Diamond Tiling: Tiling Techniques to Maximize Parallelism for Stencil Computations

Uday Bondhugula, Member, IEEE, Vinayaka Bandishti, and Irshad Pananilath

Abstract—Most stencil computations allow tile-wise concurrent start, i.e., there always exists a face of the iteration space and a set of tiling directions such that all tiles along that face can be started concurrently. This provides load balance and maximizes parallelism. However, existing automatic tiling frameworks often choose hyperplanes that lead to pipelined start-up and load imbalance. We address this issue with a new tiling technique, called diamond tiling, that ensures concurrent start-up as well as perfect load-balance whenever possible. We first provide necessary and sufficient conditions for a set of tiling hyperplanes to allow concurrent start for programs with affine data accesses. We then provide an approach to automatically find such hyperplanes. Experimental evaluation on a 12-core Intel Westmere shows that diamond tiled code is able to outperform a tuned domain-specific stencil code generator by 10% to 40%, and previous compiler techniques by a factor of 1.3× to 10.1×.

Index Terms—Compilers, program transformation, loop tiling, parallelism, locality, stencils

1 INTRODUCTION AND MOTIVATION

Stencil computation is an important class of computations appearing in many scientific and engineering applications. These computations perform updates to a grid of points where one or more values at each grid point is updated using values from neighboring grid points. When defined on structured grids, stencil computations often have a regular data access pattern that allows automatic compile-time analysis and lends itself to good locality optimization and parallelization. Figure 1 shows a stencil computation over a one-dimensional data space used to model a one-dimensional heat equation.

Loop tiling [1], [2], [3] is a key transformation used to exploit data locality and parallelism from stencil computations. Tiling has often been specified by tile shape and tile size. Tile shape is obtained from the directions chosen to slice iteration spaces of statements – these directions are represented by tiling hyperplanes [1], [4], [5]. Finding the right shape and size is the subject of numerous works with goals of improving locality, controlling the frequency of synchronization, and volume of communication where applicable. Performing parallelization and locality optimization together on stencils can often lead to pipelined startup [6], i.e., not all processors are busy during parallelized execution. This is the case with a number of general compiler techniques from the literature [7], [8], [9]. With an increasing number of cores per chip, it is beneficial to maintain load balance by enabling concurrent start of tiles along an iteration space boundary whenever possible. Concurrent start for stencil computations not only eliminates pipeline fill and drain delay, but also ensures load balance – processors execute the same maximal amount of work in parallel between two synchronization points.

Some works studied eliminating pipelined startup [10], [11] by modifying tile shapes obtained from existing frameworks. However, these approaches have undesired side-effects including difficulty in performing code generation. No implementations of these have been reported yet. The approach we propose in this paper works by determining tiling hyperplanes that have the desired property of concurrent start, as opposed to one of correcting or tweaking hyperplanes found with undesired properties. It has not been clear if and under what conditions such hyperplanes existed, and how they could be found. Our paper provides an answer to these questions.

The techniques we propose result in tiling that allows concurrent start of tiles, and we term them diamond tiling. At least three past works, that of Eissfeller and Muller [12], Orozco and Gao [13] and Strzodka et al. [14], have evaluated diamond-shaped tiles – as a specialized tiling technique without a formalism, and with only one dimension of concurrent start. Our approach proposes a generalizing formalism for diamond tiling using the polyhedral framework; this also makes it natural for automatic code generation.

In summary, our contributions are as follows:

- provide conditions under which tiling hyperplanes allow concurrent start,
- provide an approach to find such tiling hyperplanes, and
- an experimental evaluation and comparison of our technique with the state-of-the-art from compiler approaches as well as domain-specific ones.

This work is a significantly revised version of our previous work that was published as [15]. This revision primarily introduces a detailed analysis of cache performance, a comparison between full and lower dimensional diamond tiling, a different approach to find diamond tiling hyperplanes that also works for multiple statements, a more detailed
2 Background and Notation

The · operator is used for a vector dot product or a matrix vector product. We use \( \langle \vec{i}, \vec{j} \rangle \) to represent the vector obtained by concatenating the components of \( \vec{i} \) and \( \vec{j} \). For convenience, vectors may not always be expressed in column form, but also as a comma separated tuple of its individual dimension components. Vectors are denoted using lowercase letters with overhead left arrows while matrices are represented with uppercase letters. The number of rows in a matrix \( M \) is denoted by \( nr(M) \). Individual rows of a matrix are referred to using a subscript: \( M[i] \) refers to row \( i \) of matrix \( M \) as a row vector, \( 1 \leq i \leq nr(M) \). \( u_i \) represents a unit vector along the canonical direction \( i \) (\( i \geq 1 \)). \( I \) denotes the identity matrix, i.e., \( I[k] = \vec{u}_k \).

Conic and strict conic combination: A conic combination of vectors \( \vec{x}_1, \vec{x}_2, \ldots, \vec{x}_n \) is a vector of the form

\[
\lambda_1 \vec{x}_1 + \lambda_2 \vec{x}_2 + \cdots + \lambda_n \vec{x}_n, \quad \lambda_1, \lambda_2, \ldots, \lambda_n \geq 0.
\]

If all \( \lambda_i > 0 \), we call (1) a strict conic combination of \( \vec{x}_1, \vec{x}_2, \ldots, \vec{x}_n \).

2.1 Characterizing Iterative Stencils on Structured Grids

We assume and exploit the following inherent properties of stencil computations. The data structure being utilized is a discretized representation of the data grid with each location being updated using values of neighboring grid locations from previous recent time steps. Let the dimensionality of the data grid be \( d \). The input loop nest will be \( d + 1 \)-dimensional with the outermost loop iterating over time. The inner \( d \) loops iterate over dimensions of the data grid, and we refer to these as space loops. The lower and upper bounds of the loops typically make the shape of the resulting iteration space hyperrectangular.

2.2 Polyhedral Representation

Let \( S \) be a statement in the program with \( m_S \) loops surrounding it. The set of iterations or statement instances to be executed for a statement \( S \) is the domain or index set of \( S \), and is denoted by \( I_S \). An iteration of statement \( S \) is denoted by \( i_S \), i.e., \( i_S \in I_S \). \( i_S \) has \( m_S \) components corresponding to loops surrounding \( S \) from outermost to innermost. Geometrically viewed, \( I_S \) is the set of integer points in a polyhedron of dimensionality \( m_S \). Algebraically, it can be represented as a polyhedral set. As an example, an index set corresponding to the two-dimensional loop nest shown in Figure 1 with loops \( t \) and \( i \) is compactly represented as:

\[
I_S = \{(t, i) : 0 \leq t \leq T - 1, 1 \leq i \leq N - 2 \}.
\]

Any symbols in the RHS not appearing within parentheses should be treated as program parameters. \( N \) is a program parameter above. Program parameters are symbols that are not modified in the part of the program being analyzed. These symbols typically represent problem sizes and usually appear in upper bounds of loops. Let \( \vec{p} \) be the vector of those program parameters.

\[
\text{for } (t = 0; t <= T-1; t++) \\
\text{for } (i = 1; i <= N-2; i++) \\
A[(t+1)%2][i] = 0.250 * (A[i%2][i+1] + 2.0 * A[i%2][i] + A[i%2][i-1]); \\
\]

Fig. 1. Stencil: Jacobi iterations for 1-d heat equation

2.2.1 Dependences

The data dependence graph, \( G = (S, E) \) is a directed multi-graph with each vertex representing a statement in the program and edge \( e \) from \( S_i \) to \( S_j \) representing a polyhedral dependence from a dynamic instance of \( S_i \) to one of \( S_j \). Every edge \( e \) is characterized by a polyhedron, \( D_e \), called the dependence polyhedron that captures conditions under which dependences exist between dynamic instances of \( S_i \) and \( S_j \). An iteration \( t \in I_{S_i} \) depends on \( s \in I_{S_j} \) through edge \( e \) whenever \( <s, t> \in D_e \). \( S_i \) is referred to as the source statement of the dependence, and \( S_j \) the target statement. As an example, the dependence polyhedron for the flow dependence between the write access and the last read access in Figure 1 is given by:

\[
D_e = \{<s, t> : s = (t, i), t = (t', i'), i' = i + 1, \]
\[
\text{for } (t = 0; t <= T-1; t++) \\
\text{for } (i = 1; i <= N-2; i++) \\
A[(t+1)%2][i] = 0.250 * (A[i%2][i+1] + 2.0 * A[i%2][i] + A[i%2][i-1]); \\
\]

One can obtain a less powerful representation such as a constant distance vector or a direction vector from dependence polyhedra by analyzing the relation between source and target iterators. Throughout the paper, we make use of constant distance vectors in the examples for explanation as stencils typically only have constant dependences.

2.2.2 Hyperplanes

A hyperplane is an \( n - 1 \) dimensional affine subspace of an \( n \) dimensional space. For example, any line in a 2-d space and any 2-d plane in 3-d space are examples of hyperplanes. A one-dimensional affine transformation for a statement \( S \), \( \phi_S \), specifies a one-dimensional partitioning of its index set \( I_S \):

\[
\phi_S(i_S) = \left( c^S_0, c^S_1, \ldots, c^S_{m_S} \right) \cdot i_S + c^S_0, \\
\text{where } c^S_0, c^S_1, \ldots, c^S_{m_S} \in \mathbb{Z}, \\
\text{and } \vec{h}_S \cdot \vec{x} + c^S_0 \text{ where } \vec{h}_S = (c^S_1, \ldots, c^S_{m_S}).
\]
The vector $\vec{h}_S$ represents a vector normal to each of the $m_S - 1$ dimensional partitions, and refer to it as the hyperplane associated with $\phi_S$. In particular, $\vec{h}_S$ captures the orientation of the partitioning, and $e_{ik}^S$ is the translation or the constant shift component. We will shortly see that hyperplanes are used to define tile shapes.

A statement is transformed using a multi-dimensional affine transformation: each dimension of it corresponds to a hyperplane $\vec{h}$. If there are $m_S$ linearly independent hyperplanes for statement $S$, they constitute a one-to-one mapping for $S$. In the case of constant dependences and a single statement, a hyperplane $\vec{h}$ for a statement is said to carry or satisfy a dependence $\vec{d}$ if $\vec{h} \cdot \vec{d} \geq 1$. Note that this notion of dependence carrying is different from that of loops carrying dependences in that it is regardless of an ordering to allow tile-wise concurrent start. Finding tile-wise concurrent start becomes important and non-trivial when dependences span the entire iteration space.

### 2.2.3 Validity of Tiling

Prior research [9] provides conditions for hyperplanes to be valid tiling hyperplanes in the polyhedral framework. In the case of a stencil computation, we consider here the ability to tile all dimensions, i.e., the space dimensions as well as the time dimension. For $\phi_{s_1}, \phi_{s_2}, \ldots, \phi_{s_k}$ to be valid statement-wise tiling hyperplanes for $S_1, S_2, \ldots, S_k$, respectively, the following should hold for all $\langle \vec{s}, \vec{t} \rangle \in D_e$ where $\vec{s} \in I_{S_i}$, $\vec{t} \in I_{S_i}$ and for all $e \in E$:

$$\phi_{s_i}(\vec{t}) - \phi_{s_i}(\vec{s}) \geq 0.$$  

The above constraint implies that all dependences have non-negative components along each of the hyperplanes, i.e., their projections on these hyperplane normals are never in a direction opposite to that of the hyperplane normals. In the case of a single statement with constant dependences, the above condition can be stated in a simpler way [1]. A hyperplane $\vec{h}$ is a valid tiling hyperplane if for all dependences $\vec{d}_i$:

$$\vec{h} \cdot \vec{d}_i \geq 0.$$  

In addition, the set of tiling hyperplanes should be linearly independent of each other. Each statement has as many linearly independent tiling hyperplanes as its index set dimensionality, i.e., $m_S$ for statement $S$.

### 2.3 Concurrent Start in Iteration Spaces

The notion of concurrent start was first defined by Krishnamoorthy et al. [11]. If all iterations along a face can be started concurrently, the face is said to allow point-wise concurrent start. Similarly, in a tiled iteration space, if all tiles along a face can be started concurrently, the face is said to allow tile-wise concurrent start. Throughout the paper, a face of an iteration space is referred by its normal $\vec{f}$.

A face of an iteration space allows point-wise concurrent start if there are no dependences parallel to the face. As stated in [11], this condition for a single statement and constant dependences is as follows. A face $\vec{f}$ allows concurrent start if for all dependences $\vec{d}_i$:

$$\vec{f} \cdot \vec{d}_i \geq 1.$$  

An update at a grid location in a stencil computation typically uses neighboring values along all data dimensions and along either direction for each data dimension. This leads to a situation where there is no communication-free or synchronization-free parallelism [16]. Formally, communication-free parallelism exists if there exists a hyperplane $\vec{h}$ such that $\vec{h} \cdot \vec{d}_i = 0 \forall \vec{d}_i$. In the presence of a communication-free parallel loop, the best parallelization is easily achieved by parallelizing it without considering concurrent start. Finding tile-wise concurrent start becomes important and non-trivial when dependences span the entire iteration space.

### Multiple Statements and Affine Dependences

We observe that (5) is equivalent to the condition for the existence of a one-dimensional schedule along the normal to the face. We now state the concurrent start condition for multiple statements and affine dependences. For affine dependences, there exists point-wise concurrent start along a face $\vec{f}_{S_k}$ of statement $S_k$ if for all edges $e$ from $S_k$ to any $S_i$:

$$\vec{f}_{S_k} \cdot \vec{t} - \vec{f}_{S_i} \cdot \vec{s} \geq 1, \forall \langle \vec{s}, \vec{t} \rangle \in D_e, \vec{s} \in I_{S_k}, \vec{t} \in I_{S_i}.$$  

### 2.4 Inter-tile Dependences

Once a transformation involving a skewing is applied to enable tiling, the transformed space can be tiled rectangularly. Since dependences have non-negative components along all tiling hyperplanes, the inter-tile dependences in the transformed space are vectors with non-negative values. As dependences in stencil computations are short and span the entire iteration space (there is no communication-free parallelism), the canonical unit vectors in the transformed space are the inter-tile dependences. Strictly speaking, these canonical unit vectors are actually the extremal vectors of the cone of inter-tile dependences, i.e., any conic combination of these would be redundant for consideration as an inter-tile dependence. In the rest of this paper, we refer to the extremal vectors of the cone of inter-tile dependences as simply the inter-tile dependences. All inter-tile dependences will be satisfied by a scan of the tile space dimensions in a lexicographically increasing order (Figure 2).

We now show how the inter-tile dependences in the original iteration space can be determined. Let $G$ be the matrix with the inter-tile dependences in the original iteration space as its columns. As stated earlier, the transformed inter-tile dependences for stencil computations will be canonical unit vectors. If $T_R$ is a transformation with the tiling hyperplanes as its rows, $T_R \cdot G = I$, i.e., $G = T_R^{-1}$.

Consider the code in Figure 3, which has dependences $(1,0), (1,-2)$ and $(1,2)$.

![Fig. 2. Transformed iteration space after applying (1,0) and (2,1) as the tiling hyperplanes. Inter-tile dependences are just unit vectors of the canonical basis (t1, t2).](image-url)
for \( t=0, t\in T; t=s \)
for \( i=2; i<N; j+1 \)
\[ A[t+1][i][j] = (A[t+1][i][j-2] + A[t+1][i][j+2]) / 3.0; \]

Fig. 3. Example with dependences \((1,0), (1,2), (1,2)\)

The inter-tile dependences introduced by hyperplanes \((1,0)\) and \((2,1)\) can be determined as follows:

\[
G = \begin{pmatrix} 1 & 0 \\ 2 & 1 \end{pmatrix}^{-1} = \begin{pmatrix} 1 & 0 \\ -2 & 1 \end{pmatrix}.
\]

Thus, the inter-tile dependences are \((1,-2)\) and \((0,1)\) (shown in Figure 4). Also, given that we are only considering extremal inter-tile dependences (dependences that are conic combinations of these are not considered), each of the inter-tile dependences is along only one of the tiling hyperplanes and is normal to all other tiling hyperplanes.

Fig. 4. Inter-tile dependences for the example in Figure 3 introduced by hyperplanes \((1,0)\) and \((2,1)\)

**Tile schedule**: A tile schedule provides an ordering for the tiles that are to be executed sequentially. In our context, it is another affine schedule on the space of tiles. A tile schedule is valid if it carries all inter-tile dependences.

### 2.5 Limitation of Existing Cost Functions for Tiling

Existing cost functions for choosing tiling hyperplanes do not model tile-wise concurrent start. Consider again the program in Figure 3. As all dependences are self dependences with constant distances, they can be represented by distance vectors; the dependences are \(d_1 = (1,-2)\), \(d_2 = (1,0)\), and \(d_3 = (1,2)\). Inequality (4), which provides the condition for a tiling hyperplane to be valid, can now be written as

\[
\bar{h} \cdot \begin{pmatrix} 1 & 1 & 0 \\ -2 & 0 & 2 \end{pmatrix} \geq \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}. \quad (7)
\]

The coefficients of \(\bar{u}\) and \(w\) are then lexicographically minimized to obtain a solution. Repeatedly finding linearly independent hyperplanes this way has the effect of reducing reuse distances and bringing parallelism to the outer loops. In the case of a single statement and constant dependence vectors, the dependence distance of \(d_i\) along \(h_S\) is simply given by \(h_S \cdot d_i\); hence, minimizing \(\bar{u}, w\) is the same as

\[
\text{minimize } \max_{d_i} h_S \cdot d_i. \quad (9)
\]

Returning to our example (Figure 3), the cost function (9) chooses \((1,0)\) and \((2,1)\) as the linearly independent hyperplanes. Note that (9) evaluates to 1 for \(h = (1,0)\) and this is the minimum for any \(h\), and (9) evaluates to 2 for \(h = (2,1)\). With these tiling hyperplanes, there does not exist a face along which the tiles can start in parallel, i.e., there would be a pipelined startup and drain phase (Figure 6). Decreasing the tile size would reduce the startup and drain phase but would increase the frequency of synchronization. Increasing the tile size would mean a shorter steady-state. In summary, concurrent start is lost with these tiling hyperplanes.

Consider Figure 4 to analyze the loss of concurrent start by reasoning through inter-tile dependences. The tiling hyperplanes in the figure are \((1,0)\) and \((2,1)\). The inter-tile dependence satisfied by \((1,0)\) is \((1,-2)\), and that satisfied by \((2,1)\) is \((0,1)\). Hence, concurrent start along \((1,0)\) is prevented by the tile dependence \((1,1)\).

It is desirable to find a tiling that does not introduce an inter-tile dependence prohibiting concurrent start. If we had chosen \((2,-1)\), which is also valid, instead of \((1,0)\), i.e., \((2,-1)\) and \((2,1)\) had been chosen as the tiling hyperplanes, then the inter-tile dependences introduced would have been \((1,-2)\) and \((1,2)\) (Figure 7). Both have positive components along the face normal \((1,0)\), i.e., all tiles along \((1,0)\) can be started concurrently. Properties of tiling hyperplanes such as this are the subject of the next section.

### 3 TILING FOR CONCURRENT START

In this section, we provide conditions for which tiling hyperplanes of any statement allow concurrent start along a given face of its iteration space. Theorem 1 introduces the constraints in terms of inter-tile dependences. Theorem 2 maps Theorem 1 from inter-tile dependences onto tiling hyperplanes. We also prove that these constraints are both necessary and sufficient for concurrent start.
3.1 Conditions for Tile-wise Concurrent Start

**Theorem 1.** A tile schedule enables tile-wise concurrent start along a face \( \vec{f} \) iff the tile schedule is in the same direction as \( \vec{f} \).

**Proof:** Given the conditions for point-wise concurrent start in (5) and (6), we can immediately extend the same notion to tile-wise concurrent start. If the columns of \( G \) contain the inter-tile dependences, and if there is tile-wise concurrent start along \( \vec{f} \), then \( \forall i \ f \cdot G^T [i] \geq 1 \). Thus, \( \vec{f} \) satisfies all inter-tile dependences and is a valid tile schedule. This proves the sufficient condition.

Let us say the tile schedule, say \( \vec{t} \), is in the same direction as \( \vec{f} \). Since a valid tile schedule carries all inter-tile dependences:

\[
  k_1 \vec{t} = k_2 \vec{f}, \quad k_1, k_2 \in \mathbb{Z}^+,
\]

\[
  \forall i \vec{t} \cdot G^T [i] \geq 1
\]

\[
  \Rightarrow \forall i \vec{f} \cdot G^T [i] = \frac{k_1}{k_2} \vec{t} \cdot G^T [i] \geq \frac{k_1}{k_2}.
\]

Since \( \vec{f} \cdot G^T [i] \in \mathbb{Z}^{m_s} \) and \( k_1/k_2 > 0 \), \( \vec{f} \cdot G^T [i] \geq 1 \) and the iteration space is hyper-rectangular, it follows from the condition of point-wise concurrent start (5) that \( \vec{f} \) enables tile-wise concurrent start. \( \square \)

As an illustration, in Figure 8, the face allowing concurrent start and outer tile schedule are the same, and carry both inter-tile dependences. Therefore, the tiles \( t_1, t_2, t_3, t_4 \) can be started concurrently.

We now state the most important result: under what conditions does a set of valid tiling hyperplanes enable concurrent start?

**Theorem 2.** For an iteration space that has no communication-free parallelism, tile-wise concurrent start along a face \( \vec{f} \) is enabled by a set of valid tiling hyperplanes, \( \vec{h}_1, \vec{h}_2, \ldots, \vec{h}_n \) iff \( \vec{f} \) lies strictly inside the cone formed by those hyperplanes, i.e., iff \( \vec{f} \) is a strict conic combination of all those hyperplanes:

\[
  k \vec{f} = \lambda_1 \vec{h}_1 + \lambda_2 \vec{h}_2 + \cdots + \lambda_n \vec{h}_n, \quad \lambda_1, \lambda_2, \ldots, \lambda_n, k \in \mathbb{Z}^+.
\]

**Proof (sufficient):** Since \( \vec{f} \) strictly lies in the cone of \( \vec{h}_1, \vec{h}_2, \ldots, \vec{h}_n \), there exist \( k, \lambda_1, \lambda_2, \ldots, \lambda_n \in \mathbb{Z}^+ \) such that

\[
  k \vec{f} = \lambda_1 \vec{h}_1 + \lambda_2 \vec{h}_2 + \cdots + \lambda_n \vec{h}_n.
\]

Since the columns of \( G \) represent inter-tile dependences, let \( \vec{g} = G^T [i] \) be some inter-tile dependence. Then,

\[
  k \vec{f} \cdot \vec{g} = \lambda_1 (\vec{h}_1 \cdot \vec{g}) + \lambda_2 (\vec{h}_2 \cdot \vec{g}) + \cdots + \lambda_n (\vec{h}_n \cdot \vec{g}).
\]

Since \( \vec{h}_1, \vec{h}_2, \ldots, \vec{h}_n \) are all linearly independent, there exists at least one \( \vec{h}_k \) such that \( \vec{h}_k \cdot \vec{g} \neq 0 \). Hence, the RHS of (11) will be strictly positive. Hence, \( \vec{f} \cdot \vec{g} > 0 \). Since \( \vec{f} \cdot \vec{g} > 1 \), and this holds for \( \vec{g} = G^T [i] \). Consequently, from Theorem 1, tile-wise concurrent start is enabled by choosing \( \vec{f} \) as the tile schedule. \( \square \)

**Proof (necessary):** Let us assume that we have concurrent start along the face \( \vec{f} \), but \( \vec{f} \) does not strictly lie inside the cone formed by the hyperplanes, i.e., (10) does not hold true. Assume to the contrary now that there do not exist \( \vec{h}_1, \vec{h}_2, \ldots, \vec{h}_n \) which are all strictly positive integers such that (10) holds true. Note that since \( \vec{h}_1, \ldots, \vec{h}_n \) are all linearly independent, one can express \( \vec{f} \) in the form of (10). Without loss of generality, assume \( k \in \mathbb{Z}^+ \). Let \( \lambda_k \leq 0 \). Now, since there is no communication-free parallelism, there exists an inter-tile dependence \( \vec{g} \) such that \( \vec{h}_k \cdot \vec{g} \geq 1 \) and \( \vec{h}_i \cdot \vec{g} = 0 \) \( \forall i \neq k \), since every inter-tile dependence is carried by only one of the chosen hyperplanes and is orthogonal to the rest (Section 2.4). Therefore, \( \vec{f} \cdot \vec{g} \leq 0 \), which implies that concurrent start is inhibited along \( \vec{f} \). This is a contradiction. \( \square \)

**Iteration Space Shape:** Our conditions for tile-wise concurrent start are independent of the shape of the iteration space – they only state when tiles along a direction parallel to the face can start in parallel. Even in the presence of tile-wise concurrent start, the maximum number of tiles that can be run concurrently per tile wavefront can increase or decrease depending on the shape of the iteration space as we step through the tile schedule. However, if the iteration space is hyper-rectangular, nearly the same amount of tile-level parallelism will be available for each iteration of the...
tile schedule. For stencil computations in this paper’s scope, iteration spaces are always hyper-rectangular; hence, concurrent start would also imply that nearly the same number of tiles will be available to run in parallel in every tile wavefront from start to finish.

4 FINDING DIAMOND TILING HYPERPLANES

We now present a scheme that is an extension to the Pluto algorithm [9] so that hyperplanes that satisfy properties proposed in the previous section are found.

4.1 Details of the Approach

The previous section showed that additional constraints are needed to enforce tile-wise concurrent start, on top of the constraints imposed to ensure validity of tiling and to encode a cost function.

Consider a single statement stencil. A face of its iteration space that allows point-wise concurrent start can be found using prior techniques by encoding (6). Let $\vec{f}$ be the face along which we would like to start concurrently, $H$ be the set of tiling hyperplanes already found, and $n$ be the dimensionality of the iteration space (same as the number of hyperplanes to find). Then, an approach can involve the following modified or additional steps:

1) for the first $n-1$ hyperplanes, $\vec{h}$ is linearly independent of $\vec{f} \cup H$, as opposed to just $H$;
2) for the last hyperplane $\vec{h}_n$, $\vec{h}_n$ is strictly inside the cone formed by $\vec{f}$ and the negatives of the already found $n-1$ hyperplanes, i.e.,

$$\vec{h}_n = \lambda_1' \vec{f} + \lambda_1(-\vec{h}_1) + \lambda_2(-\vec{h}_2) + \cdots + \lambda_{n-1}(-\vec{h}_{n-1}), \lambda'_1, \lambda_1, \ldots, \lambda_n \in \mathbb{Z}^+.$$  \hspace{1cm} (12)

Cone complement hyperplane: We call $\vec{h}_n$, obtained by enforcing (12), the cone complement hyperplane. For finding the cone complement, the base Pluto ILP formulation is added additional variables corresponding to $\lambda'_1, \lambda_1, \ldots, \lambda_{n-1}$. The constraint (12) and the constraints on the $\lambda$s being greater than or equal to one are added. Since $\vec{h}_1, \ldots, \vec{h}_{n-1}$ are known, (12) is linear. The lexicographical minimal solution with $\lambda$s at the end is then obtained for this new ILP formulation.

The above approach is presented as Algorithm 1; only our additions to the original Pluto algorithm proposed in [9] and the surrounding context are shown. For convenience, $H$ has been treated as a set of hyperplanes instead of a matrix.

However, what Algorithm 1 does can be more easily accomplished by running the existing Pluto algorithm, discarding the concurrent start face hyperplane if it was among the hyperplanes found, and using the cone complement to replace it; if the face hyperplane was not among the ones found, we can choose the last hyperplane for replacement with the cone complement. We present this approach as Algorithm 2, and it is in a general form that works for multiple statements as well.

With the above proposed approach, we continue to use the existing cost function of Pluto, but (1) avoid the concurrent start face as being chosen as one of the hyperplanes, and (2) choose the last hyperplane so that the face lies inside the cone of hyperplanes. If it is not possible to find hyperplanes with the additional constraints, we report that tile-wise concurrent start is not possible. If there exists a set of hyperplanes that allows tile-wise concurrent start, we prove below that the above algorithm will find it.

Theorem 3. Algorithm 1 (and equivalently Algorithm 2) finds tiling hyperplanes that allow tile-wise concurrent start for any stencil computation that allows point-wise concurrent start.

Proof (soundness): When Algorithm 2 returns a set of hyperplanes, it is clear that all of them are valid tiling hyperplanes and are linearly independent of each other. In addition, they also satisfy (12), which was obtained by merely rearranging terms of (10). Thus, whenever the algorithm succeeds in finding a cone complement hyperplane, the output is correct and enables concurrent start.

Proof (completeness): We now prove that whenever our algorithm does not find concurrent-start tiling hyperplanes, such tiling hyperplanes do not exist. Assume that the first $n-1$ hyperplanes found were $\vec{h}_1, \vec{h}_2, \ldots, \vec{h}_{n-1}$. Clearly, these are linearly independent of the concurrent start face $\vec{f}$. Now we will show that the last hyperplane satisfying (10) exists and is found by Step 8 of Algorithm 1. Consider a hyperplane $\vec{h}_n$ such that

$$\vec{h}_n = \lambda'_1(\vec{f}) + \lambda_1(-\vec{h}_1) + \lambda_2(-\vec{h}_2) + \cdots + \lambda_{n-1}(-\vec{h}_{n-1}), \lambda'_1, \lambda_1, \ldots, \lambda_n \in \mathbb{Z}^+. \hspace{1cm} (13)$$

where all $\lambda$s have been chosen arbitrarily from $\mathbb{Z}^+$. Now, for a dependence $\vec{d}$, consider $\vec{h}_n \cdot \vec{d}$ given by:

$$\vec{h}_n \cdot \vec{d} = \lambda'_1(\vec{f} \cdot \vec{d}) + \lambda_1(-\vec{h}_1 \cdot \vec{d}) + \lambda_2(-\vec{h}_2 \cdot \vec{d}) + \cdots + \lambda_{n-1}(-\vec{h}_{n-1} \cdot \vec{d}), \lambda'_1, \lambda_i \in \mathbb{Z}^+. \hspace{1cm} (14)$$

Consider the RHS of (14). As $\vec{h}_1, \ldots, \vec{h}_{n-1}$ are valid tiling hyperplanes, $\vec{h}_1 \cdot \vec{d} \geq 0$, and since $\lambda_i$s are all positive, all terms of the RHS except for the first one are non-positive. For any stencil with point-wise concurrent start, $\vec{f}$ satisfies all dependences, i.e., $\forall \vec{d}, \vec{f} \cdot \vec{d} \geq 1$. If dependences are all

<table>
<thead>
<tr>
<th>Algorithm 1: Finding tiling hyperplanes that allow concurrent start for a single statement stencil</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Require:</strong> The polyhedral dependences for statement $S$</td>
</tr>
<tr>
<td>1: Initialize $H \leftarrow \phi$</td>
</tr>
<tr>
<td>2: Compute constraints for the validity of tiling, $V$ (using (4))</td>
</tr>
<tr>
<td>3: for $m_S - 1$ times do</td>
</tr>
<tr>
<td>4: Add constraints to $V$ such that the hyperplane to be found is linearly independent of $H \cup \vec{f}$</td>
</tr>
<tr>
<td>5: Add cost function constraints and minimize cost function for the optimal solution (using (8) and (9)).</td>
</tr>
<tr>
<td>6: $H \leftarrow H \cup \vec{h}$</td>
</tr>
<tr>
<td>7: end for</td>
</tr>
<tr>
<td>8: Add constraint (12) to $V$ so that the last hyperplane strictly lies inside the cone of the face and negatives of the $m_S - 1$ hyperplanes already found ($H$)</td>
</tr>
<tr>
<td>9: If a solution to Step 8 exists, concurrent start is possible. Otherwise, use the standard procedure to find the last hyperplane.</td>
</tr>
<tr>
<td><strong>Ensure:</strong> $H$: a set of $m_S$ valid linearly independent tiling hyperplanes for $S$</td>
</tr>
</tbody>
</table>
ALGORITHM 2: Finding diamond tiling hyperplanes that allow concurrent start for multiple statements

Require: The polyhedral representation of a sequence of arbitrarily nested affine loop nests that includes statement index sets and dependences
1: Run the existing Pluto algorithm, and obtain $H_S \forall S$, and the outermost non-trivial bands of permutable loops (with two loops or more)
2: Obtain the validity constraints for tiling: $V$
3: for each permutable band $B$ from Step 1 do
4:   Let $S_B$ be the statements under band $B$
5:   Let $G_S$ (where $S \in S_B$) comprise hyperplanes of $S$ corresponding to $B$ ($G_S$ is thus a contiguous set of hyperplanes from $H_S$)
6:   Find (point-wise) concurrent start faces: $f_S \forall S \in S_B$ (using (6))
7:   if ($f_S$ does not exist for some $S \in S_B$ or tiling will not lead to loss of concurrent start along $f_S$ for any $S \in S_B$) then
8:      continue
9:   end if
10:  if $f_S \in G_S$ for any $S \in S_B$ then
11:     /* The position/depth of $f_S$ in $H_S$ is chosen for eviction $\forall S \in S_B$ */
12:     $e_S \leftarrow f_S \forall S \in S_B$
13:   else
14:     /* Evict last hyperplane from the band */
15:     $e_S \leftarrow$ last hyperplane of $G_S \forall S \in S_B$
16:   end if
17: Add constraint (12) to $V$ for determining the cone complement hyperplane, and solve the ILP for the cone complements ($h_n \forall S \in S_B$)
18: if cone complement hyperplanes could not be found then
19:   Tile-wise concurrent start is not possible for $B$
20: else
21:   $G_S \leftarrow G_S \setminus e_S \cup h_n \forall S \in S_B$
22: end if
23: end for
Ensure: $\forall S \ H_S$: a set of $m_S$ valid linearly independent tiling hyperplanes for $S$

constant distance vectors, all $\lambda_i(-\vec{h} \cdot \vec{d})$ terms are independent of program parameters. Hence, we can always choose a sufficiently large $\lambda'$ such that $\vec{h}_n \cdot \vec{d} \geq 0$ for any $\vec{h}_1, \ldots, \vec{h}_{n-1}, \vec{d}$ under these conditions. Step 8 of Algorithm 1 will thus always yield a solution since a $\vec{h}_n$ satisfying (12) always exists.

General case beyond stencils: In the general case of affine dependences, our algorithm may fail to find the cone complement $\vec{h}_n$ in only two cases. The first when it is not statically possible to choose positive $\lambda_i$s. This can happen when, along the face allowing concurrent start, dependences have components that depend on program parameters. In this case, only point-wise concurrent start is possible, but tile-wise concurrent start is not possible. Figure 9 provides such an example. In order for our algorithm to succeed, one of the $\lambda$s would have to depend on a program parameter (typically the problem size), and this is not admissible. The second case is the expected one that does not allow even point-wise concurrent start along any face (condition (5)). This scenario occurs in Seidel-style stencils (eg. Figure 10) where neighboring values from the same time step are used.

Tile schedule: Once a solution for the cone complement’s ILP is obtained, we also obtain the values of $\lambda$s besides the coefficients of $\vec{h}_n$; the tile schedule is then set to $\hat{f}$. In the common case, we often obtain $\lambda_i = 1$ and $\lambda' = n$, i.e., the sum of the hyperplanes gives a vector that is a constant times the face.

4.2 Partial Concurrent Start and Lower-Dimensional Diamond Tiling

By making the outer tile schedule parallel to the face allowing concurrent start (as discussed in the previous section), one can obtain $n-1$ degrees of concurrent start, i.e., all tiles on the face can be started concurrently. We refer to this as full-dimensional or simply full diamond tiling. Full diamond tiling provides asymptotic scalability [17]. However, in practice, exploiting all of these degrees of parallelism may not be necessary. A full diamond tile has a more complex shape, and we show through experiments in the next section that it is not the best for single-thread performance. Due to the increasing and decreasing width of the diamond tile along the space dimensions (see Figure 7 for example), auto-vectorization and prefetching are expected to be not as efficient as for loops with a steady and large enough width along space dimensions. Note that parallelogram tiling, split tiling [11], [18], hexagonal tiling [19], and trapezoidal shapes do not have this “width” issue: their width along a space dimension remains steady (or large enough) with respect to time or other space dimensions.

To address the above issue with diamond tiling, diamond tiling can be restricted to a subset of the dimensions. The conditions that we derived for diamond tiling can be placed only on the first few hyperplanes to obtain what we term, partial concurrent start or lower-dimensional diamond tiling. For instance, the constraints can be placed only on the first two hyperplanes so that one degree of concurrent start is exploited. For the number of cores we typically have on a single shared-memory multicore system, this often provides sufficient parallelism for problem sizes of interest. In addition, it allows one to choose tile sizes along the remaining dimension(s) to obtain the desired width along the fastest varying space dimension. This in turn leads to better hardware prefetching performance and vectorization.
4.3 Intra-Tile Reordering

Once constraints for diamond-shaped tiles are set up, one is free to reorder execution within a diamond-shaped tile to further improve locality or vectorization. Diamond tiling hyperplanes found by our algorithm cannot be better than those found by the existing Pluto algorithm with respect to its dependence distance minimizing cost function (9). Therefore, for aspects other than concurrent start, in particular, for single-thread performance, a different schedule can be used inside a tile to aid better vectorization and hardware prefetching. We choose to use the hyperplanes found by the existing Pluto algorithm inside a tile—these are already available from Step 1 of Algorithm 2. By using diamond tiling hyperplanes to only shape tiles and using the existing Pluto algorithm’s scheduling hyperplanes to scan points inside a tile, we can obtain the desired benefits both in the tile space and inside a tile. Algorithm 2 readily allows this since the evicted hyperplane can be put back in place. Alternatively, a framework that explicitly encodes vectorization and spatial locality constraints like that of Kong et al. [20] can be used to find good intra-tile scheduling hyperplanes.

4.4 Example

We now show the complete sequence of transformations found using our approach for the stencil computation shown in Figure 11; it has two space dimensions and one time.

for (t = 0; t < T; t++)
for (i = 1; i < N+1; i++)
for (j = 1; j < N+1; j++)
A[(t+1)%2][i][j] = 0.125 \times (A[t%2][i+1][j] - 2.0 \times A[t%2][i][j+1]
- 2.0 \times A[t%2][i][j] + A[t%2][i][j-1]) + A[t%2][i][j];

Fig. 11. 2d-heat stencil (representative version)

The transformation computed by Algorithm 2 to allow full concurrent start is:

T(t, i, j) = (t + i, t + j, t - i - j), \quad (15)

with (t - i - j) being the cone complement hyperplane.

Figure 12 shows the shape of a tile formed using the above transformation.

For partial concurrent start, the cone complement is just computed using the first hyperplane, and Algorithm 2 will obtain

T(t, i, j) = (t - i, t + i, t + j). \quad (16)

The above enables concurrent start in only one dimension, i.e., not all tiles along the 2-d face will be able to start concurrently, but all tiles along one edge of the face can start concurrently.

4.4.1 Code Generation

The generation of code with diamond tiling transformations is done the same way as with any classic tiling technique [4]. As an example, using a tile size of 64x64x64 for the partial concurrent start transformation in (16) yields the transformation:

T^*((t, i, j)) = \left( \frac{t - i}{64}, \frac{t + i}{64}, \frac{t + j}{64}, t - i, t + i, t + j \right).

After creating the tile schedule and performing intra-tile reordering for the above as per Section 4.3, we obtain:

T^*((t, i, j)) = \left( \frac{t - i}{64}, \frac{t + i}{64}, \frac{t + j}{64}, t, t + i, t + j \right).

All divisions in the above formulae are integer divisions. Notice that while tiles are shaped as intended, the three inner loops that scan a tile have been changed to t, t + i, t + j, which are the hyperplanes found by the standard Pluto algorithm at Step 1 of Algorithm 2; the cone complement t - i no longer appears in it. The transformation above can be directly provided to a code generator like Cloog, and the second loop can be marked parallel using OpenMP, and the t + j loop can be vectorized.

4.5 The Case of Multiple Statements

The ability to handle stencil computations defined on stencil grids is important since several stencil codes involve them. Swim from SPECfp2000 is one example. In addition, in the presence of periodic boundary conditions, an index set splitting technique [21] leads to multiple statements, with a different transformation subsequently applied for each split statement to enable time tiling. In such cases, concurrent start and tiling has to be modeled in a coordinated manner for all statements.

To extend the formal conditions for the multiple statement case, \( \vec{h} \) has to be replaced with \( \vec{h}_{\vec{s}} \), and \( \vec{h} \cdot \vec{d} \) is to be replaced with \( \vec{h}_{\vec{s}}(\vec{t}) - \vec{h}_{\vec{s}}(\vec{s}) \), where the dependence is between \( \vec{s} \in I_{\vec{s}} \) and \( \vec{t} \in I_{\vec{s}} \), whenever \( (\vec{s}, \vec{t}) \in \mathcal{P}_e \). Hence, the dependences can be intra-statement or inter-statement. Theorem 1, Theorem 2, and their proofs presented earlier in this section thus also hold.

Algorithm 2 is already designed to work for the full general representation of the polyhedral framework—constant distance vectors need not be computed and provided as input at any point. A band is a sequence of contiguous nested loops that is permutable and can be tiled. Every outermost permutable band is tiled and parallelized by creation of a tile schedule if its outermost loop is not parallel. Condition (10) is enforced on all statements that are in the
same outermost band of permutable loops, and whenever point-wise concurrent start will be lost due to tiling. To determine the cone complement, condition (12) has to be applied for all statements simultaneously and included in the same ILP formulation. The cone complements for all statements are thus determined through a single ILP solution.

4.6 Effect of Tile Sizes

The discussion so far has assumed that tile sizes could be set later without a loss in concurrent start. However, Grosser et al. [22] show that concurrent start is lost with certain tile sizes. This is because, for certain tile sizes or tile size ratios, the tile schedule will no longer be parallel to the concurrent start face, despite the concurrent start face vector being parallel to the tile schedule vector in the way defined here. It is shown that there is only one ratio of tile sizes that will preserve concurrent start. We note that in all cases of 1-dimensional concurrent start (i.e., diamond tiling performed using only the first two dimensions), choosing equal tile sizes along those two dimensions will preserve concurrent start.

We found it convenient here to model concurrent start in a decoupled way – first to obtain the right tiling orientations as hyperplanes, and then choose tile sizes and the tile schedule so that concurrent start is preserved. Hence, strictly speaking, the conditions we have developed are not sufficient by themselves, but tile sizes have to be later chosen in a way that concurrent start is preserved [22].

5 Experimental Evaluation

We implemented our new approach in the source-to-source polyhedral tool chain, Pluto [23] (git version of ‘pet’ branch with commit version 0.11.3-131-g1fb0c27). In the rest of this section, pluto-diamond refers to diamond tiling with tile-wise concurrent start along only one dimension, i.e., it corresponds to partial concurrent start described in Section 4.2. It was generated using Pluto flags “-partltile -parallel”. Full diamond tiling is triggered with “-ltile -parallel”, and is referred to by pluto-diamond-full — it represents diamond tiling with concurrent start enabled along the entire concurrent start face (the transformation in (15) is an example). We compare performance of pluto-diamond with the standard Pluto algorithm that performs the usual time skewing for stencils (parallelogram tiles with pipelined startup) – we refer to the latter as pluto-standard and it serves as the state-of-the-art from the compiler works. Note that for pluto-standard, the tile wavefront was created from the first two hyperplanes to extract one degree of parallelism, which is the default behavior with Pluto. For all of the Pluto variants, Pet [24] was used as the frontend, Cloog [25] (version 0.18.2 with ISL backend) for code generation, and PIP [26] to solve for coefficients of hyperplanes. We also compare with the Pochoir stencil compiler [27] (version 0.5) which is representative of the state-of-the-art among domain-specific works.

All Pluto optimized codes used one level of tiling. Tile sizes used are shown in Table 2. In each case, we optimized tile sizes by empirically tuning over a range of promising candidate sizes. For 3-d grids, the large tile size along the last dimension effectively leaves it untiled so that prefetching and vectorization are efficient. For Pochoir, we used the default setup from the release since [27] presented experiments on a similar Intel Westmere-based Xeon and explored auto-tuning for it; its generated code was thus already tuned for our system.

Experimental setup: The hardware configuration we used is shown in Table 1. Intel’s C compiler (icc) 15.0.3 was used to compile all codes with flags shown in Table 1. Thread affinity type ‘scatter’ was chosen in order to evenly distribute threads across the entire system, and for reproducible results. Our arrays were defined as global variables; the memory allocation policy was the system default, which is to allocate on the local node for such variables. Although there is opportunity to exploit NUMA locality by considering memory allocation and thread affinity choices, this is beyond the scope of our work; we find that the chosen settings are reasonable for performance and reproducibility. Pochoir generated code was compiled with Intel’s C++ compiler (icpc) with the flags specified in its release: “-O3 -xHost -funroll-loops -fno-alias -fno-ffilas -fp-model precise”.

5.1 Benchmarks

The 1/2/3d-heat benchmarks solve the heat equations, and are examples of symmetric stencils. We evaluate the performance of discretized 1D, 2D and 3D heat equation stencils with non-periodic boundary conditions. Heat-1D is a 3-point stencil, while 2d-heat and 3d-heat are 5-point and 7-point stencils respectively. Conway’s Game of Life [28] is an 8-point stencil where the state of each point in the next time iteration depends on its 8 neighbors. We consider a particular version of the game called B2523, where a point is “born” if it has exactly two live neighbors, and a point survives the current stage if it has either two or three neighbors. APOP [29] is a one-dimensional 3-point stencil that calculates the price of the American put stock option. 3d27pt is an order-1 3-d 27-point stencil from the Berkeley auto-tuner framework [30] that performs 30 floating-point operations at each grid point. fdtd-2d is a 2-d finite difference time domain benchmark from Polybench/C (version 3.2) [31] involving stencil updates to multiple grids in an interconnected way. Except fdtd-2d, all benchmarks above were written as perfectly nested loops, with old and new
values stored as a 2-high dimension indexed with t%2 (as in Figure 1, 3). The problem size used for each benchmark is shown in Table 2; these sizes are similar to the ones used for evaluation in [27], and are meaningful for the computations represented. For fdtd-2d, the problem size used was Polybench’s (v3.2) extra large size. Results with a range of problem sizes for two of the benchmarks are presented in Appendix A. All benchmarks used double-precision floating-point data and computations.

### 5.2 Significance of Concurrent Start

We first demonstrate the significance of concurrent start by quantifying its benefits in greater detail for one of the cases – the 2d-heat benchmark. Although one can tweak tile sizes to reduce pipeline startup and drain phases and ensure enough tiles on the wavefront, this adjustment leads to reduced tile sizes; smaller tile sizes lead to reduced locality gains and a higher frequency of synchronization. In addition, all of this is highly dependent on the number of time iterations. For some problem sizes, one could be fortunate to find the right tile sizes such that the effect of pipeline start-up/drain is minimized. However, the diamond tiling approach does not suffer from this constraint; its load balance and scaling are only limited by data dimension extents.

For the 2d-heat benchmark, we found empirically that 64x64x64 was the best tile size configuration for locality. With pluto-standard, to fill the pipeline early and to ensure enough tiles on the wavefront, one can choose a tile size that sacrifices locality, for example, 16x64x64. We experimented pluto-diamond with tile size configurations of 64x64x64 and 16x64x64, and compared it with diamond tiled code with 64x64x64. The results are reported in Figure 13. pluto-diamond shows nearly ideal scaling. pluto-standard-16x64 shows good scaling unlike pluto-standard-64, but has lower single-thread performance. pluto-diamond performs 84% better than pluto-standard-64 and 39% better than pluto-standard-16x64.

### 5.3 Cache Miss Analysis

To further explain the behavior of various tiled versions, we collected hardware event-based cache miss data using Intel VTune (2015.3.0.403110) for each of the versions while running on a single core of the Intel system. The number of cache misses and the corresponding miss rates for all levels of the cache hierarchy are shown in Table 3. The data shows a strong correlation between the L3 cache misses and performance. pluto-standard with tile sizes 16x64 has clearly more L2 and L3 misses than with 64, pluto-diamond has the least number of accesses missing in the L2 cache as well as the L3 cache, and performs the best (single thread as well as in parallel). The low L2 miss rate of pluto-diamond shows that locality is mainly exploited for it at the L2 cache. Table 3 also shows cache miss data for Pochoir. In spite of Pochoir tiling recursively and having nearly the same number of L2 accesses as pluto-diamond, Pochoir has a significantly higher number of L2 and L3 misses than pluto-diamond. Although Pochoir’s cache-oblivious approach tiles by dividing the space and time dimensions recursively leading to a transparent tiling for all cache levels, the recursion step would imply that the effective tile size used to exploit locality at a level may not be the maximal fitting one. On the other hand, a cache-aware approach that explicitly sets a tile for a particular cache level for a maximal fit does not have this limitation. The tiling with Plato for all variants is for a single level, and the tiles are often L2 tiles – since an L1 tile would not be as efficient for vectorization. The gap in performance between the single core executions of these variants is expected to widen when running on multiple cores, owing to memory bandwidth limitations. This effect is evident when comparing their performance on 4 cores (last column of Table 3), and also in Figure 13 where ones with fewer L3 cache misses scale better. Appendix C in the supplemental material presents analysis for pluto-diamond-full, and results for a scenario where full-dimensional diamond tiling provides better performance than lower-dimensional one.

### 5.4 Performance Analysis

The execution times for different benchmarks and the improvement factors pluto-diamond obtains over other schemes are reported in Table 4. The improvement factors listed in the last three columns are for the case of parallel execution on all 12 cores for all schemes. pluto-diamond and pluto-standard have similar single-thread performance, but the former outperforms the latter on all benchmarks when running on more cores. icc-par is the icc auto-parallelized version obtained using the ‘-parallel’ compile flag in addition to the flags listed in Table 1. icc-par’s performance is lower and its scaling is much farther from ideal due to worse locality — this is supported by the number of cache misses in Table 3.
especially L2 and L3 misses, which are more than those for any other variant. pluto-diamond also performs better than Pochoir on all benchmarks; its better performance is for the reasons mentioned in Section 5.3. Figure 14 provides more information on the strong scaling trend.

For 2d-heat, we perform better than both Pochoir and pluto-standard (Figure 13). pluto-standard’s multi-core performance is poorer and does not increase beyond 8 cores. This is a result of both, pipeline startup/drain time, and an insufficient number of tiles in the wavefront to keep all processors busy. These limitations are eliminated by diamond tiling. The APOP and ‘Game of Life’ benchmarks involve integer operations and conditionals in the stencil operation, unlike the other benchmarks. For APOP, pluto-diamond performs 18% better than Pochoir. A similar trend was observed for game-of-life.

On the 3d-heat benchmark, pluto-standard does not provide sufficient parallelism beyond 5 cores, while both Pochoir and pluto-diamond scale better (Figure 14(a)) but not as well as they do for 2d-heat. In a few cases where Pochoir obtains an improvement over pluto-diamond — Pochoir uses dynamic scheduling through Cilk as opposed to the OpenMP loop-based parallelization realized by Pluto; the former leads to a better load balance when the number of tiles on the wavefront is not a multiple of the number of threads. Pochoir’s performance when scaling is thus also more predictable than that of pluto-diamond. In addition to load balance, dynamic schedule can also improve locality, and we plan to incorporate such techniques in the future. For 3d27pt (Figure 14(b)), we obtain performances of 42.6 GFLOPs (1.40 GStencils/s) on 12 cores. An improvement of 40% was obtained over Pochoir for 3d27pt (Table 4).

The varying trend in performance with Pluto tiled code while scaling is due to the same reason as that stated for 3d-heat. Polybench’s fdtd-2d is expressed in the form of a stencil involving updates to multiple grids representing electrical and magnetic fields. The extension to multiple statements (Section 4.5 and Algorithm 2) allow diamond
tiling to be applied in such cases and provide benefits over \textit{pluto-standard} similar to that for other benchmarks, while current domain-specific stencil compilers (Pochoir and SDSL) cannot handle stencils written in this manner.

6 Related work

A significant amount of work has been done on optimizing stencil computations. One body of works is in the form of developing compiler optimizations [10], [11], [32], [33] that help stencil computations, and the other on building domain-specific stencil compilers or manual optimization studies [14], [27], [30], [34], [35], [36], [37]. Among the compiler techniques, the ones based on polyhedral framework have been implemented in some production compilers [38], [39], and are also available as libraries and tools [23], [40]. Among domain-specific language (DSL) and optimization works specific to stencils, SDSL [36] and Pochoir [27] are publicly available ones. Such systems have the opportunity to provide higher performance than compiler-based ones owing to a greater amount of knowledge their compilers have about the computation and a greater freedom in choosing transformations.

Hodzic and Shang [41] used the notion of minimizing critical path to determine optimal tile shapes, in contrast to other past work that minimized communication or the ratio of communication to computation [42], [43]. The notion of critical path is more powerful than that of enabling concurrent start, and minimizing critical path may indirectly imply enabling concurrent start for stencils with hyper-rectangular iteration spaces. Interestingly, our approach can be seamlessly used if a different vector based on the critical path is chosen in place of $f_3$ (the concurrent start face); the tile schedule with then be pointed in the direction of the critical path vector once the diamond tiling hyperplanes are found. This could be useful for example for a weirdly shaped iteration space where the critical path may not be normal to the concurrent start face. In contrast to our fast ILP-based approach, Hodzic and Shang [41] modeled the critical path minimization problem as a non-linear programming one in continuous space, solving it through an iterative non-linear optimization approach to obtain the tiling hyperplanes — the results are thus approximate and not necessarily the desirable ones from an example provided therein.

Pochoir [27] uses a cache-oblivious tiling mechanism with tiles shaped like trapezoids. Such a tiling mechanism does not suffer from a pipelined startup. The input for Pochoir (and similarly SDSL [36]) is a DSL embedded in C++ using templates. This custom input can be more expressive in some ways, for example, to specify boundary conditions. Our approach, on the other hand, is a dependence-driven one; it does not require any additional meta information, and works for any affine loop nest. It can thus also naturally deal with multi-statement stencils (like fftd-2d evaluated in Section 5), while Pochoir and SDSL both do not support stencils defined using multiple grids. Experimental evaluation we presented included a comparison with Pochoir and an analysis of the advantages and disadvantages of cache-oblivious tiling in Section 5.3.

Krishnamoorthy et al. [11] developed the notion of point-wise and tile-wise concurrent start when tiling stencil computations, and proposed two approaches to allow tile-wise concurrent start. The approaches worked by starting with a valid tiling and correcting it to allow concurrent start. This was done via overlapped execution of tiles (overlapped tiling) or by splitting a tile into sub-tiles (split tiling). With such an approach, one misses natural ways of tiling that inherently do not suffer from pipelined startup. We have showed that, in all those cases, there exist valid tiling hyperplanes that allow concurrent start. In addition, our conditions for concurrent start work with arbitrary affine dependences while those by Krishnamoorthy et al. [11] were presented for dependences expressed as constant distance vectors. A problem with overlapped and split tiling is the difficulty in performing code generation automatically – this is in particular hard for overlapping tiles on shared-memory parallel systems, where tile local buffers are needed to run in parallel. In addition, redundant computation performed for overlapped tiling can be significant for 3-d stencils with a large time tile size. Although an implementation of overlapped tiling with local buffers exists in a stencil code generator for GPUs [44], with OpenCL [45], and in domain-specific compilers for image processing pipelines [46], [47], there has not been any evaluation or comparison with other state-of-the-art techniques for time-iterated stencils on general-purpose multicores.

At least three past works, that of Eissfeller and Muller [12], Orozco and Gao [13], and Strzodka et al. [14] have explored diamond tiling to enable concurrent start, although without a formalism to determine, represent, and apply diamond tiling automatically. The formalism that we developed here allows us to state why certain tile shapes provide concurrent start, and how these shapes can be specified for an arbitrary stencil. Using the polyhedral framework for the formulation allowed us to also automatically generate code without the need for a special code generation technique. Both Orozco and Gao [13] and Strzodka et al. [14] use diamond tiling with concurrent start along only one dimension. Orozco and Gao [13] reported results on 2-d FDTD. Strzodka et al. [14], [48]'s technique, cache accurate time skewing (CATS), obtains significant improvement over Pluto and other simpler manual optimization strategies for stencils on 2-d and 3-d domains. Their scheme also pays attention to additional orthogonal aspects such as mapping of tiles to threads and is presumably more cache-friendly than our scheme. We plan to explore these complementary aspects and integrate them into our technique in future. While ours is an end-to-end automatic compiler framework, CATS is a customized optimization system. Shaheen and Strzodka [48] also consider improving locality in the presence of NUMA effects – an aspect we do not explicitly model.

Di et al. [49] propose alternate tiling, which addresses the issue of pipelined startup with Gauss-Seidel relaxations through a specialized decomposition of the computation and data; without such a decomposition, Gauss-Seidel relaxations do not satisfy the point-wise concurrent start property, and diamond tiling is not applicable. Exploring diamond tiling in conjunction with such a decomposition is an interesting direction to explore.

Grosser et al. [19] propose a hybrid tiling scheme that combines a hexagonal shape along the time dimension and
one space dimension with classical tiling along the remaining space dimensions. The approach is proposed and evaluated for GPUs for which it has special benefits. In particular, the freedom to choose the width of the tile independently with respect to the time tile height provides only more flexibility: diamond tiling can be seen as its special case. However, its benefits in comparison to diamond tiling on general-purpose multicores are yet to be evaluated. Our final experimental comparison (in Figure 14 and Table 4) used one-dimensional concurrent start as it was found to provide sufficient parallelism for general-purpose multicores and deal with the “width” issue (Section 4.2). However, within a tile, hexagonal tiling allows better intra-tile parallelization along one more dimension than diamond tiling with one-dimensional concurrent start. Grosser et al. [52] compares diamond and hexagonal tiling while analyzing the effects of tile size and tile schedule choices on tile-level parallelism and compute-to-communication ratio: this allows them to determine optimal tile size ratios. Additional discussion on the effect of tile sizes was provided in Section 4.6.

Efficient vectorization of stencils is challenging. Henretty et al. [50] develop a data layout transformation technique to improve the effectiveness of vectorization for stencils. However, improvement reported was limited for architectures that provide nearly the same performance for unaligned loads as for aligned ones. For this work, we relied on the auto-vectorization capabilities of Intel’s compiler or GCC, both of which were found to be satisfactory on the code we generated.

Wonnacott and Strout [17] analyze various tiling techniques for stencils emphasizing on the ability to exploit locality while providing parallelism that scales with data. Although our approach was motivated by concurrent start, interestingly, for the cases evaluated here (stencil computations with hyperrectangular iteration spaces), tile-wise concurrent start and asymptotic scaling go together. The face allowing concurrent start for stencils is orthogonal to all data dimensions, and hence, full diamond tiling, i.e., diamond tiling that enables concurrent start along all dimensions of the face, provides parallelism that scales when scaling data size along any dimension.

Shrestha et al. [51] improve and specialize diamond tiling further to allow concurrent start hierarchically – this is beneficial when there are multiple levels of parallelism. The authors develop an approach that uses jagged polygon shaped tiles that includes finer grained L1 tiles within larger L2 tiles to map to two levels of a threading hierarchy. Improvements are reported on an Intel Xeon Phi accelerator. Malas et al. [52] utilize both diamond tiling and wavefront blocking to achieve multi-dimensional parallelization; diamond tiling is used to exploit L3 reuse and parallelism at the outer level, and in addition, wavefront parallelization is performed along multiple dimensions inside a tile. Significant improvements are demonstrated in certain cases over the diamond tiling described here.

Ghysels and Vanroose [53] use time tiling techniques to increase the arithmetic intensity of the pre-smoothing phase involved in solving a partial differential equation using the Geometric Multigrid Method. They also study the relationship between the number of pre-smoothing steps, convergence properties, and time to solution. The work reveals the importance of optimizing stencils with a few time iterations, say from four steps to up to a few tens of time steps. In such cases, diamond tiling allows extraction of scalable parallelism, while classical time skewing will result in too few tiles to run in parallel or a complete loss of parallelism.

Diamond tiling provides speedups similar to the ones reported here for stencils defined over periodic data domains as well [21], although the latter requires additional techniques to enable time tiling itself. With additional abstractions, time tiling techniques including diamond tiling have also been applied to Lattice-Boltzmann method simulations to provide significant speedups [54].

7 Conclusions

We have designed and evaluated new techniques for tiling stencil computations that, in addition to providing the usual benefits of tiling, enable concurrent start-up whenever possible. We showed that tile-wise concurrent start is enabled if the face of the iteration space that allows point-wise concurrent start strictly lies in the cone formed by all tiling hyperplanes. We then provided an approach to find such hyperplanes. The presented techniques are automatic and have been implemented in a source-level parallelizer, Pluto. Experimental evaluation on a 12-core Intel multicore shows that our code is able to outperform a tuned domain-specific stencil code generator by about 10% to 40%. In other cases, we outperform previous compiler techniques by a factor of 1.3× to 10.1× on 12 cores over a set of benchmarks.

References


Uday Bondhugula is an Assistant Professor at the Indian Institute of Science (IISc), Bangalore, India. Prior to joining IISc, he was a postdoctoral researcher with the Advanced Compiler Technologies group at IBM TJ Watson Research Center, New York. He obtained his Ph.D. in Computer Science and Engineering from the Ohio State University, USA in 2008, and his bachelor’s degree in Computer Science and Engineering from the Indian Institute of Technology, Madras in 2004.

Vinayaka Bandishtı obtained his master’s degree in Computer Science and Engineering at the Indian Institute of Science, Bangalore, India in 2014, and his bachelor’s degree in Computer Science and Engineering from SJCE, Mysore, India in 2011. He currently works as a software engineer at Veevo Inc. in Bangalore, India.

Irshad Pananilath obtained his master’s degree in Computer Science and Engineering at the Indian Institute of Science (IISc), Bangalore, India in 2014. Prior to joining IISc, he worked as a solution architect at Fujitsu Consulting Pvt. Ltd., Hyderabad. He obtained his B.Tech. degree in Computer Science and Engineering from the National Institute of Technology, Calicut, India in 2005. Currently, he is the CEO of a technological startup.
SUPPLEMENTAL MATERIAL

APPENDIX A
VARYING PROBLEM SIZES

Experimental results with different problem sizes for two benchmarks (heat-2d and heat-3d) are shown in Figures 15 and 16. We vary $N$, the extent along the data dimensions, and keep $T$ fixed to the value reported in Table 2. We did not consider it meaningful to increase $T$ further. Lower values of $T$ will further reduce the amount of parallelism available for pluto-standard. Tile sizes used are same as those reported in Table 2. We show results for sequential execution and while running on all 12 cores, while varying $N$ in both cases. Note that the problem sizes are all hypercubic, i.e., the data grid is $NxN$ for 2-d and $NxNxN$ for 3-d.

The lower performance observed for smaller problem sizes for heat-2d parallel execution of pluto-diamond is due to an insufficient number of tiles in those cases — although the tile size could be reduced, for uniformity, we choose to keep the tile size fixed to the one that is best for locality, irrespective of the number of cores or problem size. A similar issue is not observed for heat-3d since the outer tile sizes are much smaller (Table 2). The variation in performance of pluto-diamond for parallel execution in contrast to that of Pochoir is due to the same reasons stated in Section 5.4. A larger problem size yields more tiles to run in parallel for pluto-diamond (and up to a certain extent for pluto-standard as well), mitigating imbalance caused by the number of tiles not being a multiple of the number of threads. We observe that pluto-diamond performs better than Pochoir for both heat-2d and heat-3d due to consistently better single thread performance (as seen in Figure 15(a) and 16(a)) — the reasons behind which were stated in Section 5.3.

We observe from Figure 15(a) and 16(a) that the single thread performance of pluto-standard and pluto-diamond is nearly the same across all problem sizes — both for heat-2d and heat-3d. However, as expected, in the case of parallel execution, we see a difference in performance for all problem sizes. A larger problem size does not lead yield more tiles to run in parallel for pluto-standard since the number of time steps is fixed to a value that is relatively smaller. We observe a drop in performance of pluto-standard and pluto-diamond for heat-2d, $N = 16E3$ for both sequential and parallel cases. We found that this was due to conflict misses for this specific size; we confirmed that padding the array by a small factor brought the performance back on par with that of neighboring problem sizes (64 GFLOPs). In summary, we observe that pluto-diamond delivers similar improvements in performance across different problem sizes.

APPENDIX B
HIGHER ORDER STENCILS

All benchmarks evaluated in Section 5 were order-1 stencils. Diamond tiling is as beneficial for higher order stencils: with such stencils, the tile shapes are just more oblique, i.e., the slopes of the tiling hyperplanes are different than that for first order stencils. This is also the case with standard parallelogram tiling (pluto-standard). The benefits of concurrent start with pluto-diamond and locality enhancement remain.

Malas et al. [52] provide an extensive evaluation while including a 25-point stencil and diamond tiling as one of the schemes for comparison.

As an example, consider a 25-point stencil on a 3-d grid where values from four neighbors along both positive and negative directions, for each dimension, are used. The diamond tiling transformation (with one-dimensional concurrent start) in this case is given by:

$$T((t, i, j, k)) = (4t - i, 4t + i, 4t + j, 4t + k),$$

in contrast to the one in (16) for an order-1 3-d stencil.

APPENDIX C
FULL VS LOWER-DIMENSIONAL DIAMOND TILING

Although full diamond tiling maximizes parallelism and provides asymptotic scalability [17], as discussed in Section 4.2, it is not expected to be as effective for single-thread performance as pluto-diamond or pluto-standard. The data in Table 3 supports this. A much larger tile size was needed for pluto-diamond-full to obtain benefits of auto-vectorization and prefetching: we found that the best tile size for it was 128x128x128. Cache performance is thus exploited primarily at the L3 level; this is evidenced by a greater number of its L3 accesses being hits when compared to other tiled versions, and its L3 miss rate being the lowest at 6.40%. The additional parallelism pluto-diamond-full provides in this case cannot be leveraged, and pluto-diamond itself provides sufficient parallelism to keep all cores busy (Figure 13). In fact, given the problem sizes, we observe that full diamond tiling would be useful only when we have a very large number of cores, for example, of the order of 50 cores or more for the problem size we have.

To set up a scenario where pluto-diamond-full may be preferred over pluto-diamond, we performed an additional level of tiling (for the L3 cache), with tile size $512^3$ for both versions. We chose a problem size of 1500x1500 with 1000 time steps. Although reducing the problem size would have been another alternative, it would require us to set a problem size small enough to fit in cache, in turn making the comparison undesirable. The additional level of tiling improves single-thread performance from 5.5 to 6.2 GFLOPs for pluto-diamond and marginally for pluto-diamond-full. The results are shown in Figure 17. Nested OpenMP parallelism was used for pluto-diamond-full and cores were factored across both dimensions evenly — for example, 3x3 when running on 9 cores, 4x3 on 12 cores. Beyond seven cores, we observe that pluto-diamond runs out of tiles to run in parallel while pluto-diamond-full continues to scale and make use of available parallelism. The plots are not expected to be smooth as the number of parallel loop iterations available may not be a multiple of the number of cores — this makes it particularly more difficult with nested OpenMP pragmas, where the resulting imbalance is exacerbated depending on how one factors threads across the two nested OpenMP parallel for pragmas. This problem can be addressed if omp collapse is supported for triangular loops, which is currently not the case. We thus believe that, full diamond tiling will lead to a better parallelization in spite of a slightly lower single-thread performance as the degree of hardware parallelism increases.
Fig. 15. Heat-2D: varying problem sizes

Fig. 16. Heat-3D: varying problem sizes

Fig. 17. Full diamond tiling (2-d nested parallelism) vs partial diamond tiling (1-d parallelism) on Heat 2D (problem size of 1500x1500 with 1000 time steps and two levels of tiling)