole schedule. Consequently, the number of data that must be reloaded after having been replaced from cache is at least

$$\begin{aligned} Z^{out} + Z^{inOut} - W &\geq r \cdot (2C - 3\sqrt{C}) - 6N^2 + 12N - 12 - rC + C \\ &\geq \left\lfloor (N-2)^3/(C\sqrt{C}) \right\rfloor \cdot (C - 3\sqrt{C}) - 6N^2 + 12N - 12 + C \ . \end{aligned}$$

Since at most L elements are loaded per memory access, the number of capacity misses is at least

$$\begin{aligned} &(1/L) \cdot \left(\left\lfloor (N-2)^3/(C\sqrt{C}) \right\rfloor \cdot (C - 3\sqrt{C}) - 6N^2 + 12N - 12 + C \right) \\ &\qquad = \Omega(N^3/(L\sqrt{C})) \ , \end{aligned}$$

provided that N is significantly larger than $\sqrt{C}$. ■

In the following, we estimate the constant factor in this bound under the assumptions $N \geq 1000$, $C \geq 10000$ and $N^2 \geq 100C$. Imposing stronger assumptions increases the constant, and imposing weaker assumptions decreases it. With $X \underset{D}{=} (N-2)^3/(C\sqrt{C})$, we get

$$\begin{aligned} X &= \left(N^2(N-6) + 12N - 8\right) / (C\sqrt{C}) \\ &\geq \left(100C(10\sqrt{C} - 6) + 120\sqrt{C} - 8\right) / (C\sqrt{C}) \\ &= 1000 - 600/\sqrt{C} + 120/C - 8/(C\sqrt{C}) \\ &\geq 994 \ . \end{aligned}$$

Consequently, $\lfloor X \rfloor \geq X - 1 \geq (1 - 1/994)\ X$. Let $Y \underset{D}{=} \lfloor X \rfloor \cdot (C - 3\sqrt{C})$ and $Z \underset{D}{=} 6N^2 - 12N + 12 - C$. Then, $C - 3\sqrt{C} \geq 0.97C$ implies

$$Y \geq (1 - 1/994) \cdot 0.97 \cdot (N-2)^3/\sqrt{C} \geq 0.963\, N^3/\sqrt{C} \geq 9.63N^2 \ .$$

From $Z < 6N^2$ follows $Z \leq 0.63\, Y$, and thus $Y - Z \geq 0.37\, Y$. Putting it all together, the number of capacity misses is at least

$$(1/L) \cdot (Y - Z) \geq (0.37/L) \cdot 0.963 \cdot (N^3/\sqrt{C}) \geq 0.35N^3/(L\sqrt{C}) \ .$$

The proof of Lemma 1 is still open:

Lemma 1. *For any arrangement of $C\sqrt{C}$ cubes, at least $4C - 6\sqrt{C}$ cubes are touched.*

Proof: The proof is by contradiction. Let F be an arrangement with less than $4C - 6\sqrt{C}$ touched cubes. Let *column* denote a set of cubes in F that have the same i- and j-coordinates, and let *plane* denote a set of cubes in F that have the same k-coordinates. Furthermore, let w_p denote the number of cubes in plane p. We distinguish four cases and derive a contradiction in each of them.

Case 1: At least two planes p, p′ ($\mathbf{p \neq p'}$) exist such that $\mathbf{w_p \geq C}$ and $\mathbf{w_{p'} \geq C}$: The total number of k-touched cubes in F equals the sum of k-touched cubes in the individual columns (columns are independent wrt. touches in k-direction). In a column with one cube, at least three cubes are k-touched; in a column with two or more cubes, at least four cubes are k-touched. Hence, if p and p' have x columns in common, the total number of k-touched cubes is at least $4x + 3(\#\text{touched}(p) - x) + 3(\#\text{touched}(p') - x) \geq 4C$, a contradiction.

Case 2: A plane p exists with $\mathbf{w_p \geq 4C/3}$: Since there are at least $4C/3$ non-empty columns, the total number of k-touched cubes is at least $3 \cdot (4C/3) = 4C$, a contradiction.

Case 3: Exactly one plane p contains $\mathbf{C \leq w_p < 4C/3}$ cubes; all other planes contain less than C cubes: Obviously, the total number of i/j-touched cubes equals the sum of i/j-touched cubes in the individual planes (planes are independent wrt. touches in i/j-direction). In [3], it is shown that any 2D arrangement of Q unit squares induces at least $4\sqrt{Q}$ i/j-touched squares. Thus, in 3D, any plane of Q cubes i/j-touches at least $4\sqrt{Q}$ cubes. Hence, F altogether i/j-touches at least $4\sqrt{C} + \sum_p (4\sqrt{w_p})$ cubes, with $\sum_p w_p \geq C\sqrt{C} - 4C/3 \underset{D}{=} D$. We assume $\sum_p w_p = D$, which can only underestimate the number of touched cubes. Lemma 2 below shows that, for any fixed value of $\sum_p w_p$, the value of $\sum_p \sqrt{w_p}$ is maximized if the w_p take extreme values, that is, if $\lfloor D/(C-1) \rfloor$ of the w_p take value $C-1$, one w_p takes value $D - \lfloor D/(C-1) \rfloor \cdot (C-1)$, and the other w_p take value zero. Thus, the total number of i/j-touched cubes is at least

$$4\sqrt{C} + 4 \cdot \lfloor D/(C-1) \rfloor \cdot \sqrt{C-1} \geq 4D/\sqrt{C} > 4C - 6\sqrt{C} \ ,$$

which contradicts the assumption.

Case 4: All planes contain less than C cubes: In analogy to Case 3, the number of i/j-touched cubes can be estimated by

$$4 \left\lfloor C\sqrt{C}/(C-1) \right\rfloor \cdot \sqrt{C-1} \geq 4C - 4\sqrt{C-1} > 4C - 6\sqrt{C} \ ,$$

which contradicts the assumption. ■

Lemma 2 completes the lower bound proof:

Lemma 2. *For fixed values $d, e \in \mathbb{N}$, a finite set $X \subseteq \mathbb{N}$ with $\sum_{x \in X} x = d$ and $\forall x \in X : x \leq e$ minimizes $\sum_{x \in X} \sqrt{x}$ if*

– $\lfloor d/e \rfloor$ *elements of* X *have value* e,
– *one element of* X *has value* $d - \lfloor d/e \rfloor \cdot e$, *and*
– *all other elements of* X *have value zero.*

Proof: The proof is by contradiction. Let X be defined as to the items above, and let $Y \neq X$ with $\sum_{y \in Y} y = d$ and $\forall y \in Y : y \leq e$ minimize $\sum_{y \in set} \sqrt{y}$ for all finite sets with these properties. Since $Y \neq X$, elements $y_a, y_b \in Y$ exist such that $0 < y_a, y_b < e$. Let y_a be the smallest and y_b be the largest element with this property. We define $Y' \underset{D}{=} (Y \setminus \{y_a, y_b\}) \cup \{y_a - 1, y_b + 1\}$. Then, $\sum_{y' \in Y'} y' = d$ and $\forall y' \in Y' : y' \leq e$. Since for function $f(y) = \sqrt{y}$, the derivative $f'(y) = 1/(2\sqrt{y})$ is monotonely decreasing, we observe $\sqrt{y_a - 1} < \sqrt{y_a} - 1/(2\sqrt{y_a})$ and $\sqrt{y_b + 1} < \sqrt{y_b} + 1/(2\sqrt{y_b})$, which yields $\sqrt{y_a - 1} + \sqrt{y_b + 1} < \sqrt{y_a} + \sqrt{y_b}$. Thus,

$$\sum_{y' \in Y'} \sqrt{y'} = \sum_{y \in Y, y \neq y_a, y_b} \sqrt{y} + \sqrt{y_a - 1} + \sqrt{y_b + 1} < \sum_{y \in Y} \sqrt{y} \ ,$$

in contradiction to the minimality of Y. ■

3 Upper Bound

The tiled codes in Fig. 2 partition the updates into blocks of size $(N-2) \times s_j \times s_k$. Since the i-loop is not blocked, we say that the tiles proceed in i-direction. Alternatively, tiling can block the i- and k-loops so that tiles proceed in j-direction, or the i- and j-loops so that tiles proceed in k-direction. In either case, s_i, s_j, and s_k denote the tile size in directions i, j, and k. Below, we analyze the number of capacity misses assuming row-major storage order. For column-major storage order, the argumentation is analogous, except that i- and k-directions are interchanged.

The following analysis applies to both the Jacobi and Gauß-Seidel codes since tiling preserves the data dependencies of Gauß-Seidel, and since accesses to array B in the Jacobi code do not incur cache misses if a write-through cache (cache-bypassing) is used.

Tiling in k-direction Updates are carried out subplane-by-subplane, for increasing values of k. Here, similar to Sect. 2, a subplane is a set of array elements with same k index that are accessed in the present tile. Since an update refers to data from at most three subplanes, only three subplanes must be kept in cache instead of the whole tile [7].

Nevertheless, since cache line direction equals tile direction, it is not possible to keep exactly three planes in cache. Instead, the cache must hold up to $2L(s_i + 2)(s_j + 2)$ data, with factor 2 reflecting the possibility that $A[i, j, k]$ and $A[i, j, k-1]$ belong to different cache lines.

Capacity misses occur at tile boundaries only. If the updates of $A[i, j, k]$ and $A[i+1, j, k]$ (analogously of $A[i, j, k]$ and $A[i, j+1, k]$) belong to different tiles,

then both the cache line of $A[i,j,k]$ and the cache line of $A[i+1,j,k]$ must be loaded twice. Of the four misses, two are capacity misses. Since data from the same cache line are reused in successive planes, the two misses occur only once during the processing of L planes, so that the total number of capacity misses can be estimated by

$$(2/L) \cdot \text{total area of tile boundaries} \leq (2/L) \cdot (N^3/s_i + N^3/s_j) \ .$$

This bound is minimized for $s_i = s_j = \lfloor \sqrt{C/(2L)} \rfloor - 2$, which implies about $(2/L) \cdot N^3 \cdot 2 \cdot (\sqrt{2L}/\sqrt{C}) = 5.7 \cdot (N^3/\sqrt{LC})$ capacity misses. Taking cold misses into account, this value is off the lower bound by a factor of at most

$$\frac{N^3 - 8 + 5.7N^3/\sqrt{LC}}{N^3 - 8 + 0.35N^3/\sqrt{LC}} = 1 + \frac{5.35N^3/\sqrt{LC}}{N^3 - 8 + 0.35N^3/\sqrt{LC}} < 1 + 5.4/\sqrt{LC} \ .$$

Tiling in i- or j-direction We refer to i-direction; tiling in j-direction is analogous. In both cases, the cache lines proceed orthogonal to the tiles, and thus are cut by tile boundaries.

Again, the cache must hold about three subplanes of A (here: set of array elements with same i index), although the exact value is somewhat different. On one hand, it is sufficient that the cache holds about two subplanes instead of three since after updating $A[i,j,k]$, element $A[i-1,j,k]$ can be removed, and before updating $A[i,j,k]$, element $A[i+1,j,k]$ is not yet needed. On the other hand, cache lines must be stored completely, even if they are cut by tile boundaries and part of the elements is not accessed in the present tile. Hence, $C > rs_j s_k$, for some value r with $2 < r < 3$. Below we use $r = 3$, which somewhat overestimates the real cost (for reasonably large s_j, s_k).

Capacity misses occur at tile boundaries. We distinguish three cases, which are depicted in Fig. 3. The figure shows a plane of A for any fixed i. Referring to the figure, we denote tile boundaries with fixed j ($1 \leq i,k \leq N-2$) as horizontal, and tile boundaries with fixed k ($1 \leq i,j \leq N-2$) as vertical.

Cache lines along horizontal tile boundaries (such as lines a and b) must be loaded twice during the overall computation; after loading, they are used in L iterations of the k-loop. Of the four misses, two are capacity misses. Thus, the total number of capacity misses at horizontal tile boundaries is $(2/L) \cdot N^3/s_j$.

At vertical boundaries, cache lines that are cut (such as c) incur one capacity miss since each line must be loaded twice. Cache lines that end at tile boundaries (such as d and e) incur one capacity miss each, since both lines must be loaded twice. Hence, the case that cache lines are cut induces less misses than the case that cache lines end at tile boundaries. We assume here that all cache lines are cut, which can be achieved by padding the array. Then, N^3/s_k capacity misses are taken at vertical tile boundaries.

Adding the values of horizontal and vertical boundaries, we get a total of

$$(2/L) \cdot N^3/s_j + N^3/s_k$$

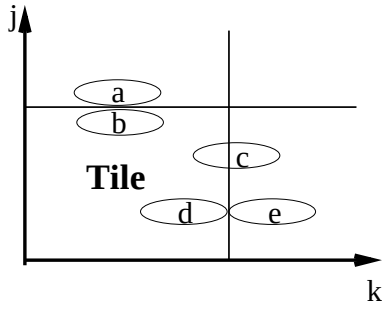

Fig. 3. Cache misses at tile boundaries

capacity misses. For $s_j = s_k \approx \sqrt{C/3}$, this value equals $\sqrt{3}(1+2/L) \cdot (N^3/\sqrt{C})$.

While Rivera and Tseng restrict consideration to square tiles [7], the following analysis suggests that rectangular tiles ($s_j \neq s_k$) perform better. Substituting $s_j = C/(rs_k)$ into the above formula yields

$$f(s_k) = (2rs_kN^3)/(LC) + N^3/s_k$$

capacity misses. This function is minimized for

$$f'(s_k) = (2rN^3)/(LC) - N^3/s_k^2 = 0 \ ,$$

that is, for $s_k = \sqrt{(LC)/(2r)}$ and $s_j = \sqrt{(2C)/(rL)}$. Here, $s_k = (L/2) \cdot s_j$, that is, rectangular tiles of shape $(N-2) \times s \times (sL/2)$ minimize the number of capacity misses. For these tiles, $2\sqrt{2r} \cdot (N^3/\sqrt{LC}) \approx 4.9 \cdot (N^3/\sqrt{LC})$ capacity misses are taken, which is slightly less than the number of capacity misses for tiling in k-direction, and less than the number of capacity misses for square tiles. Taking cold misses into account, i-direction tiling is off the lower bound by a factor of at most $1 + 4.6/\sqrt{LC}$, which is very close to one.

4 Related Work

Tiling is a well-known compiler optimization [6, 10]. Nevertheless, Douglas et al. [1] observe that current production compilers are not able to tile stencil codes. The problem is partly due to differences in the way tiling works for stencil codes as opposed to dense linear algebra codes (the focus of compiler research). In tiled stencil codes, the data sets of neighbored tiles overlap, whereas in dense linear algebra codes, the data sets are disjoint.

Several papers deal with tiling specifically for stencil codes. Closest to ours is work by Rivera and Tseng [7] who, like us, consider one iteration of the time loop for 3D arrays. Tiling is studied experimentally, including a discussion of tile size selection and array padding to reduce conflict misses. Rivera and Tseng observe that tiling speeds up 3D stencil codes by $17 - 121\%$, but they do not

relate their result to a lower bound. Other papers investigate tiling schemes for the time loop, that is, for the exploitation of type 5 reuse [1, 2, 5, 8, 9].

Part of the proof techniques in this paper have originally been developed for the simpler case of 2D stencils with $2N > C$ in [3, 4]. These references also discuss minor modifications of tiling, such as a snake-like order of tile evaluation, and a sophisticated data layout scheme that eliminates the L factor from the quotient between upper and lower bounds. The modifications can be generalized to 3D, but have little impact on running time.

5 Conclusions

This paper has analyzed the numbers of cold and capacity misses in one time step of the Jacobi and Gauß-Seidel kernels on 3D arrays. We have proven a lower bound, and have shown that the well-known technique of tiling gets very close to this bound. Moreover, we have compared different tiling schemes, and observed that tiles of shape $(N{-}2) \times s \times (sL/2)$ perform best. Future work should investigate this claim experimentally, including padding.

References

1. C. C. Douglas, U. Rüde, J. Hu, and M. Bittencourt. A guide to designing cache aware multigrid algorithms. In *Concepts of Numerical Software, Notes on Numerical Fluid Mechanics*. Vieweg-Verlag, 2001. To appear.
2. 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.
3. C. Leopold. On optimal locality of linear relaxation. To appear in Proc. IASTED Int. Multi-Conf. on Applied Informatics, 2002.
4. C. Leopold. On optimal temporal locality of stencil codes. To appear in Proc. ACM Symp. on Applied Computing, 2002.
5. C. Leopold. An analytical evaluation of tiling for stencil codes with time loop. To appear in Workshop-Proc. of Int. Parallel and Distributed Processing Symp. (4th Workshop on Advances in Parallel and Distributed Computational Models), 2002.
6. 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.
7. 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.
8. S. Sellappa. Cache-efficient multigrid algorithms. Master's thesis, University of North Carolina at Chapel Hill, Dept. of Computer Science, 2000.
9. Y. Song and Z. Li. New tiling techniques to improve cache temporal locality. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 215–228, 1999.
10. M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. *SIGPLAN Notices*, 26(6):30–44, 1991.