Effective Automatic Parallelization of Stencil Computations

Sriram Krishnamoorthy[1] Muthu Baskaran[1] Uday Bondhugula[1]
J. Ramanujam[2] Atanas Rountev[1] P. Sadayappan[1]

[1]Dept. of Computer Science and Engineering
The Ohio State University
2015 Neil Ave. Columbus, OH, USA
{krishnsr,baskaran,bondhugu,rountev,saday}@cse.ohio-state.edu

[2]Dept. of Electrical & Computer Engg. and
Center for Computation & Technology
Louisiana State University
jxr@ece.lsu.edu

Abstract

Performance optimization of stencil computations has been widely studied in the literature, since they occur in many computationally intensive scientific and engineering applications. Compiler frameworks have also been developed that can transform sequential stencil codes for optimization of data locality and parallelism. However, loop skewing is typically required in order to tile stencil codes along the time dimension, resulting in load imbalance in pipelined parallel execution of the tiles. In this paper, we develop an approach for automatic parallelization of stencil codes, that explicitly addresses the issue of load-balanced execution of tiles. Experimental results are provided that demonstrate the effectiveness of the approach.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Compilers, Optimization

General Terms Algorithms, Performance

Keywords Stencil computations, Tiling, Automatic parallelization, Load balance

1. Introduction

Stencil computations represent a practically important class of computations that arise in many scientific/engineering codes. Computational domains that involve stencils include those that use explicit time-integration methods for numerical solution of partial differential equations (e.g., climate/weather/ocean modeling [23], computational electromagnetics codes using the Finite Difference Time Domain method [27]), and multimedia/image-processing applications that perform smoothing and other neighbor pixel based computations [13]. There has been some prior work from the computer science community that has addressed performance optimization of stencil computations (e.g., [24, 19, 18, 10]). Since stencil computations are characterized by a regular computational structure, they are amenable to automatic compile-time analysis and transformation for exploitation of parallelism and data locality optimization. However, as elaborated later through an example, existing compiler frameworks have limitations in generating efficient code optimized for parallelism and data locality.

Loop tiling is the key transformation to enable parallelization and data-locality optimization of stencil codes. Much research has been published on tiling of iteration spaces [17, 29, 28, 26, 8, 25, 21, 22, 14, 7, 15, 9, 16, 3]. With few exceptions (e.g. work of Griebl [11, 12]), research on performance optimization with tiling has generally focused on one or the other of the two complementary aspects: (a) data locality optimization [2, 3, 28, 26, 8]; or (b) tile size/shape optimization for parallel execution [25, 21, 6, 14, 7, 15, 9, 16]. Tiling for data locality optimization involves maximization of data reuse, i.e., tiling along directions of the data dependence vectors. But such tiling may result in inter-tile dependences that inhibit concurrent execution of tiles on different processors. To the best of our knowledge, no prior work has addressed in an integrated fashion, the issues of tiling for data locality optimization and load balancing for parallel execution. We first use the simple example of a one-dimensional Jacobi code to illustrate the problem and introduce two approaches we propose to avoid the problem: overlapped tiles and split tiles. As an example of a stencil computation, let us consider the one-dimensional Jacobi code shown in Figure 1. Optimizing this stencil computation for reduction of cache misses requires loop fusion and tiling; in order to fuse the two inner loops, loop skewing is needed. Frameworks have been previously proposed for data locality optimizations of imperfectly nested loops. An approach proposed by Ahmed et al. [3, 4] transforms the loop nest into the one shown in Figure 2 by first embedding the iterations in the imperfectly-nested loops into a perfectly-nested iteration space. Loop transformations and tiling are then applied in the transformed perfectly-nested iteration space. The transformed iteration space is subsequently translated into efficient code by reducing/eliminating the control overhead [20]. In this paper, we focus on load-balanced parallel execution of tiled iteration

PLDI'07 Jun 11–13, 2007, San Diego, California, USA.
Copyright © 2007 ACM 978-1-59593-633-2/07/0006…$5.00

spaces that have already been embedded into a perfectly-nested iteration space using a technique such as the one in [4].

Figure 3 shows a single-statement form of the 1-D Jacobi code obtained by adding an additional dimension to array A. The flow dependences in this code are the same as that of the previously shown version, but there are no anti-dependences. Hence a single statement is sufficient in the loop body instead of a sequence of two statements for update and copy as seen in Figures 1 and 2. Although such a memory-inefficient code would not be used in practice, it is more convenient for us to use a single-statement iteration space in explaining the main ideas in this paper. However, the developed approach is not restricted to such single-statement loops, but is applicable to general multi-statement stencil codes such as the one in Figure 1. The generalization of the approach for the more memory-efficient multi-statement versions of code is explained in the Appendix. The experimental results presented later also use the memory-efficient multi-statement versions.

The perfect loop nest of Figure 3 has constant dependences $(1, 0)$, $(1, 1)$, and $(1, -1)$. Tiling for data reuse optimization (e.g. using the approach presented in [2]) results in tiles of shape as shown in Figure 4. The horizontal axis corresponds to the spatial dimension, with time along the vertical dimension. Using a sufficiently large tile size along the time dimension facilitates significant data reuse within caches/registers. However, there are inter-tile dependences in the horizontal direction, inhibiting concurrent execution of tiles by different processors. But, if the vertical tile size is reduced to one (i.e., eliminate tiling along the time dimension), all tiles along the spatial dimension (adjoining the x-axis) can be executed concurrently. Thus there is a trade-off between achieving good data reuse and load balancing of parallel execution.

Instead of the *standard* tiling described above, consider the tiling shown in Figure 5. Starting with the tiles formed by the same hyperplanes, an additional triangular region is added to the left of the tile, overlapping with the points at the right end of the neighboring tile. With this tiling, the iteration points processed by the tiles are no longer disjoint. Some of the iterations are executed redundantly by two neighboring tiles. This results in an increase in the computation cost. But doing so eliminates the dependence between tiles along the horizontal direction. All processors can start executing in parallel, eliminating the initial processor idling that results with the pipelined parallel execution of tiles in Figure 4.

While standard tiling can enhance data locality in this context, overlapped tiling can both improve data locality and

```
for t = 0 to T-1
 for i = 1 to N-1
   B[i] = (A[i-1]+A[i]+A[i+1])/3; (S1)
 for i = 1 to N-1
   A[i] = B[i]; (S2)
```

Figure 1. One-dimensional Jacobi code

```
for t = 0 to T-1
 for i = 1 to N
   if(i>=1 and i<=N-1)
     B[i] = (A[i-1]+A[i]+A[i+1])/3; (S1)
   if(i>=2 and i<=N)
     A[i-1] = B[i-1]; (S2)
```

Figure 2. Fused one-dimensional Jacobi code

```
for t = 0 to T-1
 for i = 1 to N-1
    A[t,i] =
        (A[t-1,i-1] + A[t-1,i] + A[t-1,i+1])/3;
```

Figure 3. Single-statement form of one-dimensional Jacobi code

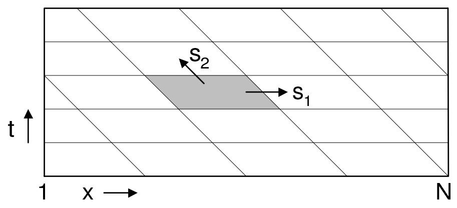

Figure 4. Standard tiling for one-dimensional Jacobi. s_1 and s_2 denote the inter-tile dependences.

eliminate the overhead of pipelined parallelism, at the cost of slightly increased computation time. However, the increased computational cost is independent of tile size, i.e. the fractional computation overhead is inversely proportional to the tile size in the direction of overlapped tiling.

An alternate approach, shown in Figure 6, splits the interior of each tile into two sub-tiles, where the points in only one of the two sub-tiles (shaded) are dependent on points in the neighbor tile, while the points in the other sub-tile are not dependent on any neighboring tile's points, and therefore executable concurrently. With this approach, each standard tile is split into two sub-tiles, and load-balanced concurrent execution is possible as a sequence of two steps: first all non-dependent sub-tiles are concurrently executed and communicate with the neighbor tiles, and then the dependent sub-tiles are all concurrently executed.

The paper is organized as follows. Section 2 defines the problem addressed in this paper. In Section 3, we characterize the conditions under which tiled iteration spaces can benefit from overlapped/split tiling. In Section 4, we show how to transform a given tiled iteration space in order for overlapped/split tiling to be applicable. Section 5 discusses code generation and Section 6 analyzes the cost benefits of overlapped tiling. Section 7 provides experimental results that clearly demonstrate the benefits of overlapped/split tiling. In Section 8, we discuss related work and conclude in Section 9 with a summary.

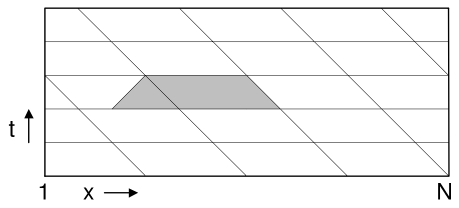

Figure 5. Overlapped tiling for 1-D Jacobi.

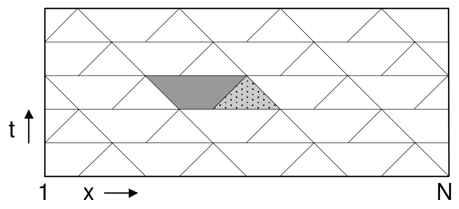

Figure 6. Split tiling for 1-D Jacobi.

2. Background and Problem Statement

This section introduces some standard background on the polyhedral model of computation, and defines the problem statement. Consider a perfectly-nested loop nest with n levels of nesting. The *iteration space polyhedron* defines an n-dimensional set of points, characterized by a set of bounding hyperplanes and modeled as $B.I \geq b$ where I is the iteration vector. The rows b_i of B define the normals to the corresponding bounding hyperplanes. For example, the iteration space for the one-dimensional Jacobi example is

$$\begin{pmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{pmatrix} \cdot \begin{pmatrix} t \\ i \end{pmatrix} \geq \begin{pmatrix} 0 \\ -T+1 \\ 1 \\ -N+1 \end{pmatrix}$$

The dependences in the computation can be represented by a matrix D where each column defines a dependence vector. The dependences in the 1-D Jacobi example are

$$D = \begin{pmatrix} d_1 & d_2 & d_3 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 \\ -1 & 0 & 1 \end{pmatrix}$$

Assume that we are given a set of *tiling hyperplanes* that tile the iteration space. These hyperplanes are encoded by a matrix H, where each row represents the normal vector of a tiling hyperplane. For example, the tiling hyperplanes corresponding to Figure 4 are encoded as

$$H = \begin{pmatrix} h_1 \\ h_2 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$$

A tiling defined by a set of tiling hyperplanes is *legal* if each tile can be executed atomically and there exists a valid total ordering of the tiles. Intuitively, a tiling is legal if no two tiles influence each other. It can be shown [17] that this validity condition is given by

$$H.D \geq 0$$

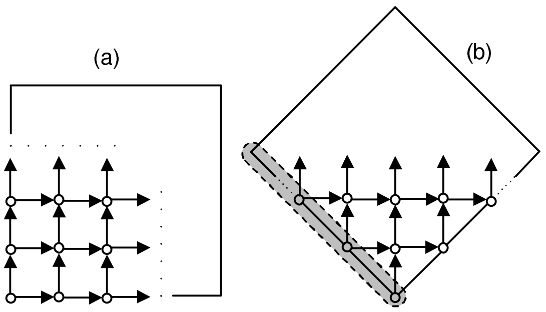

Figure 7. Iteration spaces with $(1, 0)$ and $(0, 1)$ dependencies: (a) concurrent start is not possible (b) concurrent start is possible from the gray boundary.

A schedule has a *concurrent start* if all processors can start execution in parallel, without a pipeline start-up overhead. Such a schedule is referred to as a concurrent start schedule.

Problem Statement. In this paper, we are interested in the following problem. Consider a given (non-tiled) iteration space in which a concurrent start schedule is possible. However, for a given tiling of this space defined by a set of tiling hyperplanes, it is possible that the tile dependencies in the corresponding tiled iteration space inhibit concurrent start. We consider the following question: How can concurrent start be achieved in the tiled iteration space? Our first goal is to characterize analytically the situations in which tiling inhibits concurrent start. Next, we define two approaches, *overlapped* tiling and *split* tiling, that enable concurrent start in the tiled space and recover the load-balancing properties lost due to tiling.

3. Inhibition of Concurrent Start

If the original non-tiled iteration space does not have a concurrent start schedule, tiling cannot enable such a schedule. However, if concurrent start is possible in the absence of tiling, the introduction of tiling can potentially inhibit this concurrent start. This section characterizes the conditions under which a non-tiled space supports a concurrent start schedule, and then derives a concurrent start inhibition condition for the tiled space. For simplicity of presentation, the discussion assumes an iteration space with a single statement, but we have defined a general version of the technique for multi-statement iteration spaces (outlined in the appendix).

3.1 Concurrent Start in the Non-Tiled Space

First, we describe the condition for the existence of concurrent start in the original non-tiled iteration space. Consider, for example, dependence vectors $(1, 0)$ and $(0, 1)$. Two iteration spaces with these dependences are shown in Figure 7. In Figure 7(a), the parallel computation has to begin from the origin $(0, 0)$ and suffers from pipeline start-up overhead. On the other hand, the iteration space in Figure 7(b) can be traversed by all processors in parallel starting from the boundary shown in gray.

In general, the presence of concurrent start in an iteration space depends on the boundaries that define the iteration space polyhedron. An iteration space supports concurrent start if there exists a bounding hyperplane that does not contain a dependence, i.e. carries all dependences. A hyperplane contains a dependence if both the source and destination iteration points of the dependence are contained in the hyperplane. Since the rows b_i of B define the normal vectors of the bounding hyperplanes, this property is represented by the condition

$$\exists b_i \in B \ : \ \forall d_j \in D \ : \ b_i.d_j > 0$$

Note that this condition is independent of the tiling hyperplanes. We will refer to this property as the *point-wise concurrent start condition*. When this condition does not hold, no tiled iteration space can have concurrent start. For the 1-D Jacobi example, the condition holds because the normal vector $b_1 = (1 \ \ 0)$ for one of the bounding hyperplanes satisfies $b_1.d_j > 0$ for all dependence vectors d_j.

3.2 Inhibition of Concurrent Start in the Tiled Space

Next, we consider the condition for the inhibition of the concurrent start condition in the tiled iteration space. Given the tiling hyperplanes and their normal vectors $h_i \in H$, we define the *shift vector* s_i for the hyperplane with h_i as normal to be a vector connecting two instances of the same hyperplane, while traveling parallel to all other hyperplanes. Clearly, the following holds for the set S of shift vectors:

$$\forall s_i \in S \ : \ \forall j \neq i \ : \ h_j.s_i = 0$$

For the 1-D Jacobi example, we will use shift vectors

$$S = \left(\begin{array}{cc} s_1 & s_2 \end{array} \right) = \left(\begin{array}{cc} 0 & 1 \\ 1 & -1 \end{array} \right)$$

as illustrated in Figure 4.

The execution of two adjacent tiles should be ordered if there is a dependence vector d_j such that for some iteration points i_1 and i_2 related by d_j, point i_1 is in one of the tiles and point i_2 is in the other one. Note that this is possible only if there is a dependence that passes through the hyperplane that separates the two tiles — in other words, if the following condition holds

$$\exists d_k \in D \ : \ h_i.d_k \neq 0$$

When this condition is satisfied for a given hyperplane with $h_i \in H$, the shift direction s_i along that dimension *carries the inter-tile dependence*. For the 1-D Jacobi example, both s_1 and s_2 carry inter-tile dependencies (for example, $h_1.d_1 > 0$ and $h_2.d_1 > 0$).

The inter-tile dependences can introduce dependence directions that do not exist in the original iteration space. The concurrent start condition is *inhibited in the tiled iteration space*, if for some boundary b_i, the concurrent start condition is satisfied by the dependences in the original iteration space, but not by the inter-tile dependences in the tiled iteration space. A tiling inhibits concurrent start if

$$\exists b_i \in B, h_j \in H, d_k \in D : b_i.D > 0 \wedge b_i.s_j = 0 \wedge h_j.d_k \neq 0$$

When the above condition is true, there exists an inter-tile dependence within a hyperplane parallel to the boundary b_i, precluding concurrent execution of all the tiles in the boundary. Thus, concurrent start is inhibited even though the original iteration space supports it. This situation occurs for the 1-D Jacobi example due to bounding plane normal $b_1 = (1 \ \ 0)$, tiling hyperplane normal $h_1 = (1 \ \ 0)$, and any dependence d_k for $k = 1 \ldots 3$.

4. Overlapped Tiling

The basic idea behind overlapped tiling is to eliminate certain inter-tile dependencies by "duplicating" points in the original iteration space. As a result, the same iteration point can be a member of two neighboring tiles (i.e., the tiles can overlap). This section outlines a constructive procedure to determine overlapping tiles that eliminate the inter-tile dependences, which removes the inhibition on concurrent start. The key step is the construction of a *companion hyperplane* that eliminates the dependence along a desired direction. The new tile will not have any incoming dependence along the direction in which the dependence was eliminated.

In standard tiling, a hyperplane with a normal vector h_i defines two faces of the tile. We will denote these faces as $h_i(l)$ (the back face) and $h_i(l+1)$ (the front face). The front face is shared with the subsequent tile along the shift direction defined by shift vector s_i. The back face $h_i(l)$ has no incoming dependences if $h_i.D \geq 0$. On the other hand, the front face $h_i(l+1)$, by the tiling validity condition, does not have any incoming dependences. All dependences between the hyperplanes can be eliminated if the back face of the tile is replaced by an overlapped hyperplane with a normal vector h'_i such that

$$\forall d_j \in D : h'_i.d_j \leq 0$$

Note that the hyperplanes span the iteration space and any vector in the iteration space; hence, the companion hyperplane can be defined as a linear combination of the existing hyperplanes. Scaling a given hyperplane vector h_i does not eliminate any additional dependences. In addition, we are interested in the companion hyperplane that forms the back face of the tile. Thus, it is constructed by going "backwards" on the other hyperplanes, represented by a negative linear combination of the hyperplanes, and is given by:

$$h_i.D \geq 0 \Rightarrow h'_i = h_i - \sum_{j \neq i} k_j.h_j \wedge h'_i.D \leq 0 \wedge k_j > 0$$

Such a companion hyperplane eliminates dependences along a shift vector. This procedure is repeated for every hyperplane/shift vector that inhibits concurrent start.

4.1 Cost analysis for overlapped tiling

Consider n-D Jacobi with an $n+1$ dimensional iteration space, and an n dimensional data space with a length of N along each dimension. Let B be the space tile size along each of the n space dimensions. Let p be the number of processors organized in an n-dimensional grid. $B = N/\sqrt[n]{p}$. Let t be the time tile size.

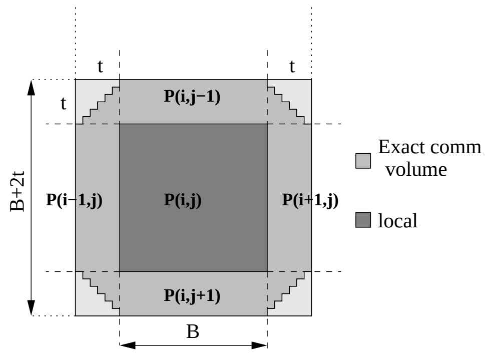

Figure 8. Overlapped tiling for 2-D Jacobi: top view

The schedule for overlapped tiling requires the processors to cycle to maintain load balance. We illustrate the determination of communication frequency using a simpler variation. Starting from orthogonal tiling, both planes can be swiveled partially to form trapezoid-like tiles for 1-D Jacobi, and a square pyramid for 2-D Jacobi, top view for which is shown in Figure 8. This overlapped tiling scheme has the same communication volume as the original one, but double the number of startups. However, code generation is simpler for this case due to the absence of the need to cycle. The number of startup's do not matter when the communication volume is higher; this is particularly true for higher dimensional Jacobi (greater than 1) for which the space tile size comes into the volume.

Consider the overlapped tiling scheme that is obtained from orthogonal tiling. The point-wise difference between the coordinates of a given processor and any of its neighbors in the processor space is an n-vector, and each of its n components being 1, 0, or -1. Discounting the all zeros case, we have $3^n - 1$ neighbors. Hence, the number of communication startups per tile (without forwarding) is given by:

$$S_1 = 3^n - 1 \tag{1}$$

For example, for 3-D Jacobi, we have 8 corners, 12 edges, and 6 faces, i.e., a total of 26 ($= 3^3$-1) neighbors to send and receive data to/from to compute the overlapped tile.

With communication forwarding, the number of communication startups per tile can be reduced to $2n$ (one for each of the faces).

$$S_1' = 2n \tag{2}$$

Similarly the number of startups for the original schedule without and with forwarding are:

$$S_2 = 2^n - 1 \tag{3}$$

$$S_2' = n \tag{4}$$

The exact communication volume assuming orthogonal tiling is given by:

$$\begin{aligned} V &= \sum_{i=1}^{n} \binom{n}{n-i} 2^i B^{(n-i)} f(i,t) \qquad (5)\\ &\approx 2ntB^{n-1} \text{ when } t \ll B \end{aligned}$$

where

$$f(k,t) = \sum_{i_{n-k+1}=1}^{t-1} \cdots \sum_{i_n=1}^{i_{(n-1)}} i_n \tag{6}$$

The communication volume for the original schedule reduces to:

$$\begin{aligned} V &= \sum_{i=1}^{n} \binom{n}{n-i} B^{(n-i)} f(i,2t) \qquad (7)\\ &\approx 2ntB^{n-1} \text{ when } t \ll B \end{aligned}$$

The communication schedule and the data being communication can be quite complex for higher dimensions. Adding a small number of points to the communication volume greatly simplifies code generation. In Figure 8, the points in each of the four corners are those that can be added. The total communication volume then becomes:

$$\begin{aligned} V' &= (B+2t)^n - B^n \\ &= {}^nC_1 B^{n-1}(2t) + {}^nC_2 B^{n-2}(2t)^2 \\ &\quad + \ldots + (2t)^n \qquad (8)\\ &\approx 2ntB^{n-1} \text{ if } t \ll B \\ &= \Theta(tB^{n-1}) \qquad (9) \end{aligned}$$

For $n = 2$:

$$V' = 4tB + 4t^2 \tag{10}$$

4.2 Split Tiling

Overlapped tiling eliminates inter-tile dependences by redundantly computing portions of a tile. While eliminating dependences, this approach increases the overall amount of computation. In this section we leverage the idea of dependence inhibition to develop an alternative approach, referred to as *split tiling*, in order to enable concurrent start without the computation overhead. In split tiling, rather than redundantly computing a portion of the predecessor tile along a dimension, the processor executing the predecessor tile first computes that portion and sends the results to its successor along that dimension.

We show that for stencil computations, a tile sub-region can be identified such that this sub-region can be executed in parallel in all tiles. This enables concurrent start. We outline an algorithm that divides a tile into sub-regions and schedules the computation and communication to achieve concurrent start and load-balanced execution in which all processors execute the same amount of work in all the steps in the schedule.

4.2.1 Tile Regions

A tile in a stencil computation is bounded by the hyperplane instances:

$$\forall I, B.I \geq b, h_j \in H : h_j.I \geq lo_j, h_j.I \leq hi_j$$

where two parallel instances of each hyperplane are defined, one bounding the tile below along that dimension and another bounding the tile from above.

Along a dimension j, dependence inhibition identifies a partner hyperplane such that the region enclosed by the partner hyperplane (h'_j) in the positive direction ($h'_j.I \geq lo'_i$ can be computed independently of the rest of the tile. This region was redundantly computed in the overlapped tiling approach.

DEFINITION 1. *The independent region along a dimension j is denoted by $\neg j$. The rest of the tile along that region will be denoted by j.*

In the subsequent discussion, it should be clear from the context whether j refers to the dimension or to the complement of the independent region along that dimension.

The region $\neg j$ is defined by making the partner hyperplane to be bounded from below along that dimension:

$$\forall I, B.I \geq b, h_k \in H, k \neq j : h_k.I \geq lo_k, h_k.I \leq hi_k$$

$$\forall I, B.I \geq b : h'_j.I \geq lo'_k, h_j.I \leq hi_j$$

Note that the hyperplanes along all the other dimensions remain unchanged.

A tile can be divided into these two regions along each of the dimensions. The various intersections of these regions divides the tile into 2^k tile components for k such dimensions. We only consider dimensions along which there is potential for dependence inhibition, which would eliminate the time dimension. For example, a tile in two-dimensional Jacobi with x and y as the dimensions can be divided into the components $\neg x \cap \neg y$, $\neg x \cap y$, $x \cap \neg y$, and $x \cap y$.

From the definition of independent region, a tile component $\neg i \cap \ldots$ is not dependent on its predecessor along dimension i. Thus, the tile component that is the intersection of the independent tile region along all the processors can be computed in parallel, without any communication — that is, all processors can start executing this in parallel, resulting in concurrent start.

Consider the tile component $i \cap \ldots$, where all other tile regions are independent. This tile component does not carry any dependence along any dimension other than i. The region in the predecessor tile that it depends on is derived as the tile-component with the same hyperplanes along all other dimensions as the tile component, with the hyperplanes along dimension i replaced by the lower-bounding hyperplane for this tile becoming the upper-bounding hyperplane, and the partner hyperplane for dependent inhibition becoming the lower-bounding hyperplane. This is the tile component $\neg i \cap \ldots$. Thus, the tile component $i \cap \ldots$ can be computed once the boundary along i computed by $\neg i \cap \ldots$ in the predecessor tile.

In general, for each dimension i along which a tile component is dependent, the inter-tile boundary is computed by the tile component in the predecessor tile obtained by replacing i by $\neg i$ For example, the tile component $x \cap y$ in two-dimensional Jacobi, can be computed after the shared boundary with $\neg x \cap y$ is received from the predecessor along x, and the one with $x \cap \neg y$ is received from the predecessor along y.

```
1. If ($n$==1), say a dimension $x$. Compute $\neg x$,
   send and receive the result along the $x$
   dimension, compute $x$ and return.
2. Execute algorithm for (n-1)-dimensional
   stencil computation for all dimensions
   except one, say z. Thus all values computed
   will be for those independent along $z$ (all
   tile sections have $\neg z$ as the z dimension
   component).
3. Send all computed values along the $z$
   dimension.
4. Execute algorithm for n-dimensional stencil
   computation for all dimensions except z.
   But this time, all values computed will be
   dependent for dependent regions along $z$.
```

Figure 9. Computation/communication scheduling algorithm for split-tiling

Figure 9 presents a scheduling algorithm with $2^n - 1$ communication steps for an n-dimensional stencil computation. In this recursive formulation, the number of communication steps is given by :

$$L(n) = 2 * L(n-1) + 1$$

with L(1)=1; that is, $L(n) = 2^n - 1$. Note that this approach does not incur any addition computation cost. In addition, only inter-tile boundaries in the spatial dimensions are communicated, thus incurring the same communication volume cost as standard tiling.

5. Code Generation

In this section, we discuss the generation of the code for the iteration space with the overlapped and split tiles. We describe the derivation of the parameters necessary to utilize the code generation framework described by Ancourt and Irigoin [5].

Each tile in the tiled iteration space is identified by a tile origin. The execution of the tiled iteration space is defined as the traversal of the tiles in terms of their origins, together with the execution of the iterations mapped to each tile as it is traversed.

The origin of the tiled iteration space defined to be the origin of the original iteration space. Given the origin, all the tile origins can be enumerated as linear combinations of the shift vectors. The tile size is defined as the distances between the tile origins along the shift vector, and is embedded in the specification of the shift vector itself.

The matrix of shift vectors specifies the traversal order of the tile origins. The shift vectors are ordered to enable an outer loop along the direction b_i so that there is parallelism-inner synchronization-outer.

Given the tile origin x_0, defined equivalently in terms of the shift vectors or as iteration points in the original iteration space, each of the hyperplanes bounding the tiles can be identified by a point in it. For hyperplanes h_i along which no overlap is identified as necessary, the iteration points x in the iteration space that form this tile satisfy the following conditions:

$$h_i.x \geq h_i.x_0 \wedge h_i.x < h_i.(x_0 + s_i)$$

Note that x_0 is a vertex on all the non-overlapped hyperplanes that form the back face of the tile. $x_0 + s_i$ is a point on the front face of the tile for all hyperplanes h_i. Since overlapping does not change the front face, this is also true for hyperplanes that utilize overlap.

When an overlapped hyperplane is identified along a dimension, we replace the back face of the original hyperplane h_i by an overlapped hyperplane h'_i. Since h'_i is constructed from h_i by only shifting it along the other hyperplanes, the point $x_0 + \sum_{j \neq i} s_i$ is a valid point on it irrespective of the choice of h'_i. Thus the boundary conditions for the tile for these hyperplanes is given by:

$$h_i.x \geq h_i.(x_0 + \sum_{j \neq i} s_i) \wedge h_i.x < h_i.(x_0 + s_i)$$

Given the tile origins and their traversals, and the shape of the overlapped tile, the code generation procedure in Ancourt and Irigoin [5] can be used to generate code. The code generated by [5] would have n outer tile space loops, each corresponding to a tiling hyperplane, and inner loops enumerating all iterations belonging to a tile. Let us assume that k hyperplanes have been identified for overlapped tiling out of the n. Overlapped tiling enables concurrent startup along a hyperplane by eliminating any inter-tile dependence along that hyperplane. Hence, the tile space loops corresponding to the remaining $n - k$ hyperplanes carry all inter-tile dependences, and can be run sequentially as the outer loops, and the k tile space loops corresponding to overlapped tiling hyperplanes can all be run in parallel by mapping to a k or lesser dimensional processor space.

The traversal of tile origins for split tiling is the same as that for standard tiling. The intra-tile code is generated for the various tile components by scanning the polytopes derived by specifying the appropriate hyperplane instances that bound the tile component, as defined earlier. The appropriate hyperplane boundaries between sub-tiles define the data to be communicated between processors for the communication phases, as discussed earlier.

6. Experimental Evaluation

Both the proposed tiling schemes—overlapped tiling and split tiling—enable load-balanced tiled execution of stencil codes that inherently satisfy the concurrent-start criterion. The degree of exploited concurrency is the same with both schemes; they differ in the computation/communication overheads relative to standard tiling. With overlapped tiling, there is a small amount of computational overhead and also a small increase in the total communication volume. Split tiling requires no additional redundant computations and requires exactly the same total communication volume as standard tiling, but requires additional messages, i.e., incurs a higher message-startup-cost overhead.

Below, we report experimental results comparing overlapped/split tiling with standard (pipelined) tiling for the 1-D Jacobi code. The experiments were conducted on a cluster consisting of 32 compute nodes each of which is a 2.8 GHz dual-processor Opteron 254 (single core) with 4GB of RAM and 1MB L2 cache, running Linux kernel 2.6.9. We used one processor per node in our experiments. The code was compiled using the Intel C Compiler with -O3 optimization flag.

The iteration space of 1-D Jacobi has a space dimension and a time dimension. Two versions of pipelined schedule were implemented: (i) one in which the processor space was mapped along the time dimension and time along the space, and (ii) the other one in which the processors were distributed in a block-cyclic fashion to execute tiles along time dimension.

First we conducted experiments to determine the optimal time tile size and space tile size for the two pipelined schedules. The experiments were conducted for 1000 time steps on 32 processors for a total problem size of 64000 elements. The execution times are shown in Figures 10 and 11. The number of communication startups decreases with an increase in the spatial tile size. This typically results in a decrease in the execution time with an increase in the space tile size. But for larger space tile sizes, the pipeline startup costs increase thus dominating and increasing the execution time. Increase in the time tile size reduces the number of time tiles and hence the number of synchronizations. But larger time tile sizes as in the case of larger space tile sizes increase the pipeline startup costs. Hence an increase in the time tile size decreases the execution time until the pipeline startup costs begin to dominate. The execution times for both the pipelined schedules, as inferred from the experiments, are minimum for a time tile size of 16 and space tile size of 1000. Hence a time tile size of 16 and space tile size of 1000 were used for subsequent evaluation of the schemes.

For overlapped and split tiling, the space tile size is fixed to be $N/nproc$, where N is the space dimension size and $nproc$ is the number of processors used for parallel execution. The time tile size is chosen to be 16 to match the choice for the pipelined schedules.

Given these choices of space and time tile sizes, the performance of the four schemes for various problem sizes is shown in Figure 12. The split and overlapped tiling schemes result in a linear increase in execution time with problem size, unlike the pipelined tiling solutions. The improvement in execution time achieved by split and overlapped tiling schemes with increase in problem size is due to the better exploitation of data locality. In addition, unlike the pipelined schedules, the communication cost is independent of the problem size.

The improved scalability of the overlapped and split tiling schemes, due to an absence of the pipeline startup cost, is shown in Figure 13. The problem size was fixed at 20000 elements per processor. The number of processors was varied to measure the weak scaling capability of the various schemes. A straight line parallel to the x-axis corresponds to linear scaling. The split tiling solution per-

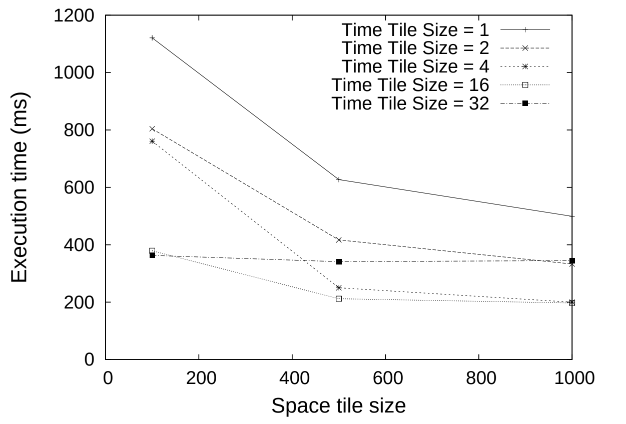

Figure 10. Optimal space and time tile size for pipelined schedule 1

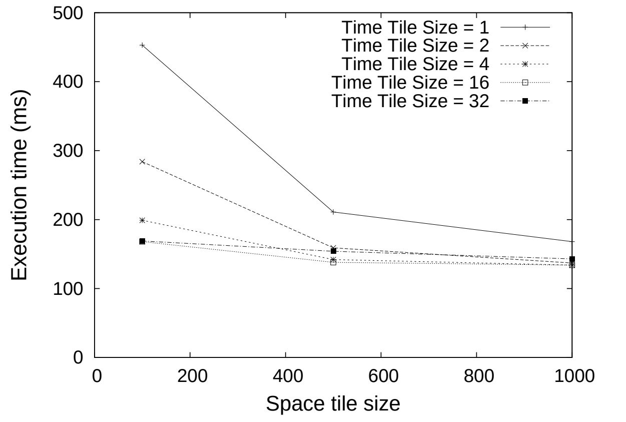

Figure 11. Optimal space and time tile size for pipelined schedule 2

forms best, followed by the overlapped tiling solution. The pipelined schedules suffer from performance degradation with increase in the number of processors.

6.1 Multi-statement stencils

We now consider multi-statement stencil codes that are representative of multimedia applications. The code is a sequence of loop nests with a producer-consumer relationship between adjacent ones as shown in Figure 14. The ‘parallel’ implementation exploits *do-all* parallelism in each loop nest with synchronization after each of the loop nests. The finite number of statements limits solutions exploiting pipelined parallelism to five processors. Figures 14 and 15 show the performance measured with overlapped and split tiling for this code. As can be seen from Figure 15, split and overlapped tiling perform better than the straightforward parallel implementation. The speedup with overlapped and split tiling is super-linear due to exploitation of data locality and enabling of concurrent start.

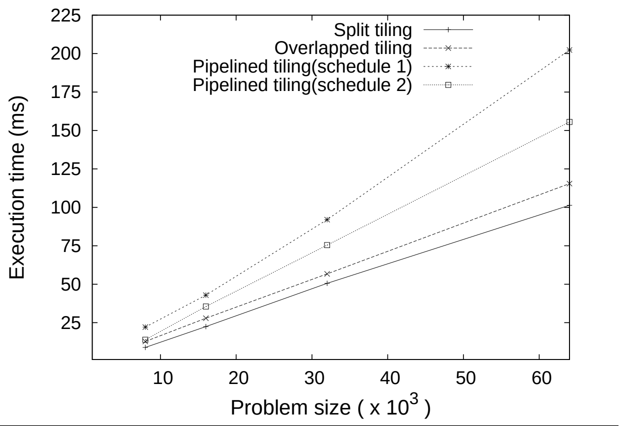

Figure 12. Execution Time as a function of problem size

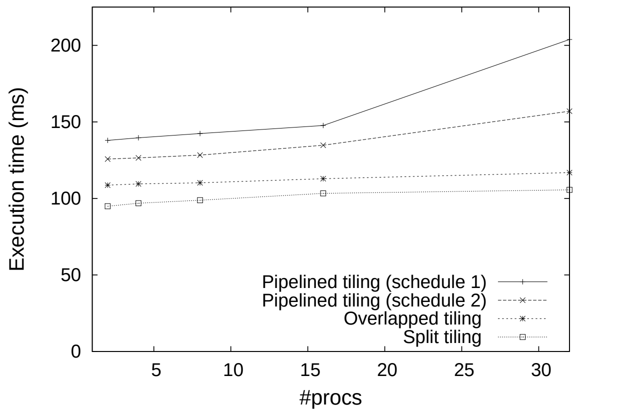

Figure 13. Execution Time for 1-D Jacobi code

```
for i=2 to N-2
    a1[i] = 0.33*(in[i-1] + in[i] + in[i+1]);

for i=3 to N-3
    a2[i] = 0.33*(a1[i-1] + a1[i] + a1[i+1]);

for i=4 to N-4
    a3[i] = 0.33*(a2[i-1] + a2[i] + a2[i+1]);

for i=5 to N-5
    a4[i] = 0.33*(a3[i-1] + a3[i] + a3[i+1]);

for i=6 to N-6
    a5[i] = 0.33*(a4[i-1] + a4[i] + a4[i+1]);
```

7. Related Work

Several recent works have presented manual optimizations and experimental studies on stencil computations [19, 18, 10]. Iteration space tiling [17, 29] is a method of aggregating a number of loop iterations into *tiles* where the tiles execute atomically; communication (or synchronization) with other processors takes place before or after the tile but not during

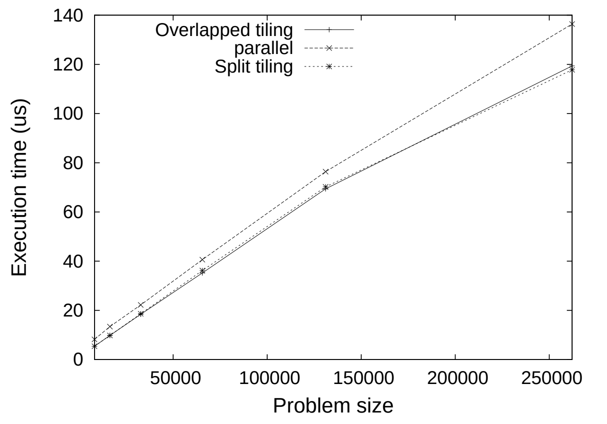

Figure 14. Multi-statement stencil: 32 processors

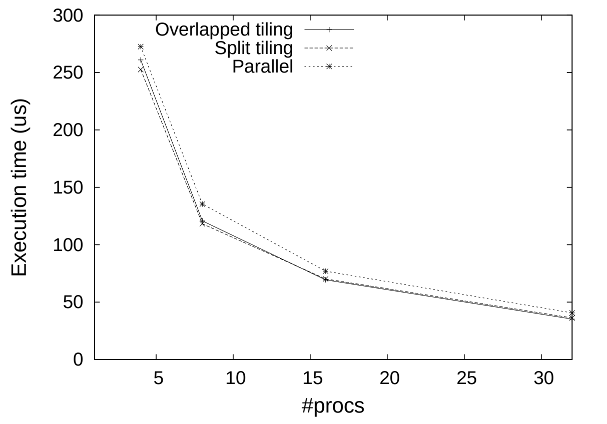

Figure 15. Multi-statement stencil: problem size = 64K

the execution of the iterations of a tile. Several works have used tiling for exploiting data locality [2, 3, 28, 26, 8]. Others have addressed the selection of tile shape and size to minimize overall execution time [25, 21, 6, 22, 14, 7]. The size of tiles has an impact on the amount of parallelism and communication: smaller tiles increase parallelism by reducing pipelined startup cost, while larger tiles reduce frequency of communication among processors. This has been studied by a number of researchers [6, 22, 14, 15, 9, 16]. Griebl [11, 12] presents an integrated framework for optimizing data locality and parallelism in the use of tiling; however, pipelining issues are not considered.

Sawdey and O'Keefe [24] describe TOPAZ the tool that explores the replicated computation of boundary values in the context of SPMD execution of stencil codes, in which the user marks regions of code to be replicated; the tool then analyzes and generates the correct code. This approach helps with reducing communication costs and improving load balance. Adve et al. [1] describe computation partitioning strategies used in the dHPF compiler that exploit replicated computation using the `LOCALIZE` directive that is available in dHPF. Both these approaches rely on user-specification of replicated computation, unlike our approach to automatic parallelization.

8. Conclusions

Iteration space tiling has received considerable attention motivated by optimizing for data locality as well as by exploiting parallelism for nested loops. The choice of the shape of iteration space tiles may result in inter-tile dependences that inhibit concurrent execution of tiles on different processors, leading to a pipelined start overhead. This paper has addressed the issue of enhancing concurrency with tiled execution of loop computations with constant dependences. Two approaches, namely *overlapped tiling* and *split tiling* were presented, that enabled the removal of inter-tile dependences, thereby enabling additional concurrency. Experimental results demonstrated the effectiveness of the proposed schemes.

Acknowledgments

This work is supported in part by the National Science Foundation through awards 0121676, 0121706, 0403342, 0508245, 0509442, and 0509467. We thank David Callahan for suggesting split tiling.

Appendix: Treatment of multi-statement iteration spaces

The characterization of the feasibility of enhanced concurrency through overlapped/split tiling is directly applicable for single-statement iteration spaces, such as the simplified (but space-wise inefficient) version of the Jacobi code of Figure 2. But the efficient version of the Jacobi code contains two distinct statements. In this Appendix, we discuss how overlapped/split tiling can be used with multi-statement iteration spaces.

Consider the Jacobi code of Figure 2. The two statements S1 and S2 are nested within the t and i loops. If we treat the entire body of the of the nested loop (i.e., S1 and S2) as the basis for defining dependences, the data dependences are (0,1), (0,2), (1,0), (1,-1), and (1,-2). Considering as before, the bounding hyperplane b corresponding to t=0, i.e., with normal vector (1,0), we find that the dependence vectors (0,1) and (0,2) have a zero dot-product with b. In other words, the point-wise concurrent start condition is not satisfied. The problem here is due to the coarse granularity used in defining dependences based on the entire loop body. Instead, it is possible to take a finer-grained view, separating out dependences due to instances of S1 and S2. There is a flow dependence (0,1) from S1 to S2, i.e., a flow dependence from S1(t,i) to S2 (t,i+1)) and anti-dependences (0,0), (0,1) and (0,2) from S1 to S2. From S2 to S1, we have flow dependences (1,0), (1,-1), and (1,-2), as well as anti-dependence (1,-1). It is clear by examining the dependences between instances of S1 and S2 (rather than the aggregate computation from S1 and S2 at each iteration space point) that all instances of S1 for a particular value of t are all concurrently executable: there are no direct dependences between them, and all incoming dependences are from instances of S2 at time-step t-1. The instances of S2 at a given time step are also concurrently executable, because the incoming dependences are all from instances of S1 at the same time-step.

Given a multi-statement iteration space for a stencil computation with statements S_1, S_2, ...S_k, we first form strongly connected components among the statements. For each strongly connected component, all self-transitive dependences are computed starting from some statement, forming all possible chains of dependences that end in an instance of the same statement. These self-transitive dependences are then used for checking for concurrent-start, instead of the single-statement dependences assumed in the treatment of Section 4. This technique allows a hyperplane-based approach, typically employed to restructure perfectly-nested loops, to be applied in this context.

Considering the Jacobi example of Figure 2, the self-transitive dependences may be computed from S1 through S2 back to S1, or vice versa (S2 through S1 to S2). The dependences from S1 to S2 are (0,0), (0,1), and (0,2), while the S2-S1 dependences are (1,0), (1,-1), and (1,-2). Forming all possible transitive dependences from S1 to S1, we get (1,-2), (1,-1), (1,0), (1,1), and (1,2). With these self-transitive dependences, it may be seen that their dot product with the t=0 boundary hyperplane (with normal (1,0)) is always positive, i.e., point-wise concurrent start is feasible for this iteration space.

References

[1] V. Adve, G. Jin, J. Mellor-Crummey, and Q. Yi. High performance fortran compilation techniques for parallelizing scientific codes. In *Proceedings of Supercomputing '98*, pages 1–23, 1998.

[2] N. Ahmed, N. Mateev, and K. Pingali. Synthesizing transformations for locality enhancement of imperfectly nested loops. In *Proceedings of ACM ICS 2000*, pages 141–152, 2000.

[3] N. Ahmed, N. Mateev, and K. Pingali. Tiling imperfectly-nested loop nests. In *Proceedings of SC'00*, page 31, 2000.

[4] N. Ahmed, N. Mateev, and K. Pingali. Synthesizing transformations for locality enhancement of imperfectly-nested loop nests. *International Journal of Parallel Programming*, 29(5), Oct. 2001.

[5] C. Ancourt and F. Irigoin. Scanning polyhedra with do loops. In *Proceedings of PPOPP '91*, pages 39–50, 1991.

[6] R. Andonov, S. Balev, S. Rajopadhye, and N. Yanev. Optimal semi-oblique tiling. *IEEE Trans. Par. & Dist. Sys.*, 14(9):944–960, 2003.

[7] P. Boulet, A. Darte, T. Risset, and Y. Robert. (Pen)-ultimate tiling? *Integration, the VLSI Journal*, 17(1):33–51, 1994.

[8] S. Coleman and K. S. McKinley. Tile size selection using cache organization and data layout. In *Proceedings of PLDI '95*, pages 279–290, 1995.

[9] F. Desprez, J. Dongarra, F. Rastello, and Y. Robert. Determining the idle time of a tiling: new results. *Journal of Information Science and Engineering*, 14:167–190, 1998.

[10] M. Frigo and V. Strumpen. The memory behavior of cache oblivious stencil computations. *Journal of Supercomputing*, 2006. to appear.

[11] M. Griebl. On tiling space-time mapped loop nests. In *Proceedings of SPAA '01*, pages 322–323, 2001.

[12] M. Griebl. *Automatic Parallelization of Loop Programs for Distributed Memory Architectures*. University of Passau, 2004. Habilitation Thesis.

[13] R. Haralick and L. Shapiro. *Computer and Robot Vision*. Addison Wesley, 1992.

[14] E. Hodzic and W. Shang. On time optimal supernode shape. *IEEE Trans. Par. & Dist. Sys.*, 13(12):1220–1233, 2002.

[15] K. Hogstedt, L. Carter, and J. Ferrante. Determining the idle time of a tiling. In *Proceedings of POPL '97*, pages 160–173, 1997.

[16] K. Hogstedt, L. Carter, and J. Ferrante. Selecting tile shape for minimal execution time. In *Proceedings of SPAA '99*, pages 201–211, 1999.

[17] F. Irigoin and R. Triolet. Supernode partitioning. In *Proceedings of POPL '88*, pages 319–329, 1988.

[18] S. Kamil, K. Datta, S. Williams, L. Oliker, J. Shalf, and K. Yelick. Implicit and explicit optimizations for stencil computations. In *Proceedings of MSPC '06*, pages 51–60, 2006.

[19] S. Kamil, P. Husbands, L. Oliker, J. Shalf, and K. Yelick. Impact of modern memory subsystems on cache optimizations for stencil computations. In *Proceedings of MSP '05*, pages 36–43, 2005.

[20] W. Kelly, W. Pugh, and E. Rosser. Code generation for multiple mappings. In *Proceedings of FRONTIERS '95*, page 332, 1995.

[21] J. Ramanujam and P. Sadayappan. Tiling multidimensional iteration spaces for nonshared memory machines. In *Proceedings of Supercomputing '91*, pages 111–120, 1991.

[22] L. Renganarayana and S. Rajopadhye. A geometric programming framework for optimal multi-level tiling. In *Proceedings of SC '04*, page 18, 2004.

[23] A. Sawdey, M. O'Keefe, and R. Bleck. The design, implementation, and performance of a parallel ocean circulation model. In *Proceedings of 6th ECMWF Workshop on the Use of Parallel Processors in Meteorology: Coming of Age*, pages 523–550, 1995.

[24] A. Sawdey and M. T. O'Keefe. Program analysis of overlap area usage in self-similar parallel programs. In *Proceedings of LCPC '97*, pages 79–93, 1998.

[25] R. Schreiber and J. Dongarra. Automatic blocking of nested loops. Technical report, University of Tennessee, Knoxville, TN, Aug. 1990.

[26] Y. Song and Z. Li. New tiling techniques to improve cache temporal locality. In *Proceedings of PLDI '99*, pages 215–228, 1999.

[27] A. Taflove and S. C. Hagness. *Computational Electrodynamics: The Finite-Difference Time-Domain Method, Third Edition*. Artech House Publishers, 2005.

[28] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proceedings of PLDI '91*, pages 30–44, 1991.

[29] M. Wolfe. More iteration space tiling. In *Proceedings of Supercomputing '89*, pages 655–664, 1989.