Data Space Oriented Tiling

Mahmut Kandemir

Department of Computer Science and Engineering
The Pennsylvania State University
University Park, PA 16802, USA
kandemir@cse.psu.edu

Abstract. An optimizing compiler can play an important role in enhancing data locality in array-intensive applications with regular data access patterns. This paper presents a compiler-based data space oriented tiling approach (DST). In this strategy, the data space (i.e., the array index space) is logically divided into chunks (called data tiles) and each data tile is processed in turn. In processing a data tile, our approach traverses the entire iteration space of all nests in the code and executes all iterations (potentially coming from different nests) that access the data tile being processed. In doing so, it also takes data dependences into account. Since a data space is common across all nests that access it, DST can potentially achieve better results than traditional tiling by exploiting inter-nest data locality. This paper also shows how data space oriented tiling can be used for improving the performance of software-managed scratch pad memories.

1 Introduction

Iteration space tiling (also called loop blocking) [9,1] is a loop-oriented optimization aiming at improving data locality. The idea behind tiling is to divide a given iteration space into chunks such that the data elements accessed by a given chunk fit in the available cache memory capacity. Previously-published work iteration space tiling reports significant improvements in cache miss rates and program execution times. Compilers use iteration space tiling mainly to create the blocked version of a given nested loop automatically. Note that, in general, it is difficult to guarantee that the array elements accessed by a given iteration space tile will fit in the cache. This problem occurs because tile shapes and tiling style are decided based on loop behavior rather than the data elements accessed. In particular, most of the current approaches to tiling do not consider the shape of the data regions (from different arrays) touched by an iteration space tile.

In this paper, we discuss and evaluate data space oriented tiling (DST), a variant of classical iteration space oriented tiling, to achieve better data locality than classical tiling. Instead of tiling iteration space first, and then considering data space requirements of the resulting tiles (data regions) in data space, DST takes a data space oriented approach. Specifically, it first logically divides data space into tiles (called data space tiles or data tiles for short), and then processes

D. Le Métayer (Ed.): ESOP 2002, LNCS 2305, pp. 178–193, 2002.

each data tile in sequence. Processing a data tile involves determining the set of loop iterations that access the elements in that data tile and executing these iterations taking into account data dependences. Since it starts its analysis from data space, DST has two main advantages over iteration space tiling:

- Since data space of a given array is shared across all nests that access the corresponding array, DST has a more global view of the program-wide access pattern (than iteration space tiling). This is especially true if one can come up with strategies to summarize access patterns of multiple nests on a given array.
- Working on data space allows compiler to take layout constraints into account better. For instance, in selecting a data tile shape, in addition to other parameters, the compiler can also consider memory layout of the array in question.

This paper describes a data space oriented tiling approach and presents a strategy for determining data tile shapes and sizes automatically. It also shows how DST can be used in conjunction with a scratch pad memory (SPM).

The remainder of this paper is organized as follows. Section 2 revises classical iteration space tiling and discusses how it improves data locality. Section 3 presents description of data space oriented tiling, focusing in particular on issues such as selection of data tile shapes, traversing iteration space, and handling data dependences. Section 4 discusses an application of DST to optimizing the effectiveness of scratch pad memories, which are compiler-managed on-chip SRAMs. Section 5 concludes the paper with a summary of our major contributions.

2 Review of Iteration Space Tiling

An important technique used to improve cache performance (data locality) by making better use of cache lines is iteration space tiling (also called loop blocking) [1,2,9]. In tiling, data structures that are too big to fit in the cache are (logically) broken up into smaller pieces that will fit in the cache. In other words, instead of operating on entire columns or rows of a given array, tiling enables operations on multi-dimensional sections of arrays at one time. The objective here is to keep the active sections of the arrays in faster levels of memory hierarchy (e.g., data caches) as long as possible so that when a data item (array element) is reused, it can be accessed from the faster memory instead of the slower memory.

For an illustration of tiling, consider the matrix-multiply code given in Figure 1(a). Let us assume that the layouts of all the arrays are row-major. It is easy to see that, from the cache locality perspective, this loop nest may not exhibit a very good performance (depending on the actual array sizes and cache capacity). The reason is that, although array U_2 has temporal reuse in the innermost loop (the k loop) and successive iterations of this loop access consecutive elements from array U_0 (i.e., array U_0 has spatial reuse in the innermost loop), the successive accesses to array U_1 touch different rows of this array. Obviously, this is not a good style of access for a row-major array. Using state-of-the-art

optimizing compiler technology (e.g., [5]), we can derive the code shown in Figure 1(b), given the code in Figure 1(a). In this optimized code, the array U_0 has temporal reuse in the innermost loop (the j loop now) and the arrays U_1 and U_2 have spatial reuses, meaning that the successive iterations of the innermost loop touch consecutive elements from both the arrays.

However, unless the faster memory in question is large enough to hold the entire $N \times N$ array U_1, many elements of this array will probably be replaced from the cache before they are reused in successive iterations of the outermost i loop. Instead of operating on individual array elements, tiling achieves reuse of array sections by performing the calculations (in our case matrix multiplication) on array sections (in our case sub-matrices). Figure 1(c) shows the tiled version of Figure 1(b). This tiled version is from [1]. In the tiled code, the loops kk and jj are called the tile loops, whereas the loops i, k, and j are called the element loops. It is important to choose the tile size (blocking factor) B such that all the $B^2 + 2NB$ array items accessed by the element loops i, k, j should fit in the faster memory (e.g., cache). In other words, the tiled version of the matrix-multiply code operates on $N \times B$ sub-matrices of arrays U_0 and U_2, and a $B \times B$ sub-matrix of array U_1 at one time. Assuming that the matrices in this example are in main memory to begin with, ensuring that $B^2 + 2NB$ array elements can be kept in cache might be sufficient to obtain high levels of performance. In practice, however, depending on the cache size, cache associativity, and absolute array addresses in memory, cache conflicts can occur. Consequently, the tile size B is set to a much smaller value than necessary [1].

```
for(i = 1; i ≤ N; i++)
  for(j = 1; j ≤ N; j++)
    for(k = 1; k ≤ N; k++)
      U2[i][j]+ = U0[i][k] * U1[k][j];
```

(a)

```
for(i = 1; i ≤ N; i++)
  for(k = 1; k ≤ N; k++)
    for(j = 1; j ≤ N; j++)
      U2[i][j]+ = U0[i][k] * U1[k][j];
```

(b)

```
for(kk = 1; kk ≤ N; kk = kk + B)
  for(jj = 1; jj ≤ N; jj = jj + B)
    for(i = 1; i ≤ N; i++)
      for(k = kk; k ≤ min(N, kk + B − 1); k++)
        for(j = jj; j ≤ min(N, jj + B − 1); j++)
          U2[i][j]+ = U0[i][k] * U1[k][j];
```

(c)

Fig. 1. (a) Matrix-multiply nest. (b) Locality-optimized version. (c) Tiled version.

3 Description of Data Space Oriented Tiling

The traditional iteration space tiling tiles the iteration space taking into account data dependences in the code. In such a tiling strategy, it is not guaranteed that the array elements accessed by a given iteration space tile form a region (which we can refer to as data tile) that exhibits locality. Also, since each nest is tiled independently from other nests in the code, it may not be possible to exploit potential data reuse between different nests due to common array regions accessed.

In contrast, data space oriented tiling (DST) takes a different approach. Instead of focussing on iteration space, it first considers data space (e.g., an array). Specifically, it divides a given array into logical partitions (data tiles) and processes each data tile in turn. In processing a data tile, it traverses the entire iteration space of all nests in the code and executes all iterations (potentially coming from different nests) that access the current data tile being processed. In doing so, it takes data dependences into account. Note that since a data space is common across all nests that access it, DST can potentially achieve better results than traditional tiling by exploiting inter-nest locality. Note also that, as opposed to tradition tiling, this data oriented tiling approach can also handle imperfectly-nested loops easily as it is not restricted by the way the loops in the nests are structured.

In [4], Kodukula et al. present a data oriented tiling strategy called data shackling, which is similar to our approach in spirit. However, there are significant differences between these two optimization strategies. First, the way that a data tile shape is determined in [4] is experimental. Since they mainly focus on linear algebra codes, they decided that using square (or rectilinear) tiles would work well most of the time; that is, such tiles lead to legal (semantics-preserving) access patterns. In comparison, we first summarize the access patterns of multiple nests on data space using data relation vectors, and then select a suitable tile shape so as to minimize the communication volume. Second, their optimization strategy considers only a single imperfectly-nested loop at a time, while we attempt to optimize all the nests in the code simultaneously. Therefore, our approach is expected to exploit inter-nest data reuse better. Third, we also present an automated strategy to handle data dependences. Instead, their work is more oriented towards determining legality for a given data tile (using a polyhedral tool). Finally, the application domains of these two techniques are also different. The approach discussed in [4] is specifically designed for optimizing cache locality. As will be explained later in the paper, we instead mainly focus on improving memory energy consumption of a scratch pad memory based architecture. Therefore, in our case, optimizing inter-nest reuse is more important.

To illustrate the difference between iteration space oriented tiling and data space oriented tiling, we consider the scenario in Figure 2 where an array is manipulated using three separate nests and there are no intra-nest or inter-nest data dependences in the code. As shown on the upper-left portion of the figure, the array is divided into two sections (regions): a and b. The iteration spaces of the nests are divided into four regions. Each region is identified using letters a or b to indicate the data region it accesses. Figure 2 also shows three possi-

ble execution orders. In the execution order (I), the traditional tiling approach is shown, assuming that the sections in the iteration space are processed from left-to-right and top-to-bottom. We clearly see that there are frequent transitions between a-blocks and b-blocks, which is not good from the data locality perspective. The execution order (II) illustrates a data oriented approach which restricts its optimization scope to a single nest at a time (as in [4]). That is, it handles nests one-by-one, and in processing a nest it clusters iterations that access a given data region. Consequently, it does not incur transitions between a-blocks and b-blocks in executing a given nest, a big advantage over the scheme in (I). However, in going from one nest to another, it incurs transitions between a-blocks and b-blocks. Finally, the execution order (III) represents our approach. In this strategy, we process data regions one-by-one, and in processing a region, we execute all iterations from all nests that access the said region. Therefore, our approach first executes iterations (considering all nests) that access a-block, and then executes all iterations that access b-blocks. Consequently, there is only one transition between a-blocks and b-blocks. However, there are many issues that need to be addressed. First, in some cases, inherent data dependences in the program may not allow interleaving loop iterations from different nests. Second, in general, a given code may contain multiple arrays that need to be taken into account. Third, shape of the data regions might also have a significant impact on the success of the strategy (in particular, when we have data dependences). Data space oriented tiling is performed in two steps: (i) selecting an array (called the seed array) and determining a suitable tile shape for that array, and (ii) iterating through data tiles and for each tile executing all iterations (from all nests in the code) that manipulate array elements in the data tile. In the remainder of this paper, we address these issues in detail.

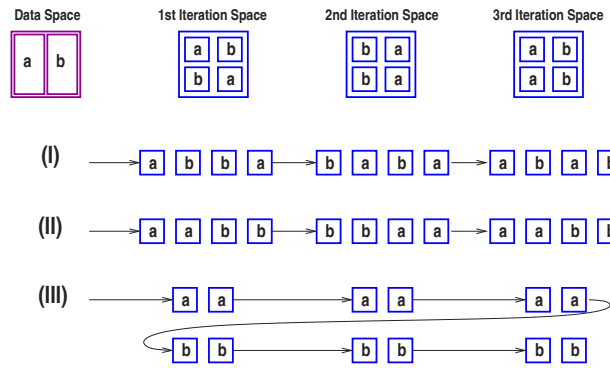

Fig. 2. Comparison of iteration space oriented tiling and data space oriented tiling.

The loop iterators surrounding any statement can be represented as an $n \times 1$ column vector: $\boldsymbol{i} = [i_1, i_2, \cdots, i_n]^T$, where n is the number of enclosing loop iterators. The loop bounds of the iterators can be described by a system of inequalities which define the polyhedron $\mathcal{A}\boldsymbol{i} \leq \boldsymbol{b}$ where $\mathcal{A}$ is an $l \times n$ integer

matrix and $\boldsymbol{b}$ is an l vector. The integer values taken by $\boldsymbol{i}$ define the iteration space of the iterators.

The data storage of an array U_0 can also be viewed as a (rectilinear) polyhedron. The index domain of array U_0 can be described using index vectors: $\boldsymbol{a} = \left[a_1, a_2, \cdots, a_{dim(U_0)}\right]^T$, where $dim(U_0)$ refers to the dimensionality of U_0. The index vectors have a certain range which describe the size of the array, or data space: $\boldsymbol{\mu}_{LB} \leq \boldsymbol{a} \leq \boldsymbol{\mu}_{UB}$, where the $dim(U_0) \times 1$ vectors $\boldsymbol{\mu}_{LB}$ and $\boldsymbol{\mu}_{UB}$ correspond to lower and upper bounds of the array, respectively. In this paper, we assume that $\boldsymbol{\mu}_{LB} = [1, 1, \cdots, 1, 1]$; that is, the lowest index value in each subscript position is 1.

The subscript function for a reference to array U_0 represents a mapping from iteration space to data space. An iteration vector $\boldsymbol{i}$ is said to access (or reference) an array element indexed by $\boldsymbol{a}$ if there exists a subscript function (or array reference) $R_{U_0}(.)$ such that $R_{U_0}(\boldsymbol{i}) = \boldsymbol{a}$. In our context, an array reference can be written as an affine mapping that has the form $L\boldsymbol{i} + \boldsymbol{o}$, where L is a $dim(U_0) \times n$ matrix and $\boldsymbol{o}$ is a $dim(U_0) \times 1$ vector. For example, an array reference such as $U_0[i-1][i+j+2]$ in a two-level nested loop (where i is the outer loop and j is the inner loop) can be represented as

$$R_{U_0}(\boldsymbol{i}) = L\boldsymbol{i} + \boldsymbol{o} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} -1 \\ 2 \end{bmatrix},$$

where $\boldsymbol{i} = [i \quad j]^T$. When there is no confusion, we write $R_{U_0} \in N_k$ to indicate that the reference $R_{U_0}(.)$ appears in nest N_k.

3.1 Array Selection and Tile Shapes

The first step in DST is selecting a suitable array (called the seed array) from among the arrays declared in the code and determining a suitable data tile shape for this array. Once the shape of the tile has been determined, its sizes in different dimensions can be found by scaling up its shape.

Let us assume for now that we have already selected a seed array, U_0. A data tile corresponds to set of array elements in a data space (array). To define a suitable data tile for a given seed array, we need to consider the access pattern of each nest on the said array. For a given seed array U_0 and a nest N_i, the seed element of U_0 with respect to nest N_i, denoted $\boldsymbol{s}_{U_0,N_i}$, is the lexicographically smallest element of the array accessed by N_i. Based on this definition, the global seed element $\boldsymbol{g}_{U_0}$ for array U_0 is the smallest array element accessed by all nests in the code. In cases where there is not such a global seed, we select an element which is accessed by most of the nests.

Using this global seed element, we determine a seed iteration for each nest as follows. The seed iteration of nest N_i with respect to array U_0 is an iteration $\boldsymbol{i}_{\boldsymbol{s}U_0,N_i}$ that among the elements accessed by this iteration, $\boldsymbol{g}_{U_0}$ is the smallest one in lexicographic sense. If there are multiple seed iterations (for a given nest), we select the lexicographically smallest one. Then, we define the footprint of nest N_i with respect to array U_0 (denoted $\mathcal{F}_{U_0,N_i}$) as the set of elements accessed by $\boldsymbol{i}_{\boldsymbol{s}U_0,N_i}$. More precisely, $\mathcal{F}_{U_0,N_i} = \{\boldsymbol{f} \mid \boldsymbol{f} = R_{U_0}(\boldsymbol{i}_{\boldsymbol{s}U_0,N_i}) \text{ for all } R_{U_0} \in N_i\}$.

Let us define a set of vectors ($\mathcal{V}_{U_0,N_i}$), called data relation vectors, on the data space of U_0 using the elements in the footprint $\mathcal{F}_{U_0,N_i}$. Specifically, let $\mathcal{F}_{U_0,N_i} =$

$\{\boldsymbol{f}_1, \boldsymbol{f}_2, \cdots, \boldsymbol{f}_k\}$, where the elements in this set are ordered lexicographically, $\boldsymbol{f}_1$ being the lexicographically smallest one. Each $\boldsymbol{v}_j \in \mathcal{V}$ represents a vector between $\boldsymbol{f}_i$ and $\boldsymbol{f}_k$, where $k > i$. In other words, by doing so, we define a set of lexicographically positive vectors between all data point pairs in $\mathcal{F}_{U_0,N_i}$. We can write $\mathcal{V}_{U_0,N_i}$ as a matrix $[\boldsymbol{v}_1; \boldsymbol{v}_2; \cdots; \boldsymbol{v}_L]$. This matrix is termed as the local data relation matrix.

The global data relation matrix of array U_0 (denoted $\mathcal{G}_{U_0}$) is the combination of local data relation matrices coming from individual nests; that is, $\mathcal{G}_{U_0} = [\mathcal{V}_{U_0,N_1}; \mathcal{V}_{U_0,N_2}; \cdots \mathcal{V}_{U_0,N_P}]$, where P is the number of nests in the code. If desired, the (column) vectors in $\mathcal{G}_{U_0}$ can be re-ordered according to their frequency of occurrence. Our approach uses $\mathcal{G}_{U_0}$ to define tile shapes on data space. Specifically, we first find the vectors in $\mathcal{G}_{U_0}$ and cover the entire data space (of the array in question) using these vectors. The positions of these vectors on the data space is used in selecting a data tile shape. Our objective in selecting a data tile shape is to ensure that, when executing a group of iterations that access the elements in a given data tile, the number of non-tile elements accessed should be minimized as much as possible. Obviously, the shape of the data tile plays a major role in determining the number of the non-tile elements accessed.

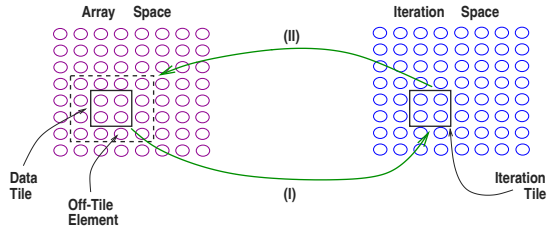

Fig. 3. Going from data tile to iteration tile and off-tile (non-tile) elements.

We next define communication volume as the number of non-tile elements accessed during the execution of iterations that manipulate the elements in the tile. It should be noted that, for a given data tile, the execution of each nest might incur a non-zero communication volume. We then try to minimize the global (over all nests) communication volume. It should also be noted that a non-tile element access occurs due to a relation vector that crosses a tile boundary (i.e., one of its end-points are inside the tile whereas the other end-point lies outside the tile). As an example, consider the iteration space and data space shown in Figure 3. Considering the data tile on the left side of the figure, our approach determines an iteration tile (on the iteration space). This activity is marked (I) in the figure. The iterations in the iteration tile are the ones that access the array elements in the data tile. We will make a more accurate definition of iteration tile later in the paper. Next, the entire set of array elements accessed by this iteration tile is determined. This step corresponds to (II) in the figure. These array elements are delimited using a dashed box in the figure. The array elements

that are within the dashed box but outside the data tile are called off-tile (or non-tile) elements. The objective of our tile selection strategy is to minimize the number of off-tile elements.

A given data tile can be defined using a set of hyperplanes. Specifically, data tiles in an M-dimensional space can be defined by M families of parallel hyperplanes (or planes), each of which is an $(M-1)$-dimensional hyperplane. Data tiles so defined are parallelepipeds (except for those near the boundary of the data space) and each tile is an M-dimensional subset of the data space. Thus, the shape of the tiles is defined by the families of planes and the size of the tiles is defined by the distance of separation between adjacent pairs of parallel planes in each of the M families. We can represent a given tile to array U_0 using M vectors, where the ith vector $\boldsymbol{p}_i$ $(1 \leq i \leq M)$ corresponds to the ith boundary of the tile. These vectors can collectively be written as a matrix $P_{U_0} = [\boldsymbol{p}_1; \boldsymbol{p}_2; \cdots; \boldsymbol{p}_M]$. Alternatively, a given data tile can be defined using another matrix, H_{U_0}, each row of which is perpendicular to a given tile boundary. It can be shown that $H_{U_0} = {P_{U_0}}^{-1}$. Consequently, to define a data tile, we can either specify the columns of P_{U_0} or the rows of H_{U_0}.

We then try to select a tile shape such that the number of data relation vectors intersected by tile boundaries will be minimum. As mentioned earlier, each such vector (also referred to as the communication vector) represents two elements, one of which is within the tile whereas the other is outside the tile. Note that such vectors are the most important ones to concentrate on as the vectors with both the ends are outside can be converted to either the communication vectors or the vectors which are contained completely in the tile by making the tile large enough. It should also be noted that using H_{U_0} and $\mathcal{G}_{U_0}$, we can represent the communication requirements in a concise manner. Specifically, since data tiles are separated by tile boundaries (defined by H_{U_0}), a communication vector must cross the tile boundary between the tiles. A non-zero entry in ${\mathcal{G}'}_{U_0} = H_{U_0}\mathcal{G}_{U_0}$, say the entry in (i, j), implies that communication is incurred due to the jth communication vector poking the ith tile boundary. The amount of communication across a tile boundary, defined by the ith row of H_{U_0}, is a function of the sum of the entries in the ith row of ${\mathcal{G}'}_{U_0}$.

Based on this, we can formulate the problem of finding tiling planes as that of finding a transformation H_{U_0} such that the communication volume (due to communication vectors) will be minimum. Note that the communication volume is proportional to:

$$\sum_{i=1}^{M}\sum_{j=1}^{S}\sum_{k=1}^{M} h_{i,k}v_{k,j}.$$

As an example, let us consider the code fragment given below, which consists of two separate nests. Figure 4(a) shows the local data relation vectors for each nest as well as the global data relation vector (only the first 3×3 portion of the array is shown for clarity). Figure 4(b) shows how the global data relation vectors can be used to cover the entire data space. This picture is then used to select a suitable tile shape. It should be noted that the global data relation matrix in this example is:

$$\mathcal{G}_{U_0} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}.$$

```
for(i = 1; i ≤ N − 1; i++)
  for(j = 1; j ≤ N − 1; j++)
    {U0[i][j], U0[i + 1][j + 1]};
for(i = 1; i ≤ N; i++)
  for(j = 1; j ≤ N − 1; j++)
    {U0[i][j], U0[i][j + 1]};
```

Assuming a data tile capacity (size) of six elements, Figure 4(c) shows three alternative data tile shapes with their communication vectors. It should be noted that each tile in this figure has a different communication volume. For example, the data tile in (I) has a communication volume of 12, corresponding to six in-coming edges and six out-going edges. The tile in (II), on the other hand, has a communication volume of 14. Finally, the tile (III) has a communication volume of 10. Consequently, for the best results, tile (III) should be selected. In fact, it is easy to see that, in this example, if the dimension sizes of the rectangular data tile (as in (I) and (II)) are n (vertical) and m (horizontal), then the communication volume is $4n + 2(m - 1)$. For the tile in (III), on the other hand, the corresponding figure is $2(m+n)$. As an example, if $n = m = 50$, the communication volume of the tile in (III) is 32% less than the one in (I).

To show how our approach derives the tile shown in (III) for the example code fragment above, let us define H_{U_0} as:

$$H_{U_0} = \begin{bmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{bmatrix}.$$

Consequently,

$$\mathcal{G}'_{U_0} = H_{U_0}\mathcal{G}_{U_0} = \begin{bmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} h_{11} + h_{12} & h_{12} \\ h_{21} + h_{22} & h_{22} \end{bmatrix}.$$

To minimize the communication volume, the sum of (the absolute values of) the entries in this last matrix should be minimum. This is because, as we have discussed above, each non-zero entry in $\mathcal{G}'_{U_0}$ represents a communication along one surface of the tile. In mathematical terms, we need to select h_{11}, h_{12}, h_{21}, and h_{22} such that $|h_{11} + h_{12}| + |h_{12}| + |h_{21} + h_{22}| + |h_{22}|$ should be minimized. A possible set of values for minimizing this is $h_{11} = 1$, $h_{12} = 0$, $h_{21} = -1$, and $h_{22} = 1$, respectively, which gives us:

$$H_{U_0} = \begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix},$$

which, in turn, means

$$P_{U_0} = {H_{U_0}}^{-1} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}.$$

Recall that each column of the P_{U_0} matrix represents a boundary of data tile. So, the P_{U_0} matrix above represents the data tile (III) illustrated in Figure 4(c). We next explain how a data tile is actually scaled up.

After selecting a data tile shape, it is scaled up in each dimension. In scaling up a data tile, we consider the iterations that follow the seed iteration in execution order. The left part of Figure 4(d) shows the global data relation vectors defined by the seed iteration and three other iterations that follows it. We

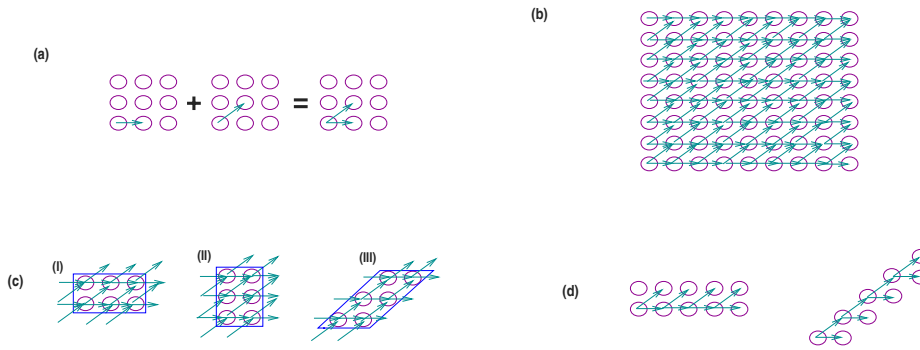

Fig. 4. (a) Local and global data relation vectors. (b) Data space covered by global data relation vectors. (c) Three different data tile shapes with their communication vectors. (d) Scaling up tile size based on array layout. (e) Tiling the entire data space.

can include as many iterations as possible as long as the maximum (allowable) capacity of a data tile is not exceeded. As will be discussed later in the paper, the maximum capacity of a data tile depends on the application at hand and the memory architecture under consideration. The iterations that (follow the seed iteration and) are included in determining the size of a tile constitute an iteration tile. It should be noted, however, that we do not necessarily include the iterations that immediately follow the seed. Instead, we can take into account array layout and determine a suitable iteration tile such that the spatial locality is exploited as much as possible. For example, as we can see on the left portion of Figure 4(d), for this example, progressing the relation vectors (that is, stretching the data tile) along the horizontal axis makes sense since the array layout is row-major. If, however, the array layout was column-major, it would be more beneficial to stretch the tile along the vertical exis as illustrated one the right side of Figure 4(d). Our current implementation takes the maximum capacity of the tile and the array layout into account, and determines the iterations in the iteration tile. Note that, for a given data tile, each nest may have a different iteration tile.

As will be explained in detail in the next section, once a suitable data tile (shape/size) has been selected, our approach considers data tiles one-by-one, and for each data tile, executes iterations that access the array elements in the tile. It should be noted, however, iterations that manipulate elements in a given tile may also access elements from different arrays. Assuming that we have tiles for these arrays as well, these accesses may also incur communication (i.e., accesses to non-tile elements). Consequently, just considering the seed array and its communication volume may not be sufficient in obtaining an overall satisfactory performance (that is, minimizing the communication volume due to all arrays and all nets). It should also be noted that the selection of the seed array is very important as it determines the execution order of loop iterations,

how the iterations from different nests are interleaved, and tile shapes for other arrays. Our current approach to the problem of selecting the most suitable seed array is as follows. Since the number of arrays in a given code is small, we consider each array in turn as the seed array and compute the size of the overall communication set. Then, we select the array which leads to overall minimum communication when used as the seed.

3.2 Traversing Iteration Space

In this subsection, we assume that there exists no data dependences in the code. Once a seed array has been determined and a data tile shape/size has been selected, our approach divides the array into tiles. The tiles are exact copies of each other except maybe at the boundaires of the array space. It then re-structures the code so that the (re-structured) code, when executing, reads each data tile, executes loop iterations (possibly from different nests) that accesses its elements, and moves to the next tile. Since, as explained in the previous section, we are careful in selecting the most suitable tile shape, in executing iterations for a given tile, the number of off-tile (non-tile) elements will be minimum.

However, we need to be precise in defining the iterations that manipulate the elements in a given tile. This is because even iterations that are far apart from each other can occasionally access the same element in a given tile. For this purpose, we use the concept of the iteration tile given above. Let us focus on a specific data tile of T elements:

$$DT_{U_0} = \{\boldsymbol{a}_1, \boldsymbol{a}_2, \cdots, \boldsymbol{a}_{T-1}, \boldsymbol{a}_T\},$$

assuming that $\boldsymbol{g}_{U_0} = \boldsymbol{a}_1$ and H_{U_0} is the corrsponding tile matrix. Let

$$IT_{U_0,N_i} = \{\boldsymbol{i}_1, \boldsymbol{i}_2, \cdots, \boldsymbol{a}_{T-1}, \boldsymbol{a}_T\}$$

be the corresponding iteration tile for nest N_i, where $\boldsymbol{i}_{\boldsymbol{s}U_0,N_i} = \boldsymbol{i}_1$.

When DT_{U_0} is processed, the corresponding IT_{U_0} is determined. This IT, in turn, determines the data tiles for the other arrays in the code. This is depicted in Figure 5, assuming that there are three arrays in the code and a single nest: U_0 (the seed array), U_1, and U_2. We first determine the data tile for U_0 (the seed array). Then, using this data tile, we find the corresponding iteration tile. After that, using this iteration tile, we determine data tiles for arrays U_1 and U_2. Once this iteration tile is executed, our approach processes the next tile from U_0 and so on. If there exist multiple nests in the code being optimized, when we process the data tile, we execute all iterations from the corresponding iteration tiles of all nests. Let us number the tiles in a given data space (array) from 1 to Y. Let us also denote $DT_{U_0}(j)$ the jth data tile (from array U_0) and $IT_{U_0,N_i}(j)$ the corresponding iteration tile from nest N_i. We process data tiles and execute corresponding iteration tiles in the following order (in Y steps):

$$\begin{aligned}
DT_{U_0}(1) &: IT_{U_0,N_1}(1), IT_{U_0,N_2}(1), \cdots, IT_{U_0,N_P}(1)\\
DT_{U_0}(2) &: IT_{U_0,N_1}(2), IT_{U_0,N_2}(2), \cdots, IT_{U_0,N_P}(2)\\
&\vdots \vdots \quad \vdots \quad \vdots\\
DT_{U_0}(Y) &: IT_{U_0,N_1}(Y), IT_{U_0,N_2}(Y), \cdots, IT_{U_0,N_P}(Y)
\end{aligned}$$

In other words, the iterations from different nests are interleaved. This is possible as we assumed that no data dependence exists in the code. When there are data dependences, however, the execution order of loop iterations is somewhat restricted as discussed in the next section.

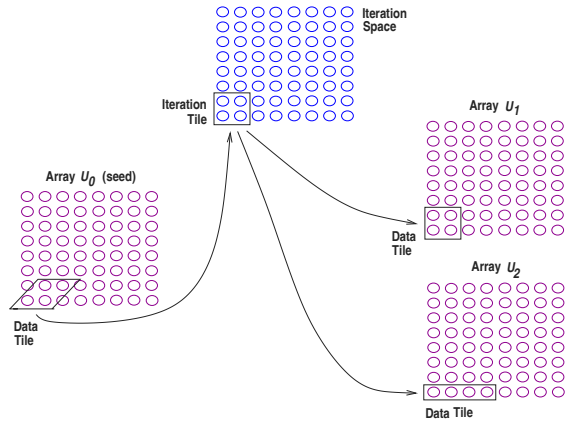

Fig. 5. Determining data tiles of non-seed arrays using the iteration tile defined by the data tile of the seed array.

3.3 Handling Data Dependences

As mentioned earlier, ideally, we would like to execute loop iterations as follows. We consider data tiles from the array one-by-one, and for each data tile, we execute all loop iterations (and only those iterations) that access array elements in the tile. However, if the communication volume of data tile is not zero, this ideal execution pattern would not happen. This is because, in executing some iterations, we might need to access elements from other tiles as well. Obviously, if we are able to select a good data tile (using the strategy explained earlier), the number of such non-tile accesses will be minimized. Note that, even in a loop without data dependences, we can experience non-tile accesses. However, when data dependences exist in the code, we can expect that such off-tile accesses will be more as the iterations that access the elements in the current tile might involve in data dependence relationships with other iterations.

For the sake of presentation, let us assume that there are two nests in the code (N_1 and N_2) and a single array (U_0). Assume that $DT_{U_0}(1)$ is a data tile for array U_0 and let $IT_{U_0,N_1}(1)$ and $IT_{U_0,N_2}(1)$ be the corresponding iteration tiles for nests N_1 and N_2. Assume further that $IT'_{U_0,N_1}(1)$ is the set of iterations (in N_1) other than those in $IT_{U_0,N_1}(1)$. Note that $IT_{U_0,N_1}(1)$ and $IT'_{U_0,N_1}(1)$ are disjoint and their union gives the iteration space of nest N_1. We can define a similar $IT'_{U_0,N_2}(1)$ set for nest N_2. Consider now the iteration sets $IT_{U_0,N_1}(1)$, $IT'_{U_0,N_1}(1)$, $IT_{U_0,N_2}(1)$, and $IT'_{U_0,N_2}(1)$ shown in Figure 6(a). If there are no data dependences in the code, when processing $DT_{U_0}(1)$, we

can execute $IT_{U_0,N_1}(1)$ followed by $IT_{U_0,N_2}(1)$. Note that this corresponds to the ideal case as these two iteration sets, namely, $IT_{U_0,N_1}(1)$ and $IT_{U_0,N_2}(1)$, access the same data tile, so executing them one after another (without an intervening iteration from $IT'_{U_0,N_1}(1)$ or $IT'_{U_0,N_2}(1)$) represents the best possible scenario. Note also that even if there are data dependences between iterations in $IT_{U_0,N_1}(1)$ (and/or between iterations in $IT_{U_0,N_2}(1)$) but not across iterations of different sets, we can still execute $IT_{U_0,N_1}(1)$ followed by $IT_{U_0,N_2}(1)$, provided that we execute iterations in $IT_{U_0,N_1}(1)$ (and also in $IT_{U_0,N_2}(1)$) in their original execution order. This execution order is also valid if there are dependences from $IT_{U_0,N_1}(1)$ (resp. $IT_{U_0,N_2}(1)$) to $IT'_{U_0,N_1}(1)$ (resp. $IT'_{U_0,N_2}(1)$) only. These cases are superimposed in Figure 6(b). Once all the iterations in $IT_{U_0,N_1}(1)$ and $IT_{U_0,N_2}(1)$ have been executed, we can proceed with $DT_{U_0}(2)$. The dashed arrow in Figure 6(b) represents the execution order of these sets.

Suppose now that there exists a dependence from an iteration $\boldsymbol{i'} \in IT'_{U_0,N_1}(1)$ to an iteration $\boldsymbol{i} \in IT_{U_0,N_1}(1)$ as shown in Figure 6(c). Assume further that there exists a dependence from an iteration $\boldsymbol{i'} \in IT'_{U_0,N_2}(1)$ to an iteration $\boldsymbol{i} \in IT_{U_0,N_2}(1)$. In this case, it is not possible to execute $IT_{U_0,N_1}(1)$ followed by $IT_{U_0,N_2}(1)$ as doing so would modify the original semantics of the code (i.e., violate data dependences). To handle this case, our approach breaks $IT'_{U_0,N_1}(1)$ into two groups, $IT'_{U_0,N_1}(1a)$ and $IT'_{U_0,N_1}(1b)$, such that there is a dependence from $IT'_{U_0,N_1}(1a)$ to $IT_{U_0,N_1}(1)$, but not from $IT'_{U_0,N_1}(1b)$ to $IT_{U_0,N_1}(1)$. This situation is depicted in Figure 6(d). Note that in the degenerate case one of $IT'_{U_0,N_1}(1a)$ and $IT'_{U_0,N_1}(1b)$ can be empty. Similarly, we also divide $IT'_{U_0,N_2}(1)$ into two groups: $IT'_{U_0,N_1}(1a)$ and $IT'_{U_0,N_1}(1b)$. Then, a suitable order of execution (during processing $DT_{U_0}(1)$) is $IT'_{U_0,N_1}(1a)$, $IT'_{U_0,N_2}(1a)$, $IT_{U_0,N_1}(1)$, $IT_{U_0,N_2}(1)$, which is also illustrated in Figure 6(d). It should be noticed that, in this scenario, although we need to execute sets $IT'_{U_0,N_1}(1a)$, $IT'_{U_0,N_2}(1a)$ before $IT_{U_0,N_1}(1)$, $IT_{U_0,N_2}(1)$, we are still able to execute $IT_{U_0,N_1}(1)$ and $IT_{U_0,N_2}(1)$ one after another, which is good from the locality viewpoint.

Let us now consider the scenario in Figure 6(e) that indicates data dependences from $IT_{U_0,N_1}(1)$ and $IT'_{U_0,N_1}(1)$ to $IT_{U_0,N_2}(1)$ and $IT'_{U_0,N_2}(1)$. To handle this case, we break $IT'_{U_0,N_1}(1)$ into two subsets, $IT'_{U_0,N_1}(1a)$ and $IT'_{U_0,N_1}(1b)$, such that there are no dependences from the set $IT'_{U_0,N_1}(1b)$ to the set $IT'_{U_0,N_2}(1)$. Then, the preferred execution order is shown Figure 6(f).

4 Application of Data Space Oriented Tiling

There are several applications of data space oriented tiling. One of these is improving cache locality in array-dominated applications. Since DST captures data accesses in a global (procedure-wide) manner, it has better potential for improving cache locality compared to conventional iteration space oriented tiling. In this section, however, we focus on a similar yet different application area: using data space oriented tiling for exploiting an on-chip scratch pad memory (SPM).

Scratch pad memories (SPMs) are alternatives to conventional cache memories in embedded computing world [7,8]. These small on-chip memories, like

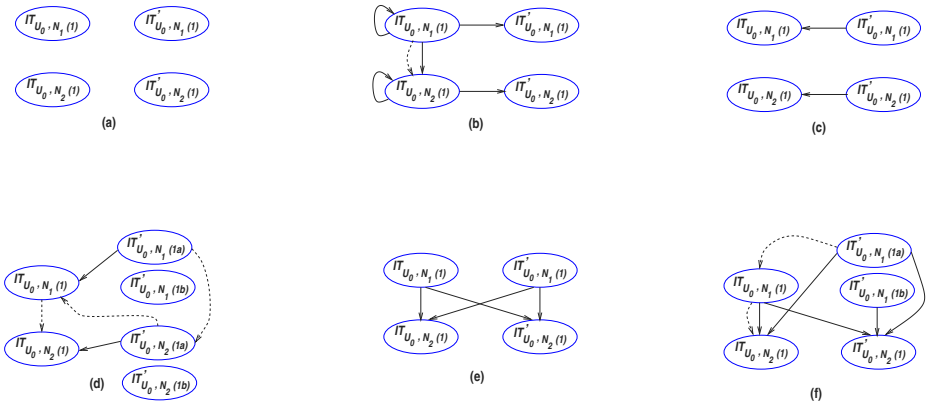

Fig. 6. Different iteration sets and dependences between them. Note that a solid arrow denotes a data dependence, whereas a dashed arrow denotes a legal execution order.

caches, provide fast and low-power access to data and instructions; but, they differ from conventional data caches in that their contents are managed by software instead of hardware. Since the software is in full control of what the contents of the SPM will be at a given time, it is easy to predict memory access times in an SPM-based system, a desired property for real-time embedded systems. Since there is a large difference between access latencies and energy consumptions of these memories, it is important to satisfy as many data requests as possible from SPM. Our compiler-based approach to SPM management determines the contents of the SPM (at every program point) and schedules all data movements between the SPM and off-chip data memory at compile-time. The actual data movements (between SPM and off-chip data memory), however, take place at run-time. In other words, we divide the task of exploiting the software-controlled SPM between compiler and run-time (hardware). It should also be mentioned that in order to benefit from an SPM, the energy (and performance) gains obtained through optimized locality should not be offset by the runtime overheads (e.g., explicit data copies between SPM and off-chip memory). That is, a data item (array element) should be moved to the SPM only if it is likely that it will be accessed from the SPM large number of times (that is, if it exhibits high data reuse).

Our approach works as follows. It first optimizes the code using DST as explained above. It then reads data tiles from off-chip data memory to SPM and executes all iterations that manipulate the data in the SPM. When these iterations are finished, a new set of data tiles are brought into SPM and the corresponding loop iterations are executed, and so on. It should be noted that if, during its stay in the SPM, the data tile has been modified (through a write command) it should be written back to the off-chip memory when it needs to be replaced by another data tile. Figure 7 gives a sketch of the SPM optimization algorithm based on data space oriented tiling. To keep the presentation clear, we assume that all arrays are of the same size and dimensionality, and all arrays are

accessed in each nest. After determining the seed array and the most suitable tile shape from the viewpoint of communication volume, the first loop in this figure iterates over data tiles. In each iteration, we read the corresponding data tiles from the off-chip memory to the SPM. Then, the second loop nest determines the corresponding iteration tiles from all nests and also computes the set of iterations ($\mathcal{I}'$) that should be executed before these iterations (due to data dependences). The compiler then generates code to execute these iterations and updates the iteration sets by eliminating the already executed iterations from further consideration. It should be noted that this is a highly-simplified presentation. In general, the iterations in $\mathcal{I}'$ and $IT_{U_0,N_j}(i)$ might be dependent on each other. In executing the iterations in a given set, we stick to the original execution order (not to violate any dependences). After that, it checks whether the data tiles have been updated while they are in the SPM. If so, they need to be written back to the main memory. This concludes an iteration of the outermost for-loop in Figure 7.

```
INPUT: a set of nests N_j, 1 ≤ j ≤ P accessing K arrays
  I_j: the iteration set of the jth nest

ALGORITHM:
determine the seed array U_0 and data tile shape;
for each data tile i, 1 ≤ i ≤ Y
  generate code to read DT_{U_0}(i), DT_{U_1}(i), ..., DT_{U_K}(i)
    from main memory;
  I_res = ∅;
  for each nest N_j, 1 ≤ j ≤ P
    I_rem = ⋃ I_k, 1 ≤ k ≤ j
    determine IT_{U_0,N_j}(i) and IT'_{U_0,N_j}(i);
    determine I' ⊂ I_rem such that:
      (i) there is a dependence from I' to IT_{U_0,N_j}(i)
      (ii) there is no dependence from (I_rem − I')
        to IT_{U_0,N_j}(i)
      I_res = I_res ⋃ I';
  endfor;
  for each nest N_j, 1 ≤ j ≤ P
    generate code to execute iterations in I'
      (if they have not been executed so far);
    generate code to execute iterations in IT_{U_0,N_j}(i)
      (if they have not been executed so far);
  endfor;
  update I_k, 1 ≤ k ≤ j
  for each U_l, 1 ≤ l ≤ K
    if DT_{U_l}(i) is modified, then write it back
      to main memory;
  endfor;
endfor;
```

Fig. 7. An SPM optimization algorithm based on DST.

5 Conclusions

This paper presents a compiler-based strategy for optimizing data accesses in regular array-dominated applications. Our approach, called data space oriented tiling, is a variant of classical iteration space tiling. It improves over the latter by working with better data tile shapes and by exploiting inter-nest data reuse. This paper also shows how data space oriented tiling can be used to improve the effectiveness of a scratch pad memory.

References

1. S. Coleman and K. McKinley. Tile size selection using cache organization and data layout. In Proc. *the ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1995. 178, 179, 180, 180
2. F. Irigoin and R. Triolet. Super-node partitioning. In Proc. *the 15th Annual ACM Symposium on Principles of Programming Languages*, pages 319–329, January 1988. 179
3. M. Kandemir, J. Ramanujam, M. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh. Dynamic management of scratch-pad memory space. In Proc. *the 38th Design Automation Conference,* Las Vegas, NV, June 2001.
4. I. Kodukula, N. Ahmed, and K. Pingali. Data-centric multi-level blocking. In Proc. *the SIGPLAN Conference on Programming Language Design and Implementation*, June 1997. 181, 181, 181, 182
5. W. Li. *Compiling for NUMA Parallel Machines*. Ph.D. Dissertation, Computer Science Department, Cornell University, Ithaca, NY, 1993. 180
6. M. O'Boyle and P. Knijnenburg. Non-singular data transformations: Definition, validity, applications. In Proc. *the 6th Workshop on Compilers for Parallel Computers*, pages 287–297, 1996.
7. P. R. Panda, N. D. Dutt, and A. Nicolau. Efficient utilization of scratch-pad-memory in embedded processor applications. In Proc. *the European Design and Test Conference (ED&TC'97),* Paris, March 1997. 190
8. L. Wang, W. Tembe, and S. Pande. Optimizing on-chip memory usage through loop restructuring for embedded processors. In Proc. *the 9th International Conference on Compiler Construction,* March 30–31 2000, pp. 141–156, Berlin, Germany. 190
9. M. Wolfe. *High Performance Compilers for Parallel Computing*, Addison-Wesley Publishing Company, 1996. 178, 179
10. J. Xue and C.-H. Huang. Reuse-driven tiling for data locality. In *Languages and Compilers for Parallel Computing*, Z. Li et al., Eds., Lecture Notes in Computer Science, Volume 1366, Springer-Verlag, 1998.