Towards Optimal Multi-level Tiling for Stencil Computations

Lakshminarayanan Renganarayana, Manjukumar Harthikote-Matha,
Rinku Dewri, and Sanjay Rajopadhye
Department of Computer Science, Colorado State University, USA
{ln,manjuhmm,rinku}@cs.colostate.edu and Sanjay.Rajopadhye@colostate.edu

Abstract

Stencil computations form the performance-critical core of many applications. Tiling and parallelization are two important optimizations to speed up stencil computations. Many tiling and parallelization strategies are applicable to a given stencil computation. The best strategy depends not only on the combination of the two techniques, but also on many parameters: tile and loop sizes in each dimension; computation-communication balance of the code; processor architecture; message startup costs; etc. The best choices can only be determined through design-space exploration, which is extremely tedious and error prone to do via exhaustive experimentation. We characterize the space of multi-level tilings and parallelizations for 2D/3D Gauss-Siedel stencil computation. A systematic exploration of a part of this space enabled us to derive a design which is up to a factor of two faster than the standard implementation.

1. Introduction

Stencil computations form the basis for a wide range of scientific applications from simple Jacobi to complex multigrid solvers. Their inclusion in major benchmarks like SPEC [25], HPFBENCH [9], PARKBENCH [19], and NAS Parallel Benchmarks [18], clearly show their importance. The development of special purpose stencil compilers [4] and implementation of pattern matchers in general compilers [24] to identify stencil computations, highlight the potential for performance improvements from loop transformations and optimizations.

Tiling [10, 31, 32] is a loop transformation that can be used for (i) partitioning data and computations among parallel processors and (ii) reordering computations within a single processor to improve data locality. For stencil computations a variety of multi-level tiling schemes are possible. For example, consider just two levels of tiling: an outer level for parallelism and an inner level for data locality. For every outer level tiling strategy, many parallelizations are possible, and for each such parallelization, several inner level (for locality) tiling strategies are possible. The best schemes are those with lowest execution times, which depend on optimal choices of tiling and parallelization strategies and parameters. Not only are there many such schemes, for each of them the space of the tile sizes is also huge. The global question is *which combination of tiling and parallelization strategy with which parameters produces the minimum running time for a given set of program size parameters and a given parallel machine?* It is time consuming and error prone to develop parallel implementations for each combination of tiling and parallelization scheme and experiment with them to find a good one, or to even eliminate the obviously poor ones.

There have been extensive studies [27, 23, 14, 30, 6, 11] on tiling stencil computations for locality. Schemes for tiling stencil computations for parallelism can be classified based on whether or not they tile the outermost time loop. The commonly used data partitioning scheme [7] does not tile the time loop and uses the "owner-computes" rule to determine the computation distribution. Early work by Wolfe [28] shows that skewing can be used to enable tiling of the time loops. Recently, Wonnacott [29] shows that time skewing can be used to tile for parallelism as well as locality. Several important issues are not addressed by these authors. For a given stencil computation,

- what is the space of legal tiling and parallelization schemes?
- what are the trade-offs between these schemes?
- how do the tiling choices at the parallelization level affect the choices at locality[1]?
- what are the globally optimal tile sizes?

1-4244-0910-1/07/$20.00

[1] Mitchell *et al.* [16] point out that ignoring such tiling interactions will lead to suboptimal solutions.

A study of these issues will enable us to develop high performance, multi-version, platform specific implementations of stencil computations. As an analogy, consider the matrix multiplication code generated by ATLAS [26]. The generated final code has different versions for different shapes of matrices, and makes several platform specific choices for optimizations. Our experiments show that stencil computations are similar, i.e., the optimal strategy depends on the shape of the domain (size of the grid and the number of time steps). We envision a tool that explores the space of legal tiling and parallelization schemes, selects optimal parameters and generates a multi-version high performance implementation of a given stencil computation. As a first step towards such a tool, this paper makes the following contributions.

- We characterize the space of possible legal tilings and load balanced parallelizations for 2D/3D Gauss-Siedel 9-point stencil. We focus on two candidates from this space to illustrate the need to explore this space. Even this partial exploration led us to derive a new strategy which is up to a factor of two faster than the standard implementation.
- We develop analytical models for the parallel execution times of the two strategies. We formulate a constrained optimization problem for the optimal tile sizes and transform it to a convex optimization problem, which can be solved efficiently.
- For both the strategies, we study an additional level of tiling for locality and analyze the interactions between the choices at different levels.
- We experimentally validate our analytical models. We discuss the performance improvements and trade-offs obtained with various strategies. We show how the best strategy depends on the shape of the stencil iteration space. This leads to a division of the input space into regions where different strategies perform better.

In the next section we characterize the space of legal tilings and parallelizations. In Sections 3 and 4 we discuss in detail the tilings and parallelizations for the two strategies and derive analytical models for their execution times. We present experimental validation and discuss performance improvements and trade-offs in Section 5. We discuss related work in Section 6 and present our conclusions and future work in Section 7.

2. Space of Tiling and Parallelizations

We consider 2D/3D stencil computations in which a two dimensional data grid of size $N_i \times N_j$ is updated iteratively over N_k time steps. We call N_i, N_j, and N_k as the loop size parameters and let $\vec{N} = (N_i, N_j, N_k)$. As a representative of this class (3D stencils) we consider the Gauss-Siedel 9 point stencil computation given in Figure 1 (left). The computation domain is a 3D cuboid of size $N_i \times N_j \times N_k$. A graphical view of the nine dependences are shown in Figure 1 (right). Gauss-Siedel (in place updates) stencils are expected to have faster convergence than the Jacobi stencils, which use all the 9 values from previous time steps. On the other hand, the dependences of the Jacobi stencil are easier to tile and/ or parallelize. We consider the difficult (to tile and parallelize) but faster converging Gauss-Siedel stencils. Our characterization and models are directly adaptable and applicable to other types of 2D/3D stencils.

2.1. Tiling and parallelization model

Tiling [10, 31] partitions the iteration space into groups which are executed in an atomic fashion – all iterations in a given tile are executed by a processor before any iteration of its next tile. Note that this notion of atomicity still permits any legal (re)ordering of the computation and communication steps within a tile. A *rectangular tiling* is one where rectangles are used for partitioning. We consider rectangular tiling possibly preceded by a skewing transformation to make it legal. We denote the tile sizes along the dimensions i, j, and k of the 3D iteration space by $s_i, s_j,$ and s_k, respectively. The *tile graph* consists of nodes representing tiles and edges between them representing the dependences between tiles. It is well known that [1, 32] if the s_i's are large as compared to the elements of the dependence vectors of the original loop, then the dependencies between the tiles are *unit vectors* (or binary combinations thereof, which can be neglected for analysis purposes without loss of generality). An important property is that the tile graph with such unit dependence vectors can be viewed as an n-dimensional system of uniform recurrence equations [12]. Such a view allows us to use the powerful systolic array synthesis methods [20, 21] to formally reason about optimal parallelizations of the tile graph. In the context of exploring the space of possible tiling and parallelizations, such a formal reasoning helps in constraining the search space to a few valid and good candidates.

In any parallelization, the dependences in the tile graph induce some delay before which all the processors can start executing. We call this initial delay the *latency* of a parallelization strategy. Once all the processors begin to execute, any idle time incurred by a processor is a consequence of the chosen parallelization. We restrict ourselves to parallelizations that are free of such idle times. We call such parallelizations *idle-free.* We also restrict ourselves to allocations that are *load-balanced,* i.e., to ones that allocate an equal amount (except at boundaries) of computation to

```
for k = 1 ... N_k
  for i = 1 ... N_i
    for j = 1 ... N_j
      A[i,j] = ω ⋆ (A[i−1,j] + A[i−1,j−1]+
                    A[i,j−1] + A[i−1,j+1]+
                    A[i+1,j] + A[i+1,j+1]+
                    A[i,j] + A[i,j+1] + A[i+1,j−1])
```

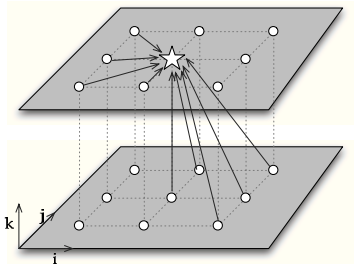


Figure 1. (Left) Gauss-Siedel style successive over-relaxation code. 9 point stencil computation. (Right) Dependences of the 9 point stencil computation.

every processor. For the stencil computations this can always be achieved. Practical experience as well as our analytical models predict that optimal performance can only be achieved under such idle free load balanced parallelizations. Further, for allocation functions we restrict to orthogonal projections—ones that are parallel to some canonical axes.

To summarize, we consider rectangular tiling and idle-free load balanced parallelizations only. As shown in the later sections, the set of choices to be considered after these restrictions is still rich.

2.2. Need for and implications of skewing

Skewing is a loop transformation that changes the dependence distances in the stencil code. In the context of stencil computations, skewing is often used to transform the dependence distances into non-negative ones, thus making tiling legal. Given the dependences of the 9-pt stencil (*cf.* Figure 1), tiling certain dimensions require certain skewing transformations to make it legal. However, as a side effect, skewing also changes the shape of the iteration space. For instance, skewing a rectangular iteration space will make it a parallelogram. As a consequence, a rectangular tiling of the parallelogram iteration space will result in both full (rectangular) and partial (non-rectangular) tiles. Partial tiles increase the tiling overhead and also makes analytical modeling difficult. Hence, there is a trade-off: extra tiling overhead introduced by skewing versus ability to tile additional dimensions.

2.3. Space of tilings and allocations for parallelization

The space of possible rectangular tilings for the 9-pt stencil (*cf.* Figure 1) corresponds to the choice of which and how many dimensions do we choose to tile. Note that we are not characterizing the space of tile sizes, which we will do later for each possible tiling. Tiling different dimensions requires a different set of skews of the iteration space. The choices and the corresponding skews are discussed below and a graphical view of them is shown in Figure 2 (top box).

1. **Tile the program with *no skewing*.** The program dependences limit such tilings. For example, in order to tile either the i or the j loops, s_k has to be 1. Furthermore, in order to tile the j loop, s_i must also be 1. Thus the possible tilings are: (i) the trivial $1 \times 1 \times 1$, $N_k \times N_i \times N_j$ and $1 \times N_i \times N_j$ tiles which we discarded for obvious reasons; (ii) $1 \times s_i \times N_j$ tiles; and (iii) $1 \times 1 \times s_j$ tiles, which we discard because the computation-to-communication balance of the tiles is too low[2]. We pursue the $1 \times s_i \times N_j$ tiling. For this tiling strategy, a parallelization on an 1D processor array is the only choice, where the processors are aligned along the i axis (*cf.* Figure 3 (left)). The choices are shown in the left-most branch of Figure 2.

2. **Tiling both i and j dimensions.** We need to skew i with respect to j to make tiling along j legal. This case also covers the case of tiling just along j when the tile size along i is 1. For this tiling, we can parallelize the tile graph on an 2D or 1D processor array. For an 1D processor array we align it along the i axis. For the 2D processor array, we can either align the processors along the ik-plane or the ij-plane. For the ij-plane alignment the processors are arranged in a parallelogram shaped grid and for the ik-plane alignment they are arranged in a rectangular grid. The choices are shown in the second branch (from left) in Figure 2.

3. **Tiling both i and k dimensions.** We need to skew i with respect to k to make tiling along k valid. Based on a similar reasoning as above, this choice also covers

[2]It would be easy to use the ideas in this paper to confirm analytically and experimentally that this is indeed the case.

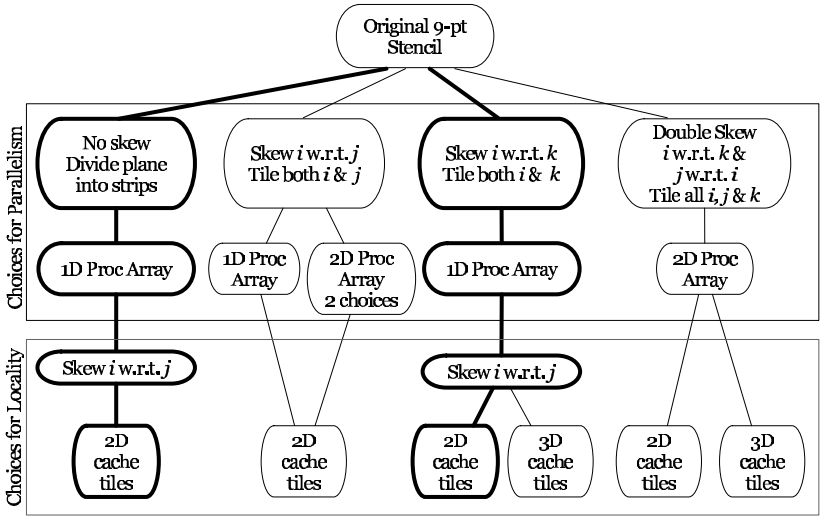


Figure 2. Space of multi-level tilings and parallelizations for the 9-pt. stencil. The choices (path) shown in bold correspond to the two strategies explored in detail.

the tiling just along k. For this tiling, a parallelization on an 1D processor array is the only choice. The processors are aligned along the k axis as shown in Figure 4 (right). The choices for this strategy are shown in the third branch (from left) in Figure 2.

4. **Tiling all the three (i, j and k) dimensions.** We need to first skew i with respect to k and then skew j with respect to i. This choice also covers the case of tiling just j and k. For this tiling, with orthogonal processor allocations, only an 2D processor array is possible. The 2D processor array is aligned along the ik-plane. However, if we expand our space and consider non-orthogonal processor allocations, there are two linear array parallelizations possible[3]. We do not discuss these choices further. The choices related to the 2D processor array is shown in the right most branch in Figure 2.

One might wonder why the last choice above does not cover all the other cases by appropriately letting the corresponding dimensions (i, j, and/or k) equal to 1? The answer is, skewing creates partial tiles and leads to a different cost function for the total computation time. The overhead of partial tiles should be avoided whenever possible, so that we can derive simpler parallel implementations and more precise execution time models.

[3]These non-orthogonal projections make every communication non-local, which would result in higher communication costs. Based on this intuition we have restricted ourselves to orthogonal projections. It is not clear whether this is always globally optimal but it definitely makes the space more tractable.

2.4. Space of tilings for locality

After an outer level of tiling for parallelism we can tile another level for locality. We call a tile from the outer level of tiling (for parallelism) as a *parallel-tile* and a tile from the inner level of tiling (for locality) a *cache-tile.* Correspondingly we also refer to their sizes as *parallel-tile sizes* and *cache-tile sizes.*

We have two choices for cache tiling: tile i and j dimensions only or tile i, j and k dimensions. Both may require additional skewing transformations to make them legal. This additional skewing is not required if it has already been done for the outer (parallelism) level tiling. Given that the data of the stencil is 2D, tiling just the i and j dimensions will allow us to exploit the limited amount of spatial locality. To exploit temporal locality we need to tile the (time) k dimension. The choices are shown in Figure 2 (lower box).

2.5. Interactions between tilings

Two types of interactions ensue, *viz.*, (i) skewing transformations at parallelism level can enable or disable tiling along certain dimensions for locality, and (ii) the parallel tile sizes restrict the lower and upper bounds of the cache tile sizes. These interactions stem from the fact that a parallel-tile becomes the iteration space for the cache tiling.

We describe below two instances where a decision made at the parallelism level affects the choices in the inner level.

Consider the case in which we do not tile the k loop at the parallelism level. This choice leads to parallel-tiles which are slices of the ij-plane and disables cache level tiling of the k loop. These slices become the iteration space for the cache-level tiling. So, we can see that the cache-tiles are forced to be 2 dimensional and hence can only exploit spatial locality. (Recall that we need to also tile the k dimension to exploit temporal locality.) The first two branches (from left) in Figure 2 shows this consequence — observe the leaves showing the possibility of only 2D cache tiles.

When tiling j loop is made legal by skewing transformations at the outer level, there is no additional skewing required at the inner cache-level to get 2D cache-tiles. This case is shown in the second and fourth branches (from left) in Figure 2. On the other hand, notice that for the strategies shown in the first and third branches (from left) in Figure 2, we need to skew i loop with respect to j to make tiling j loop legal and hence get 2D cache-tiles.

3. 1D Strips

In this section we consider the first strategy (left most branch in Figure 2) and develop an analytical model for the parallel execution time. In the modeling, we use three parameters, *viz.*, α, β and τ, to model the quantities that are dependent on the loop program and parallel architecture on which it is to be executed. α represents the time to execute an iteration of the given loop program. β and τ represent respectively the start up cost of a MPI communication call and the time to communicate a `double` precision data value. We use an affine communication model, *viz.,* $\tau x + \beta$, to estimate the cost of communicating a message of size x.

In this tiling strategy, each tile is a $1 \times s_i \times N_j$ rectangular parallelepiped, i.e., only the i loop is effectively tiled. The j loop has a single "tile" of size N_j, and the k loop "tiles" have unit size. Because there is only one tile in the j dimension, the resulting tile graph can be viewed as a 2D grid in the (i, k) plane as shown in Figure 3 (left). The dependence between a producer tile at (p_i, p_j) and a consumer tile at (c_i, c_j) is given by $(c_i - p_i, c_j - p_j)$. The dependences between tiles are $(0, 1)$ to the north, with a data "volume" of $N_j s_i$, and $(1, 0)$ (east) and $(-1, 1)$ (north-west), both with volume N_j.

To explore different parallelizations, we first derive the optimal wavefront schedule for the tile graph, which is $t(i, k) = i + 2k$. This schedule is shown as dotted lines across the tile graph in Figure 3 (left). It is optimal in the sense that the total execution time for this schedule (assuming unbounded processors) is $\frac{N_i}{s_i} + 2N_k - 1$, which equals the length of the longest path in the graph.

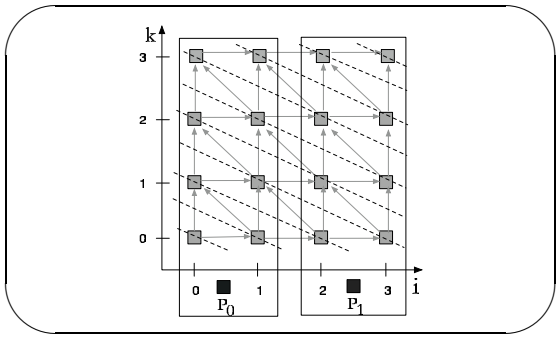
Next, we choose an appropriate allocation of tiles to (virtual) processors. For our rectangular tile graph, only two allocations lead to a load balanced parallelization, namely by columns, or by rows. Allocation by rows, where each processor sequentially executes all the tiles in a row of the tile graph, leads to a parallelization that allows multiple passes. We developed an analytical model for it and determined the optimal tile size. However, this parallelization is almost always outperformed by the column wise allocation, and is not described further in the interests of brevity.

Allocation by columns, where tile (i, k) is performed by virtual processor i, yields a parallelization (i.e., a "macro systolic array") that has bidirectional communication: processor i sends to $i + 1$ for the $(1, 0)$ dependence, and to $i - 1$ for the $(-1, 1)$ dependence. This has two important consequences.

- Every processor is active only on alternate time steps. This problem can easily be corrected by a well known systolic technique called clustering or serialization [5]. We allocate two adjacent virtual processors to a single physical processor which alternates between the two tiles and is thus always busy. This combined two-tile unit is called a "macro tile" or a *Strip*.
- It precludes adaptation to run on fewer processors in multiple passes, using another common systolic technique called LPGS (for Locally Parallel Globally Sequential) partitioning [17]. This means that $s_i = \frac{N_i}{2p}$, i.e., each macro tile is a $\frac{N_i}{p} \times N_j$ strip.

A processor performs the following steps: receive data required to execute the strip, execute the strip and send computed data to neighbors. As a latency hiding optimization, we can relax the strict order between the receive-compute-send steps, and interleave them. In the execution of a strip, if we allow the processors to move the sends as early as possible and the receives as late as possible, we get the optimized code shown in Figure 3 (right). In this version, a processor receives the data to compute its left-most column, computes it and sends the new values immediately. Then it computes the middle region of the strip, receives the data to compute the right-most column, computes it, and sends the new values.

There are no tiling parameters to choose optimally, and the analytical model developed below predicts the running time for this parallelization. The single pass execution implies that the tile size s_i along i has to be $\frac{N_i}{P}$. Let p_i denote the i^{th} processor, and $p' = p_{P-1}$ the last processor. The total execution time of this tiling and parallelization can be modeled as $T_{\text{strip}} = \mathsf{Latency}(p') + \mathsf{TPP}(p) \times \mathsf{TET}(s_i)$. Where, P is the number of processors, $s_i = \frac{N_i}{P}$ is the tile size, $\mathsf{Latency}(p')$ is the latency for last processor to start, $\mathsf{TPP}(p)$ is the number of tiles allocated to any processor




```
for k = 1 ... N_k
  for each strip S
     receive Rcol[k] of p_{i-1}
     compute Lcol[k] of S
     send Lcol[k] of S to p_{i-1}
     compute MiddleRegion[k] of S
     receive Lcol[k-1] of p_{i+1}
     compute Rcol[k] of S
     send Rcol[k] of S to p_{i+1}
```

Figure 3. (Left) Tile graph of 1D strips tiling. The fastest schedule is shown in dotted lines. (Right) Steps performed by each (non-boundary) processor in 1D Strips tiling. `Lcol[]`, `Rcol[]`, and `MiddleRegion[]` corresponds to the left column, right column and middle portion of a strip. The index k and $k-1$ indicates, respectively, whether they are from the same k plane or the previous plane.

p, and $\mathsf{TET}(s_i)$ is the time to compute a tile. To compute the time to execute a tile, we observe that during the computation of a tile a processor performs $N_j s_i$ computations and communicates its left and right columns, of size N_j, to the previous and next processors. Hence, we have $\mathsf{TET}(s_i) = \alpha \times N_j \times s_i + 4(\tau N_j + \beta)$, where, α, τ, and β are as discussed earlier. Every processor is allocated N_k macro tiles (or strips), hence $\mathsf{TPP}(p) = N_k$.

The last processor can only start after it receives the right most column of its left neighbor, *i.e.,* p' can start after the first $P-1$ processors execute their tiles. Hence, $\mathsf{Latency}(p') = (P-1) \times \mathsf{TET}(s_i)$. By plugging in these functions we get

$$T_{\text{strip}} = (N_k + P - 1) \times \left(\alpha \left(\frac{N_i N_j}{P}\right) + 4(\tau N_j + \beta)\right) \quad (1)$$

3.1. Cache tiling

Each processor executes a set of parallel-tiles, each of size $\frac{N_i}{P} \times N_j$. This strip can be further tiled to exploit some limited amount of spatial locality. However, to make the tiling of the strip legal, we need to skew the j loop with respect to the i loop. We perform this transformation and then tile both the i and the j loop to obtain 2D cache-tiles. Note that the decision of not tiling the k loop at the outer level results in a situation where we cannot tile the k loop at the inner (cache) level, to exploit temporal locality. We select the best cache-tile sizes empirically, *i.e.,* by running the cache-tiled code for several tile sizes and selecting the best. The space of tile sizes is chosen such that the tile footprint is smaller than the cache capacity.

4. Semi-oblique Strips

In this tiling strategy, we seek to tile the k and i dimensions. To make this tiling legal we first skew the i loop with respect to the k loop with the transformation $(k, i, j) \mapsto (k, i+k, j)$. The transformed dependences are shown in Figure 4 (left). We then tile the k and i loops with tile sizes s_k and s_i, respectively. We do not tile the j loop and allow $s_j = N_j$. The skewed iteration space together with the tiling is shown in Figure 4 (right).

We parallelize this tiled iteration space on a linear array of P processors aligned along the k axis as shown in Figure 4 (left). The target architecture is a distributed memory machine and the execution model is SPMD style with MPI based communication. Note that depending on whether $\frac{N_k}{s_k} > P$ or not, there might be more than one pass. Every processor executes one or more rows (along i) of $\frac{N_i}{s_i}$ tiles of size $s_k \times s_i \times N_j$. Such an allocation is load balanced—all the processors execute the same amount of computation (assuming P divides $\frac{N_k}{s_k}$ evenly). During the execution of a tile, a processor receives the bottom ($i\,j$) face of the tile from the processor below it, computes the tile, and sends the top ($i\,j$) face to the processor above it. These faces communicated between processors are of size $s_i N_j$.

The execution time of the tiled parallelized loop program is given by $T_{\text{SOS}} = \mathsf{Latency}(p_{P-1}) + \mathsf{TPP}(p) \times \mathsf{TET}(s_i, s_k)$, where we have $\mathsf{TET}(s_i, s_k) = (\alpha s_i s_k N_j + 2(\tau N_j s_i + \beta))$. The number of tiles allocated to a processor is $\mathsf{TPP}(p) = \frac{N_k}{s_k P} \times \frac{N_i + s_k}{s_i}$. This follows from the fact that $\frac{N_k}{s_k P}$ gives the number of passes executed by a processor and $\frac{N_i + s_k}{s_i}$ is the number of tiles executed by a processor in one pass.

The slope $\frac{s_k}{s_i}$ (also known as the *rise* [8]) plays a fun-

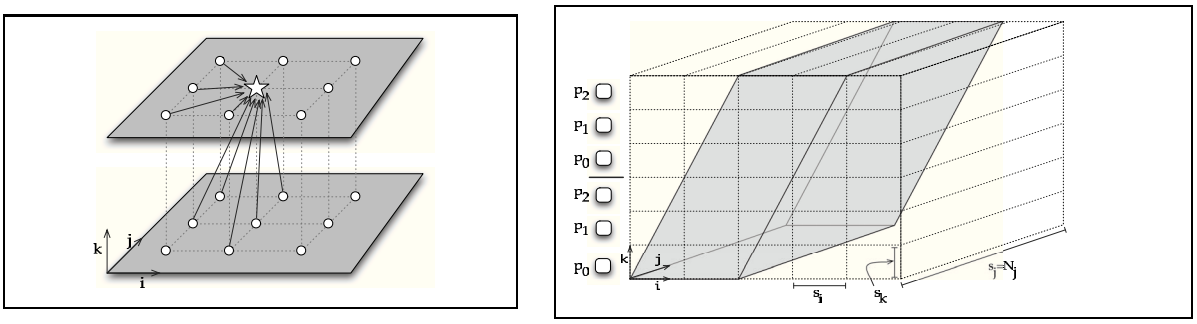


Figure 4. (Left) Skewed dependences that make this tiling legal. (Right) Semi-oblique strips tiling.

damental role in determining the latency. Processor p_{P-1} can start its first tile only after $(P-1) \times \left(\frac{s_k}{s_i}+1\right)$ tiles are executed. Hence, we have $\mathsf{Latency}(p_{P-1}) = (P-1) \times \left(\frac{s_k}{s_i}+1\right) \times \mathsf{TET}(s_i, s_k)$. To ensure that there is no idle time between the passes, we need to make sure that by the time the first processor finishes all the tiles from its first pass, the last processor should have finished at least one tile. This constraint is given by $P\left(\frac{s_k}{s_i}+1\right) \leq \frac{N_i+s_k}{s_i}$. Putting them all together we get the following constrained optimization problem

$$\begin{aligned} \text{minimize} \quad T_{\text{SOS}} = \; & \left[\left(\frac{N_k}{s_k P} \times \frac{N_i+s_k}{s_i}\right) + (P-1) \times \left(\frac{s_k}{s_i}+1\right)\right] \times \\ & (\alpha s_i s_k N_j + 2(\tau N_j s_i + \beta)) \qquad (2) \\ \text{subject to} \quad & P\left(\frac{s_k}{s_i}+1\right) \leq \frac{N_i+s_k}{s_i} \end{aligned}$$

We can transform this optimization problem (Eqn. 2) into a Geometric Program (GP), which is a special class of optimization problems. The key insight that permits this transformation is the property that the tile sizes always take positive values only. An introduction of GPs is beyond the scope of this paper. We refer the interested reader to [22] which shows the use of GPs to solve optimal tiling problems. Geometric programs can be transformed into convex optimization problems using a variable substitution [3] and solved efficiently using polynomial time interior point methods [13]. Integers solutions can be found by using a branch-and-bound algorithm. We use YALMIP [15] – a tool that provides an high level symbolic interface in MATLAB to define and solve GPs for integer solutions. The number of (tile) variables of our GPs are related to number of dimensions tiled and hence are often small. In our experience with solving GPs related to tiling, the integer solutions were found in a few (less than ten) iterations of the branch-and-bound algorithm. The (wall clock) running time of this algorithm was just a few seconds, even with the overhead of using the symbolic MATLAB interface.

4.1. Cache tiling

Each parallel-tile is a semi-oblique block of size $s_k \times s_i \times N_j$. Each block can be further tiled for locality. Since we have tiled the k loop at the outer level we can have both 2D as well as 3D tiles at the inner (cache) level. To make tiling the j loop legal, we have to skew it with respect to the i loop. After this transformation, we can either choose to tile only the i and j loop to obtain 2D cache-tiles, which can exploit spatial locality, or tile all the three loops to obtain 3D cache-tiles and exploit both spatial and temporal locality. We explore only the choice of 2D cache-tiles and leave 3D cache-tiles as future work. Note that the optimal parallel-tile size s_i from the outer level affects the iteration space sizes for the 2D cache-tiling, *viz.*, $s_i \times N_j$. We select the best cache-tile sizes empirically, *i.e.,* by running the cache-tiled coded for several tile sizes, which are within the bounds of s_i and N_j, and selecting the best.

5. Experimental Results

We have implemented two versions, one with cache tiling and one without, for both the 1D Strips and Semi Oblique Strip (SOS) strategies. The implementation of the 1D Strips is the optimized latency hiding version. For both the strategies, we selected the best cache-tile sizes by running the cache-tiled code (on a single processor) for a range of tile sizes (within the bounds imposed by parallel-tile sizes). For the 1D Strips, we observed that for the small tile sizes range, there is a steep decrease in the running time as we increase the tile sizes. This trend stops and the running time becomes relatively constant for larger tile sizes. After experimenting with several strip sizes, we found the cache-tile of size 60×140 to be the best and used it for our experiments. For the SOS strategy, the running time had a similar behavior with respect to cache-tile sizes. After experimenting with several grid sizes, we found that the optimal parallel-tile size s_i is always very small, and hence we let the cache-tile size along the i dimension to be the same as s_i. This results

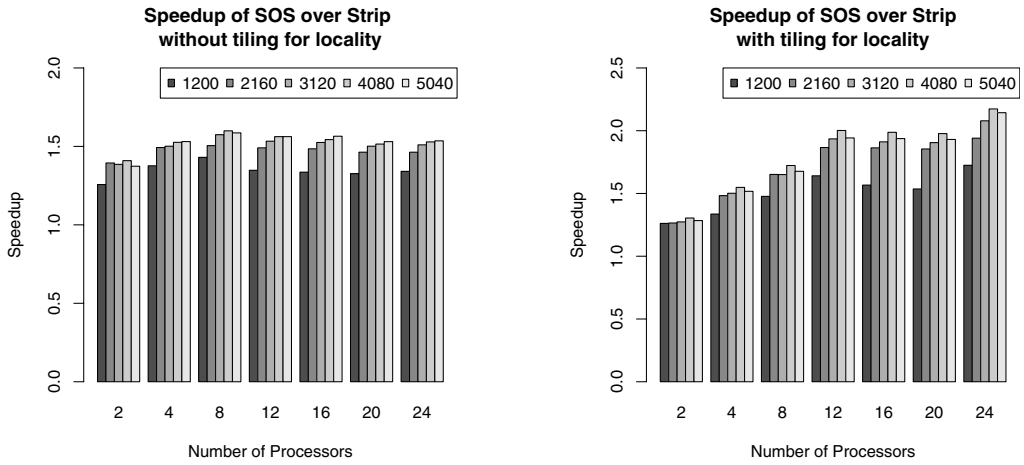


Figure 5. Speedups for SOS over Strip strategy without (left) and with (right) cache tiling. Results for five different grid sizes $N_i = N_j = 1200, 2160, 3120, 4080,$ and 5040, each for a set of small time steps $N_k = P$ (the number of processors), are shown.

in no cache-tiling along i. For the cache-tile size along j we selected a value of 50 which belongs to the flat execution time region.

We used a IBM Cluster 1600 running AIX, at the National Center for Atmospheric Research, Colorado, for our experiments. The IBM Cluster is a Symmetric Multiprocessing (SMP) system. The nodes are made of 1.3-GHz POWER4 processors. The processors in a single node can communicate via shared memory, and the nodes themselves communicate via an SP Switch2 interconnect. We used the IBM `mpcc` compiler for our experiments with standard `-O3` optimization levels. Our parallel implementations are written using the MPI message passing library.

We obtained values of $\alpha = 5.5 \times 10^{-8}, \beta = 4.1 \times 10^{-6}$ and $\tau = 5.3 \times 10^{-9}$ as follows. We ran the loop body of the stencil computation for 1000 iterations and took the average execution time as α – the time to compute one iteration. We estimated the cost of communicating one double value (τ), and the `MPI` communication call start up cost (β), with a ping-pong style MPI program that mimics the communication pattern of our tiled programs.

Stencil computations used in PDE solvers have fast convergence and the number of time steps are usually small, such as $8, 16,$ or 24. The type of stencil computations used in simulations, such as water models, have large number of time steps. For our experiments we considered these two type of stencils over square grids, *i.e.,* $N_i = N_j$. We found that for small time step stencils the SOS strategy performs better than Strips, and as the number of time steps increases, its performance becomes comparable to that of Strips. This divides the input space into two regions where one of the strategies is clearly preferable over the other.

We present experimental validation and performance improvements for the small number of time steps case. Five different square ($N_i = N_j$) grid sizes *viz.*, $1200, 2160, 3120, 4080,$ and 5040, with small time steps N_k were used for experiments. For such small time steps in the SOS strategy, the number of processors is set to N_k, since the processors are aligned along the k dimension. Figure 5 shows the speedup achieved by the SOS over the Strip strategy (without and with tiling for locality) for the five different grid sizes with number of processors $P = N_k$. Without cache tiling we obtained speedups of up to 60% (with an average of 40%). Also, we observed higher speedups for larger grids. We observed in single processor experiments that cache tiling helps SOS (30%) more than Strips (5%). These improvements are reflected in their parallel implementations (Figure 5(right)). Speedups up to a factor of 2.1 are seen with cache tiling (see $P = 24$ in Figure 5(right)). Here, we see a similar trend of higher speedups for larger grid sizes. Clearly, for stencils with small time steps, the new SOS strategy performs much better than the standard Strips strategy. A two fold decrease in running time is significant for such applications.

We validated our analytical models for the two strategies, using more than 100 different combinations of stencil grid sizes and number of processors, and have found them to be reasonably accurate. We present here a subset of them. The percentage error in predicted with respect to the actual for SOS is shown in Figure 6(left) and for Strips in Figure 6

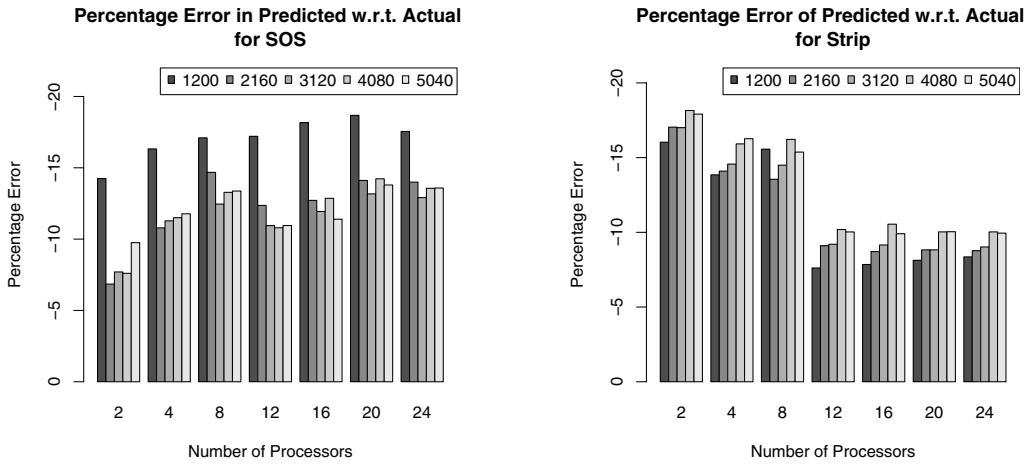


Figure 6. Percentage error in predicted with respected to actual for SOS (Left) and Strip (Right) strategies without cache tiling. Results are reported for five different grid sizes ($N_i = N_j$) each for a set of time steps N_k equal to number of processors P.

(right). Our models consistently under predict the execution time. Overall, the predictions are within 20% of the actual execution time, which is good for tiling and design space exploration. Further, for SOS, we conducted experiments to see how close is the predicted to the actual at the optimal tile sizes (s_i and s_k), obtained by solving the constrained optimization problem (*cf.* Eqn. 2). We found that near the optimal running time the predictions are fairly close (within 20%) and at points far from optimum the difference is higher. This is the desired behavior for such analytical models.

6. Related Work

There have been extensive studies [27, 23, 14, 30, 6, 11] on tiling stencil computations for locality. In the space of multi-level tilings that we characterize, these locality improving techniques can be leveraged to improve the implementations at the leaf (uni-processor) level. Data or domain decomposition [7] is a standard scheme used in tiling stencil computations for parallelism. Early work by Wolfe [28] shows that skewing and tiling transformations can be combined to tile for both parallelism and locality. Recently, Wonnacott [29] shows that time skewing can be used to tile for parallelism as well as locality. As discussed in Section 1, these authors propose transformations that can enable and make tiling beneficial for parallelism and locality. We characterize the space of possible multi-level tilings and parallelizations with the goal of systematically deriving the best implementation for a given stencil.

Andonov *et al.* [1] consider 2D (1D data and 2D iteration space) computations and propose an analytical model similar to ours for estimating the execution time of a tiled program and present analytical closed form solutions for the optimal tile sizes and the number of processors. We consider an 3D iteration space and characterize the possible multi-level tilings and parallelizations. Our analytical BSP style cost models are inspired by theirs. Bordawekar *et al.* [2] present a technique for optimizing communication for out-of-core distributed stencil computations. They show how a compiler can choose the tiling parameters based on the stencil computation and processor information. Their goal is to minimize the communication, whereas our goal is to find the tiling strategy and tile sizes that minimize the total execution time.

7. Conclusions and Future Work

We have characterized the space of legal multi-level tilings and parallelizations for the 2D/3D 9-pt Gauss-Siedel stencil computations. We have shown that a systematic exploration of a part (2 strategies) of this space leads to a new strategy which achieves up to a factor of two improvement over the standard implementation. A two fold decrease in running time is significant for such applications. This illustrates the importance of exploring this space. Further, the exploration helped us to divide the input space into regions where different designs are better. This shows us the need for runtime data dependent choice of the best implementation. We consider our results as a first step towards a

complete exploration of this space.

As a future work, we envision to build a framework that will take a stencil computation as input and will automatically determine the required skewing transformation, and generate analytical models for different tiling and parallelization strategies, and select the best strategy. Majority of the required theory for this is known, and we believe that our GP framework is general enough to integrate all these techniques into a single tool. As an immediate future work, we wish to implement and explore other tiling strategies.

Acknowledgments. We would like to thank the Colorado State University IsTeC organization, COGRID, and the National Center for Atmospheric Research (NCAR) for allowing us to use their computational resources for experimental validation. This research was supported in part, by the National Science Foundation, under the grant EI-030614: HiPHiPECS: High Level Programming of High Performance Embedded Computing Systems.

References

[1] R. Andonov, S. Balev, S. V. Rajopadhye, and N. Yanev. Optimal semi-oblique tiling. *IEEE Trans. Parallel Distrib. Syst.*, 14(9):944–960, 2003.

[2] R. Bordawekar, A. Choudhary, and J. Ramanujam. Automatic optimization of communication in compiling out-of-core stencil codes. In *ICS '96: Proceedings of the 10th international conference on Supercomputing*, 1996.

[3] S. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.

[4] M. Bromley, S. Heller, T. McNerney, and J. Guy L. Steele. Fortran at ten Gigaflops: the connection machine convolution compiler. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, 1991.

[5] A. Darte. Regular partitioning for synthesizing fixed-size systolic arrays. *Integration, The VLSI J.*, 12(3):293–304, 1991.

[6] M. A. Frumkin and R. F. V. der Wijngaart. Tight bounds on cache use for stencil operations on rectangular grids. *J. ACM*, 49(3):434–453, 2002.

[7] W. D. Gropp. Solving pdes on loosely-coupled parallel processors. *Parallel Computing*, 5(1-2):165–173, 1987.

[8] K. Högstedt, L. Carter, and J. Ferrante. Determining the idle time of a tiling. In *POPL*, 1997.

[9] Y. C. Hu, G. Jin, S. L. Johnsson, D. Kehagias, and N. Shalaby. Hpfbench: a high performance fortran benchmark suite. *ACM Trans. Math. Softw.*, 26(1):99–149, 2000.

[10] F. Irigoin and R. Triolet. Supernode partitioning. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1988.

[11] S. Kamil, P. Husbands, L. Oliker, J. Shalf, and K. Yelick. Impact of modern memory subsystems on cache optimizations for stencil computations. In *MSP '05: Proceedings of the 2005 workshop on Memory system performance*, 2005.

[12] R. M. Karp, R. E. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *J. ACM*, 14(3):563–590, 1967.

[13] K. O. Kortanek, X. Xu, and Y. Ye. An infeasible interior-point algorithm for solving primal and dual geometric programs. *Math. Program.*, 76(1):155–181, 1997.

[14] Z. Li and Y. Song. Automatic tiling of iterative stencil loops. *ACM Trans. Program. Lang. Syst.*, 26(6):975–1028, 2004.

[15] J. Löfberg. YALMIP : A toolbox for modeling and optimization in MATLAB. In *Proceedings of the CACSD Conference*, 2004.

[16] N. Mitchell, K. Högstedt, L. Carter, and J. Ferrante. Quantifying the multi-level nature of tiling interactions. *International J. of Parallel Programming*, 26(6):641–670, 1998.

[17] D. I. Moldovan and J. A. B. Fortes. Partitioning and mapping algorithms into fixed size systolic arrays. *IEEE Trans. Comput.*, 35(1):1–12, 1986.

[18] NAS Parallel Benchmarks. Available from `http://www.netlib.org/parkbench/`.

[19] PARKBENCH: PARallel Kernels and BENCHmarks. Available from `http://www.netlib.org/parkbench/`.

[20] P. Quinton and V. Van Dongen. The mapping of linear recurrence equations on regular arrays. *Journal of VLSI Signal Processing*, 1(2):95–113, 1989.

[21] S. V. Rajopadhye and R. M. Fujimoto. Synthesizing systolic arrays from recurrence equations. *Parallel Computing*, 14:163–189, June 1990.

[22] L. Renganarayana and S. Rajopadhye. A geometric programming framework for optimal multi-level tiling. In *SC '04: Proceedings of the ACM/IEEE conference on Supercomputing*, 2004.

[23] G. Rivera and C.-W. Tseng. Tiling optimizations for 3d scientific computations. In *Supercomputing '00: Proceedings of the ACM/IEEE conference on Supercomputing*, 2000.

[24] G. Roth, J. Mellor-Crummey, K. Kennedy, and R. G. Brickner. Compiling stencils in high performance fortran. In *Supercomputing '97: Proceedings of the ACM/IEEE conference on Supercomputing*, 1997.

[25] SPEC CPU2000 benchmark. `http://www.spec.org`.

[26] R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. In *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, 1998.

[27] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, 1991.

[28] M. Wolfe. More iteration space tiling. In *Supercomputing '89: Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, 1989.

[29] D. Wonnacott. Using time skewing to eliminate idle time due to memory bandwidth and network limitations. In *IPDPS '00: Proceedings of the 14th International Symposium on Parallel and Distributed Processing*, 2000.

[30] D. Wonnacott. Achieving scalable locality with time skewing. *Int. J. Parallel Program.*, 30(3):181–221, 2002.

[31] J. Xue. On tiling as a loop transformation. *Parallel Processing Letters*, 7(4):409–424, 1997.

[32] J. Xue. *Loop tiling for parallelism*. Kluwer Academic Publishers, 2000.