Tiling Stencil Computations to Maximize Parallelism

Vinayaka Bandishti, Irshad Pananilath, and Uday Bondhugula
Department of Computer Science and Automation
Indian Institute of Science, Bangalore 560012
{vbandishti,irshad,uday}@csa.iisc.ernet.in

***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 hyperplanes 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 that ensures concurrent start-up as well as perfect load-balance whenever possible. We first provide necessary and sufficient conditions on tiling hyperplanes to enable concurrent start for programs with affine data accesses. We then provide an approach to find such hyperplanes. Experimental evaluation on a 12-core Intel Westmere shows that our code is able to outperform a tuned domain-specific stencil code generator by 4% to 27%, and previous compiler techniques by a factor of 2× to 10.14×.**

***Index Terms*—Compilers, Program transformation**

I. Introduction and Motivation

Stencils are a very common class of programs appearing in many scientific and engineering applications that are computationally intensive. They are characterized by regular computational structure and hence allow automatic compile-time analysis and transformation for exploiting data-locality and parallelism.

Stencil computations are characterized by update of a grid point using neighboring points. Figure 1 shows a stencil over a one-dimensional data space used to model, for example, a 1-d heat equation. They exhibit a number of properties that lend themselves to optimization. Locality optimization and parallelization are the most important among them. Loop tiling [1], [36], [38] is a key transformation used to exploit data locality and parallelism from stencil computations.

```
for (t = 1; t <= T; t++) {
  for (i = 1; i < N+1; i++) {
    S1: B[i] = 0.125 * ( A[i+1] - 2.0 * A[i] + A[i-1] );
  }
  for (i = 1; i < N+1; i++) {
    S2: A[i] = B[i];
  }
}
```

Fig. 1. Stencil: 1d heat equation

Loop tiling is often characterized 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* [19], [2], [28], [5]. More formal definitions are provided in the next section. Finding the right shape and size are the subject of numerous works with goals of improving locality, controlling frequency of synchronization, and volume of communication where applicable. Performing parallelization and locality optimization together on stencils can often lead to pipelined startup, i.e., not all processors are busy during parallelized execution. This is the case with a number of general compiler techniques from the literature [23], [14], [5]. With increasing number of cores per chip, it is very beneficial to maintain load balance by enabling concurrent start of tiles along an iteration space boundary whenever possible. Concurrent start-up for stencil computations not only eliminates pipeline fill-up and drain delay, but also ensures perfect load-balance. Processors end up executing the same maximal amount of work in parallel between two synchronization points.

Some works have looked at eliminating pipelined startup [37], [22] by tweaking or modifying already obtained tile shapes from existing frameworks. However, these approaches have undesired side-effects including difficulty in performing code generation. No implementations of these have been reported to date. The approach we propose in this paper works by actually searching for tiling hyperplanes that have the desired property of concurrent start, instead of fixing or tweaking hyperplanes found with undesired properties. To the best of our knowledge, prior to this work, it was not clear if and under what conditions such hyperplanes existed, and how they could be found. In addition, their performance on aspects other than concurrent start in comparison with existing compiler techniques has to be studied, though the work of Strzodka et al. [33] does study this but in a more specific context than we intend to here. A comparison of compiler-based and domain-specific stencil optimization efforts has also not been performed in the past. We address all of these in this paper. In summary, our contributions are as follows:

- Provide conditions under which tiling hyperplanes allow concurrent start without the need for further tweaking and in the presence of arbitrary affine dependences.
- An approach to finding such tiling hyperplanes.
- An experimental evaluation of our technique and comparison with the current state-of-the-art from compiler works as well as tuned domain-specific ones.

The rest of the paper is organized as follows. Section II

SC12, November 10-16, 2012, Salt Lake City, Utah, USA
978-1-4673-0806-9/12/$31.00 ©2012 IEEE

provides background and introduces notation used. Section III characterizes concurrent start conditions. Section IV describes our approach to find solutions with the desired properties. Section V presents experimental evaluation and conclusions are presented in Section VII.

II. Background and Notation

A. Outer parallelism and concurrent start

Achieving concurrent start in iteration spaces that have at least one degree of outer parallelism or communication-free parallelism is fairly simple. Finding concurrent start becomes complicated when dependences span the entire iteration space. In this paper, since we consider stencils, we are looking at programs which have no outer parallelism. However, there still exists opportunity to start concurrently along at least one face of the iteration space.

B. Characterizing a stencil program

We assume and exploit the following inherent properties of stencil computations.

- As a grid point is always updated using values from neighboring grid-points in recent time steps, the loop nests always have a 'time-step' loop as the outermost loop.
- Entire space-time grid of $d+1$ dimensions uses an array of d dimensions, outermost index representing different time steps.
- They can always be written in single statement form as:

$$value_p[t+1] = f(value_p[t], value_{neighbors\_of(p)}[t], \dots)$$

C. Conic and strict conic combination

A *conic combination* of a set of vectors $\vec{x_1}, \vec{x_2}, \dots, \vec{x_n}$ is a vector of the form

$$\lambda_1\vec{x_1} + \lambda_2\vec{x_2} + \cdots + \lambda_n\vec{x_n} \quad (1)$$
$$\lambda_i \geq 0.$$

If all λs are strictly positive, we call (1) a *strict conic combination* of $\vec{x_1}, \vec{x_2}, \dots, \vec{x_n}$.

D. Dependences and tiling hyperplanes

Let S_1, S_2, ..., S_n be the statements of a program, and let $\mathbf{S} = \{S_1, S_2, \dots, S_n\}$. The iteration space of a statement can be represented as a polyhedron. The dimensions of the polyhedron correspond to surrounding loop iterators as well as *program parameters*. Program parameters are symbols that do not vary in the portion of the code we are representing; they are typically the problem sizes. Each integer point in the polyhedron, also called an iteration vector, contains values for induction variables of loops surrounding the statement from outermost to innermost. The *data dependence graph*, $DDG = (\mathbf{S}, E)$ 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 P_e, called *dependence polyhedron* which precisely captures all the dependences between the dynamic instances of S_i and S_j. One can obtain a less powerful representation such as a constant distance vector or direction vector from dependence polyhedra by analyzing the relation between source and target iterators. For all examples in the paper, we use constant distance vectors for simplicity. The reader is referred to [4] for a more detailed explanation of the polyhedral representation.

A hyperplane is an $n-1$ dimensional affine subspace of an n dimensional space. For example, any line is a hyperplane in a 2-d space, and any 2-d plane is a hyperplane in 3-d space. A hyperplane for a statement S_i is of the form:

$$\phi_{S_i}(\vec{x}) = \vec{h}\cdot\vec{x} + h_0 \quad (2)$$

where h_0 is the translation or the constant shift component, and $\vec{x}$ is an iteration of S_i. $\vec{h}$ itself can be viewed as a vector oriented in a direction normal to the hyperplane.

Prior research [24], [5] provides conditions for a hyperplane to be a valid tiling hyperplane. For $\phi_{s_1}, \phi_{s_2}, \dots, \phi_{s_k}$ to be valid statement-wise tiling hyperplanes for $S_1, S_{2,}, \dots, S_k$ respectively, the following should hold for each edge e from S_i to S_j:

$$\phi_{s_j}(\vec{t}) - \phi_{s_i}(\vec{s}) \geq 0, \quad \langle\vec{s}, \vec{t}\rangle \in P_e, \forall e \in E \quad (3)$$

The above constraint implies that all dependences have non-negative components along each of the hyperplanes, i.e., their projections on the these hyperplane normals are never in a direction opposite to that of the hyperplane normals.

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 loop nest dimensionality.

Among the many possible hyperplanes, the optimal solution according to a cost function is chosen. A cost function that has worked in the past is based on minimizing dependence distances lexicographically with hyperplanes being found from outermost to innermost [5].

If all iteration spaces are bounded, there exists an affine function $v(\vec{p}) = u\cdot\vec{p} + w$ that bounds $\delta_e(t)$ for every dependence edge e:

$$v(\vec{p}) - (\phi_{s_i}(\vec{t}) - \phi_{s_j}(\vec{s})) \quad \geq \quad 0, \quad \langle\vec{s}, \vec{t}\rangle \in P_e, \forall e \in E \quad (4)$$

where $\vec{p}$ is a vector of program parameters, i.e., symbols appearing in loop bounds and accesses; these often represent problem sizes. The coefficients u, w are then minimized.
This ensures the following:

- In the transformed space, all dependences have non-negative components along all bases.
- Any rectangular tiling in the transformed space is valid.

```
for (t=0; t<T; t++)
 for (i=2; i<N-3; i++)
   S1: A[t+1][i] = (A[t][i-2] + A[t][i] + A[t][i+2])/3.0;
```

Fig. 2. Example program

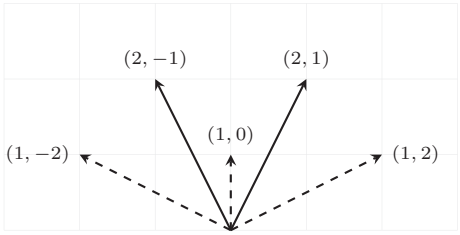


Fig. 3. $\vec{d_1} = (1, 2)$, $\vec{d_2} = (1, 0)$ and $\vec{d_3} = (1, -2)$ are the dependences. For any hyperplane $\vec{\phi}$ in the cone formed by (2,1) and (2,-1),$\vec{\phi} \cdot d_1 \geq 1 \quad \wedge$ $\vec{\phi} \cdot d_2 \geq 1 \ \wedge \ \vec{\phi} \cdot d_3 \geq 1$ holds good. Cost function chooses (1,0) and (2,1) as tiling hyperplanes.

Example: Consider the program in Figure 2. As all the dependences are self dependences and also uniform, they can be represented by constant vectors. The dependences are:

$$\begin{pmatrix} \vec{d_1} & \vec{d_2} & \vec{d_3} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 \\ -2 & 0 & 2 \end{pmatrix}$$

Equation 3 can now be written as

$$\phi \cdot \begin{pmatrix} 1 & 1 & 1 \\ -2 & 0 & 2 \end{pmatrix} \quad \geq \quad \begin{pmatrix} 0 & 0 & 0 \end{pmatrix} \tag{5}$$

Equation 5 forces the chosen hyperplane to have non-negative components along all dependences. Therefore, any hyperplane in the cone of (2,-1) and (2,1) is a valid one (Figure 3). However, cost function (4) ends up choosing (1,0) and (2,1)

E. Rectangular tiling and inter-tile dependences

As mentioned earlier, once the transformation is applied, the transformed space can be tiled rectangularly. Hence, by assuming that inter-tile dependences in the transformed space are unit vectors along all the bases, we can be sure that any inter-tile dependence is satisfied (Figure 4). It is important to note here that this approximation is actually accurate for stencils i.e., it is also necessary that we consider inter-tile dependences along every base as the dependences span the entire iteration space (there is no outer parallelism). In the rest of the paper we refer to these safely approximated inter-tile dependences as simply *inter-tile dependences*. Thus if F' is the matrix whose columns are the approximated inter-tile dependences in the transformed space, F' will be a unit matrix.

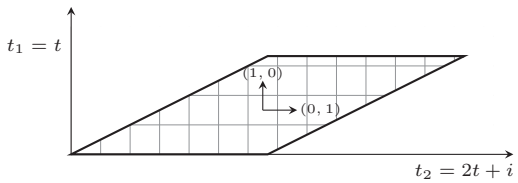


Fig. 4. Transformed iteration space after applying (1,0) and (2,1) as the tiling hyperplanes. Approximating unit inter-tile dependences along every base accounts for any actual inter-tile dependence.

Let T_R be the *reduced transformation matrix* which has only the $\vec{h}$ components (Eqn. 2) of all hyperplanes as rows. T_R of any statement is thus a square matrix which can be obtained by eliminating the columns producing translation and any other rows meant to specify loop distribution at any level [21], [10], [5]. As the inter-tile dependences are all intra-statement and are not affected by the translation components of tiling hyperplanes, we have:

$$F' = T_R \cdot F$$

where F corresponds to approximate inter-tile dependences in original iteration space.

As mentioned earlier, F' can be safely considered a unit matrix. Therefore,

$$F = T_R^{-1}$$

Example:

The tiling dependences introduced by hyperplanes (1,0) and (2,1) can be found as follows:

$$\begin{pmatrix} \vec{c_1} & \vec{c_2} \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 2 & 1 \end{pmatrix}^{-1} = \begin{pmatrix} 1 & 0 \\ -2 & 1 \end{pmatrix}$$

Thus, the inter-tile dependences are (1,-2) and (0,1). The tiling dependences introduced in the iteration space of a statement solely depend on the hyperplanes chosen for that statement. Also, note that every inter-tile dependence, given the way we defined it, is carried by only one of the tiling hyperplanes and has a zero component along any other tiling hyperplane.

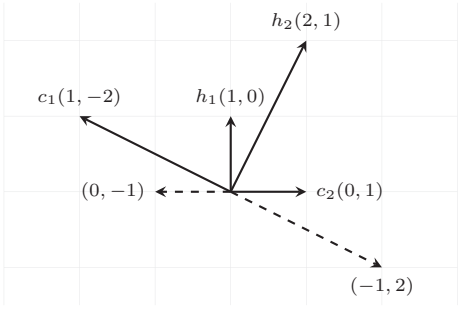


Fig. 5. Inter-tile dependences for the example in Figure 2 introduced by the hyperplanes (1,0) and (2,1)

Consider Figure 5. The tiling hyperplanes in the figure are (1,0) and (2,1). Let us find the inter-tile dependence carried by (1, 0). There are only two unit vectors in the orthogonal subspace of the other hyperplane (2, 1). The only two dependences which have zero component along (2,1) are (1,-2) and (-1,2). Among these two, the one which is satisfied by (1,0) is (1,-2). So, the inter-tile dependence satisfied by (1,0) is (1,-2). Similarly, the dependence satisfied by (2,1) is (0,1).

In the above example, concurrent start along (1,0) is prevented by the tile dependence (0,1). There does not exist a face along which the tiles can start in parallel (Figure 6), i.e., there would be a pipelined start-up and drain phase. Decreasing the tile size would decrease the start-up and drain phase but would increase the frequency of synchronization. Increasing the tile

size would mean a shorter steady-state. In summary concurrent start-up is lost.

Transform should be found in such a way that it does not introduce any inter-tile dependence that prohibits concurrent start. If we had chosen (2,-1), which is also a valid, instead of (1,0) as tiling hyperplane, i.e., (2,-1) and (2,1) were chosen as the tiling hyperplanes, then the inter-tile dependences introduced would be (1,-2) and (1,2) (Figure 7). Both have positive components along the normal (1,0), i.e., all tiles along the normal (1,0) could be started concurrently.

In the next section, we provide conditions on hyperplanes to avoid (1,0) and select (2,-1) instead.

III. Tiling for concurrent start

If all the iterations along a face can be started concurrently, the face is said to allow point-wise concurrent start. Similarly, if all the 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 $\vec{f}$. 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 and Theorem 3 map Theorem 1 from inter-tile dependences onto tiling hyperplanes. We also prove that these constraints are both necessary and sufficient for concurrent start.

A. Constraints for concurrent start

Theorem 1: *For a statement, a transformation enables tile-wise concurrent start along a face $\vec{f}$ iff the tile schedule is in the same direction as the face and carries all inter-tile dependences.*

Let t^o be the outer tile schedule. If $\vec{f}$ is the face allowing concurrent start and C is the matrix containing approximate inter-tile dependences of the original iteration space, then,

$$k_1\vec{t^o} = k_2\vec{f}, \quad k_1, k_2 \in \mathbf{Z}^+$$
$$\vec{t^o}\cdot C \geq \vec{1}$$

Hence,

$$\vec{f}\cdot C \geq \vec{1}$$

In Figure 8, the face allowing the concurrent start and outer tile-schedule are the same and carry both the tile-dependences. Therefore, the tiles t_1, t_2, t_3, t_4 can be started concurrently.

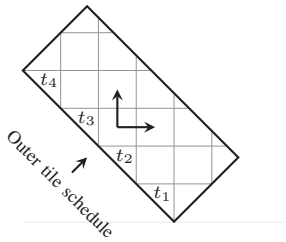


Fig. 8. Tile schedule and normal to the face should be in the same direction.

Theorem 2: *Concurrent start along a face $\vec{f}$ can be exposed by a set of hyperplanes iff $\vec{f}$ lies strictly inside the cone formed by the hyperplanes, i.e., iff $\vec{f}$ is a strict conic combination of all the hyperplanes.*

$$k\vec{f} = \lambda_1\vec{h_1} + \lambda_2\vec{h_2} + \cdots + \lambda_n\vec{h_n} \tag{6}$$
$$\lambda_i, k \in \mathbf{Z}^+$$

Proof (sufficient): As mentioned earlier every inter-tile dependence is satisfied by only one hyperplane and has zero component along every other hyperplane. Consider the following expression :

$$\lambda_1(\vec{h_1}\cdot\vec{c}) + \lambda_2(\vec{h_2}\cdot\vec{c}) + \cdots + \lambda_n(\vec{h_n}\cdot\vec{c}) \tag{7}$$

For any tile-dependence $\vec{c}$, as we constrain all the λs to be strictly positive, (7) will always be positive. Thus, by choosing all $\vec{h}$ such that

$$k\vec{f} = \lambda_1\vec{h_1} + \lambda_2\vec{h_2} + \cdots + \lambda_n\vec{h_n}$$

we ensure $\vec{f}\cdot\vec{c} \geq 1$ for all inter-tile dependences. Therefore, from **Theorem 1**, concurrent start is enabled. □

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., (6) does not hold good. Without loss of generality, we can assume that $k \in Z^+$, but there exist no λs which are all strictly positive integers. For at least one inter-tile dependence $\vec{c}$, the sum (7) ≤ 0 because every tile dependence is carried by only one of the chosen hyperplanes and there exists at least one λ such that $\lambda \leq 0$. Therefore, $\vec{f}\cdot\vec{c}$ will also be zero or negative, which means concurrent start is inhibited along $\vec{f}$. This is a contradiction. □

For the face $\vec{f}$ that allows concurrent start, let $\vec{f'}$ be its counterpart in the transformed space. We now provide an alternative result for concurrent start that is equivalent to the one above.

Theorem 3: *A transformation T allows concurrent start along $\vec{f}$ iff $\vec{f}\cdot T_R^{-1} \geq \vec{1}$.*

Proof: From Theorem 1 we have, for the transformed space, the condition for concurrent start becomes

$$\vec{f'}\cdot C' \geq \vec{1} \tag{8}$$

We know that in the transformed space, we approximate all inter-tile dependences by unit vectors along every base. Since the normals are not affected by translations, the normal $\vec{f'}$ after transformation T is given by $\vec{f}\cdot T_R^{-1}$. Also, C' is a unit matrix.

Therefore, (8) becomes

$$\vec{f}\cdot T_R^{-1} \geq \vec{1}$$

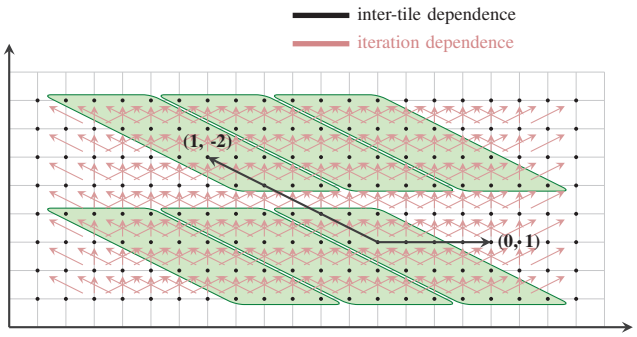


Fig. 6. Pipelined start-up

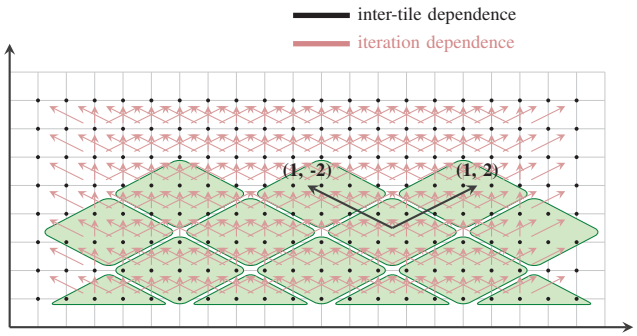


Fig. 7. Concurrent start-up

The above can be viewed in a different manner. We know that the inter-tile dependences introduced in the original iteration space can be approximated as

$$C = T_R^{-1}$$

From Theorem 1 we have,

$$\vec{f} \cdot T_R^{-1} \geq \vec{1} \qquad \square$$

B. Partial concurrent start

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. But, in practice, exploiting all of these degrees may result in more complex code. The above conditions can be placed only on the first few hyperplanes to obtain what we term *partial concurrent start*. For instance, the constraints can be placed only on the first two hyperplanes so that one degree of concurrent start is exploited. Such transformations may not only generate better code owing to prefetching and vectorization, but also achieve coarser grained parallelization.

C. The case of multiple statements

In case of multiple statements, every strongly connected component in the dependence graph can have only one outer tile schedule. So, all statements which have to be started concurrently and together should have the same face that allows concurrent start. If they do not have the same face allowing concurrent start, one of those has to be chosen for the outer tile schedule. These are beyond the scope of this paper, since in the case of stencils, the following are always true:

- Every statement's iteration space has the same dimensionality and same face $\vec{f}$ that allows concurrent start.
- Transitive self dependences are all carried by this face.

For example, for the stencil in Figure 1 (an imperfect nest with 2 statements), hyperplanes found by our scheme for S1 are (2,1, 0) and (2,-1, 0) and those for S2 are (2,1, 1) and (2,-1, 1) i.e., these correspond to 2t+i, 2t-i for S1, and 2t+i+1, and 2t-i+1 for S2.

IV. Approach for finding hyperplanes iteratively

We now present a scheme that is an extension to the Pluto algorithm [5] so that hyperplanes that satisfy properties introduced in the previous section are found. Though Theorem 2 and Theorem 3 are equivalent to each other, we use Theorem 2 in our scheme as it provides simple linear inequalities and is easy to implement.

A. Iterative scheme

Along with constraints imposed to respect dependences and encode an objective function, the following additional constraints are added while finding hyperplanes iteratively.

For a given statement, let $\vec{f}$ be the face along which we would like to start concurrently, H be the set of hyperplanes already found, and n be the dimensionality of the iteration space (same as the number of hyperplanes to find). Then, we add the following additional constraints:

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.,*

$$k\vec{h_n} = \lambda'\vec{f} + \lambda_1(-\vec{h_1}) + \lambda_2(-\vec{h_2}) + \cdots + \lambda_{n-1}(-\vec{h_{n-1}}), \tag{9}$$
$$\lambda_i, k \in \mathbf{Z}^+$$

In Algorithm 1, we show only our additions to the iterative algorithm proposed in [5].

If it is not possible to find hyperplanes with these additional constraints, we report that tile-wise concurrent start is not possible. If there exists a set of hyperplanes that exposes tile-wise concurrent start, we now prove that the above algorithm will find it.

Proof: (soundness)
When the algorithm returns a set of hyperplanes, we can be sure that all of them are independent of each other and that they also satisfy Equation (9) which is obtained by just rearranging the terms of Equation (6). Trivially, our scheme

Algorithm 1 Finding tiling hyperplanes that allow concurrent start

1: Initialize $H = \varnothing$
2: **for** $n-1$ times **do**
3: Build constraints to preserve dependences.
4: **Add constraints such that the hyperplane to be found is linearly independent of** $\vec{f} \cup H$
5: Add cost function constraints and minimize cost function for the optimal solution.
6: $H = \vec{h} \cup H$
7: **end for**
8: **Add constraint (9) for the last hyperplane so that it strictly lies inside the cone of the face and negatives of the** $n-1$ **hyperplanes already found** (H)

of finding the hyperplanes is sound, i.e., whenever our scheme gives a set of hyperplanes as output, the output is always correct. □

Proof: (completeness)

We prove by contradiction that whenever our scheme reports no solution, there does not exist any valid set of hyperplanes. Suppose there exists a valid set of hyperplanes but our scheme fails to find them. The scheme can fail at two points.

Case 1: While finding the first $n-1$ hyperplanes

The only constraint the first $n-1$ hyperplanes have is of being linearly independent of one another, and of the face that allows concurrent start. If there exist n linearly independent valid tiling hyperplanes for this computation, since the face with concurrent start $\vec{f}$ is one feasible choice, there exist $n-1$ tiling hyperplanes linearly independent of $\vec{f}$. Hence, our algorithm does not fail at this step if the original Pluto algorithm [5] is able to find n linearly independent ones.

Case 2: While finding the the last hyperplane

Let us assume that our scheme fails while trying to find the last hyperplane $\vec{h_n}$. This implies that for any hyperplane strictly inside the cone formed by the face allowing concurrent start and the negatives of the already found hyperplanes, there exists a dependence which makes the tiling hyperplane an invalid one, i.e., there exists a dependence distance d such that

$\vec{h_n} \cdot \vec{d} \leq -1$

From Equation (9), we have

$$k\vec{h_n} \cdot \vec{d} = \lambda'(\vec{f} \cdot \vec{d}) + \lambda_1(-\vec{h_1} \cdot \vec{d}) + \lambda_2(-\vec{h_2} \cdot \vec{d}) + \ldots + \lambda_{n-1}(-\vec{h_{n-1}} \cdot \vec{d}) \quad (10)$$

Consider RHS of Equation (10). As all hyperplanes are valid, $(\vec{h_i} \cdot \vec{d}) \geq 0$ and λ_is are all positive, all terms except for the first one are negative. For any stencil, the face allowing concurrent start always carries all dependences.

$$\vec{f} \cdot \vec{d} \geq 1, \forall \vec{d}$$

If dependences are all constant distance vectors, all $\lambda_i(-\vec{h} \cdot \vec{d})$ terms are independent of program parameters. So, we could always choose λ' large enough and λ_is small so that $\vec{h_n} \cdot \vec{d} \geq 1$ for any dependence $\vec{d}$.

In the general case of affine dependences, our algorithm can fail to find $\vec{h_n}$ in only two cases. The first when it is not statically possible to choose λ_is. This can happen when both of the following conditions are true:

- Dependences are non-uniform
- 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). In order for our algorithm to succeed, one of the λs 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 even allow point-wise concurrent start. Here, $\vec{f}$ does not carry all dependences (Figure 10). □

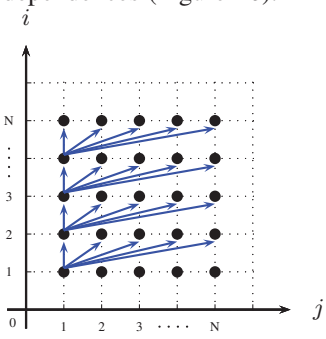


Fig. 9. No tile-wise concurrent start possible

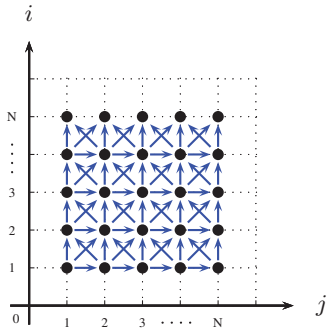


Fig. 10. Even point-wise tile-wise concurrent start is not possible

B. Example

We provide an example showing hyperplanes found by our scheme for a 3-d stencil. For simplicity, we show the memory-inefficient version (with a data dimension for the time iterator) of the 2-d heat stencil in Figure 11.

```
for (t = 0; t < T; t++) {
 for (i = 1; i < N+1; i++) {
  for (j = 1; j < N+1; j++) {
   A[t+1][i][j]
       = 0.125 *(A[t][i+1][j] -2.0*A[t][i][j] +A[t][i-1][j])
       + 0.125 *(A[t][i][j+1] -2.0*A[t][i][j] +A[t][i][j-1])
       + A[t][i][j];
  }
 }
}
```

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

The transformation computed by Algorithm 1 for full concurrent start is:

$$\begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & -1 & -1 \end{bmatrix}$$

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

For partial concurrent start, the transformation computed by Algorithm 1 is:

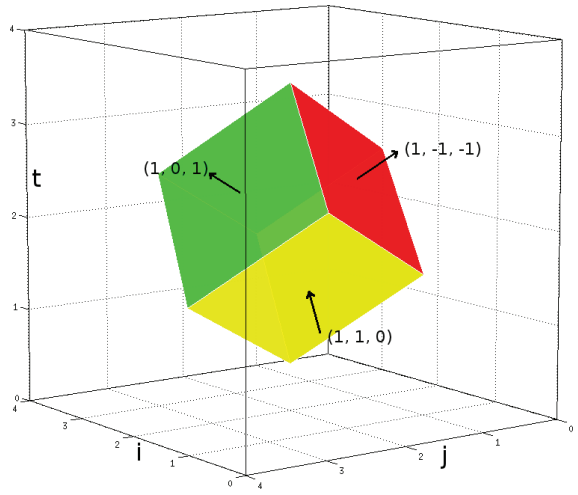


Fig. 12. Tile shape for 2d-heat stencil obtained by our algorithm. The arrows represent hyperplanes.

$$\begin{bmatrix} 1 & 1 & 0 \\ 1 & -1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

This enables concurrent start in only one dimension i.e., not all the tiles along the face allowing concurrent start, but all tiles along one edge of the face can be started concurrently.

V. Experimental Evaluation

We implement our approach on top of the publicly available source-to-source polyhedral tool chain: Pluto [26]. It itself uses the Cloog [9] library for code generation, and PIP [25] to solve for coefficients of hyperplanes. PrimeTile [16] is used to perform unroll-jam on Pluto generated code. We compare the performance of our system with Pluto serving as the state-of-the-art from the compiler works, and the Pochoir stencil compiler [34] representative of state-of-the-art among domain-specific works.

We use two hardware configurations as shown in Table I. All benchmarks use double-precision floating-point computation. icc is used to compile all codes with options "-O3 -fp-model precise"; hence, only value-safe optimizations are performed. The optimal tile sizes and unroll factors are determined empirically with a limited amount of search. The problem sizes for various benchmarks are shown in Table II.

	Intel Xeon E5645	*AMD Opteron 6136*
Microarchitecture	Westmere-EP	Magny-Cours
Clock	2.4 GHz	2.4 GHz
Cores / socket	6	8
Total cores	12	16
L1 cache / Core	32 KB	128 KB
L2 cache / Core	512 KB	512 KB
L3 cache / Socket	12 MB	12 MB
Peak GFLOPs	57.6	153.6
Compiler	icc 12.1.3	icc 12.1.3
Compiler flags	-O3 -fp-model precise	-O3 -fp-model precise
Linux kernel	2.6.32	2.6.35

TABLE I
Details of architectures used for experiments

A. Benchmarks

Benchmark	Problem Size
1d-heat	1600000x1000
2d-heat	16000^2x1000
3d-heat	150^3x100
game-of-life	16000x500
apop	160^3x100
3d7pt	160^3x100

TABLE II
Problem sizes for the benchmarks

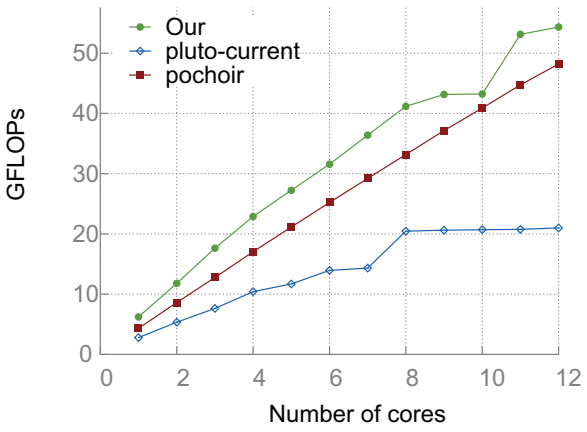
- **Heat 1/2/3D:** Heat equations are examples of symmetric stencils. We evaluate the performance of discretized 1D, 2D and 3D heat equation stencils with non-periodic boundary conditions. Heat-1D is a 3-point stencil, while heat 2D and heat 3d are 5-point and 7-point stencils.
- **Game of Life:** Conway's Game of Life [13] 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 B2S23, where a point is "born" if it has exactly two live neighbors and a point survives the current stage if it has exactly either two or three neighbors.
- **3d7pt:** An order-1 3D 7-point stencil [11] from the Berkeley auto-tuner framework.
- **APOP:** APOP [18] is a one dimensional 3-point stencil that calculates the price of the American put stock option.

B. Significance of concurrent start

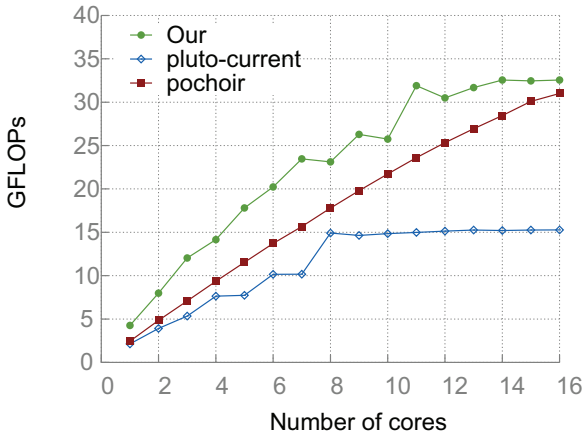
In this subsection, we demonstrate the importance of our enhancement over existing schemes quantitatively. Though pipeline drain and startup can be minimized by tweaking tile sizes (besides making sure there are enough tiles in the wavefront), this could constrain tile sizes leading to loss of locality and/or higher frequency of synchronization. In addition, all of this is highly problem-size dependent. It is true that for some problem sizes one can be fortunate enough to find the right tile sizes such that the effect of pipeline start-up/drain is very low, but the diamond tiling approach does not suffer from this constraint. Figure 13 demonstrates how the performance, load-balance and scale-up provided by diamond tiling remain unaffected for various problem sizes.

In previous scheme for a reasonable problem size, to make sure there are enough tiles in the wavefront and also to saturate the pipeline early, we might be forced to choose a tile-size which is not the best. But, in our scheme, any tile-size can be chosen.

Further, hyperplanes found by our scheme cannot be better than those found by Pluto with respect to the cost function. Therefore, single-thread performance of code generated by our scheme may reduce. However, by using hyperplanes found by our scheme to only demarcate tiles and the hyperplanes that would be found by the Pluto algorithm without our enhancement to scan points inside a tile, we obtain desired benefits for intra-tile execution.



(a) Heat 2D (Intel)



(b) Heat 2D (AMD Opteron)

Fig. 15. Heat 2D

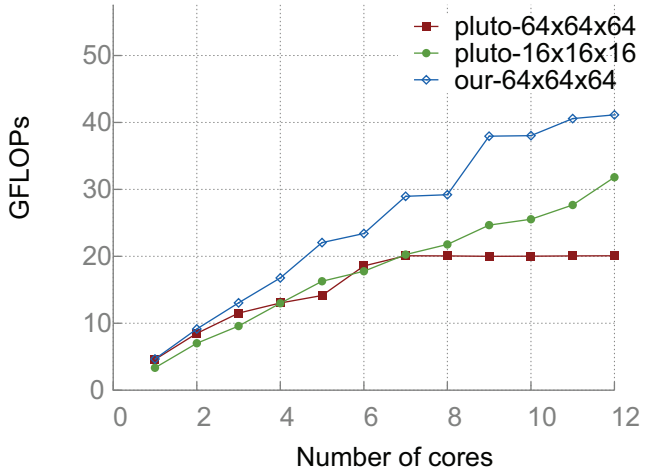


Fig. 13. Heat-2D running on Intel machine for a problem size of 1600*1600*500 the best tile size for locality being 64*64*64

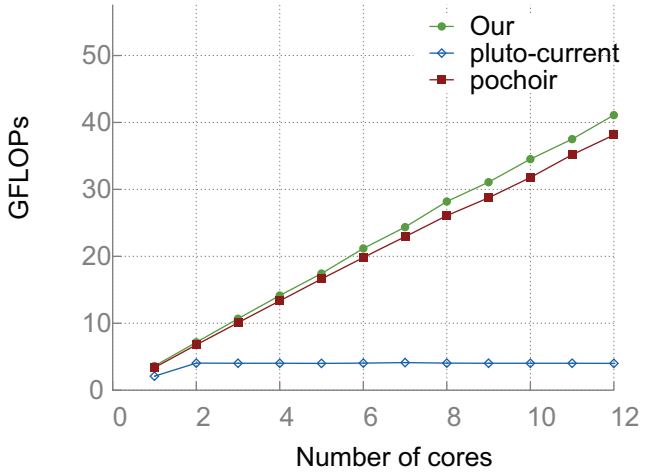


Fig. 14. Heat 1D

C. Results

Our scheme consistently outperforms Pluto for all benchmarks. We perform better than Pochoir on Heat 1D, Heat 2D, Heat 3D, 3d7pt and perform comparably with Pochoir for APOP and Game of Life. The running times for different benchmarks and the speedup factors we get over other schemes are presented in Table III. The speedup factors reported are when running all of them on 12 cores. "icc-par" is icc with "-parallel" flag added to the compiler flags listed in Table I.

Heat 1D shows the effect of concurrent start for tiling enabled by our scheme. Figure 14 shows the load balance we achieve with increasing core count. Pluto generated code performs at around 2 GFLOPs and does not scale beyond two cores. This is a result of both: pipeline startup/drain time, and an insufficient number of tiles in the wavefront necessary to keep all processors busy. Using our new tiling hyperplanes, one is able to distribute iterations corresponding to the entire data space equally among all cores. The same maximal amount of work is done by processors between two synchronizations as the tile schedule is parallel to the face that allows concurrent start. We achieve 72.9% of the theoretical machine peak compared to 66.2% for Pochoir.

For Heat 2D, we perform better than both Pochoir and Pluto (Figure 15(a)). Performance of Pluto generated code saturates after 8 cores for the same reasons as mentioned for 1d-heat. Our scheme performs 12.7% better than Pochoir when using 12 cores on the Intel configuration. We also tested the three schemes on the AMD Opteron setup, and we see similar improvements with our scheme (Figure 15(b)).

On the Heat 3D benchmark, Pluto performs poorly while both Pochoir and our scheme scale well (Figure 17). We use partial concurrent start in one dimension for our code for this example. Pochoir achieves about 21% of the machine peak and our scheme achieves 24.2%. CPU utilization is as expected with our technique while Pluto suffers from load imbalance here. For our scheme, though scaling seen with 1,2,4,8,12 cores is almost ideal, it is flat between 5 and 8 cores. This

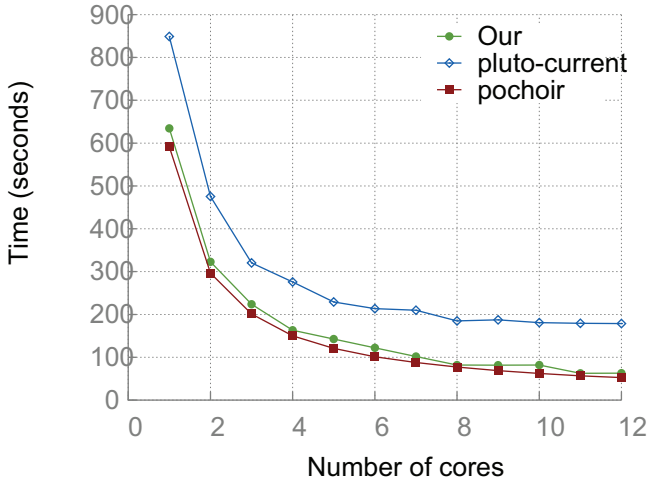


Fig. 16. Game of Life

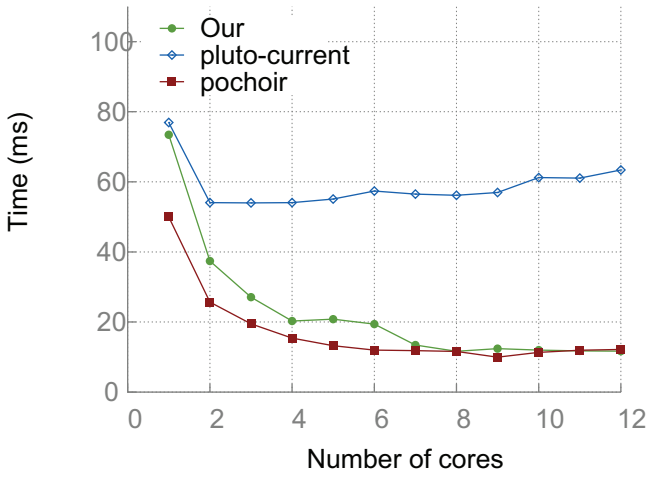


Fig. 18. American Put Option Pricing

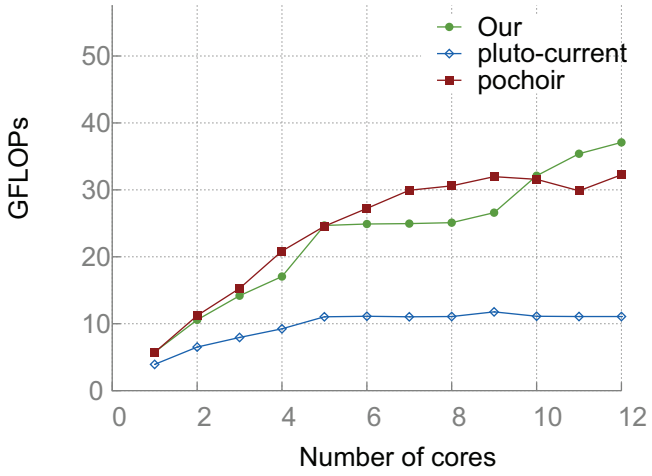


Fig. 17. Heat 3D

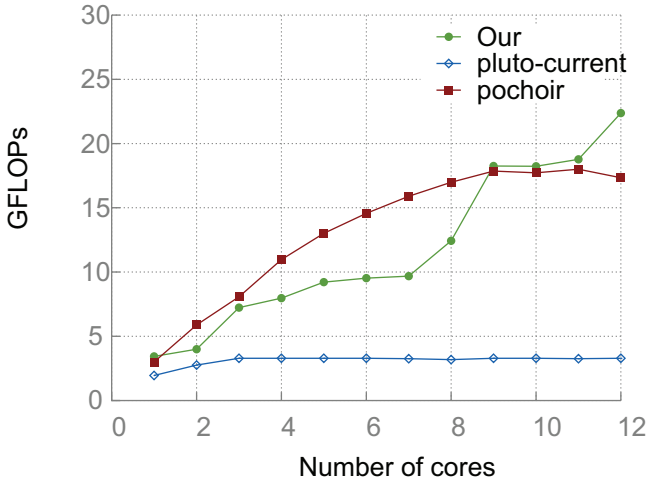


Fig. 19. 3D 7-point Stencil

is due to the number of tiles available not being a multiple of the number of threads. This load imbalance can easily be eliminated using dynamic scheduling techniques [3] and we plan to do this in future. In addition, the tile to thread mapping we employed is not conscious of locality, i.e., inter-tile reuse can be exploited with another suitable mapping. Integrating the complementary techniques presented in [33] may improve performance further.

The 'Game of Life' does not use any floating-point operations in the stencil. Performance is thus reported directly via running time in Figure 16. All schemes see a substantial decrease in running time when moving from single core to two cores. Pluto generated code does not provide much improvement beyond 8 cores.

Both, our scheme and Pochoir exhibit a similar performance trend for APOP. Beyond seven cores, there is no increase in performance owing to the problem size.

For 3d7pt, we again outperform Pochoir and Pluto (Figure 19). The variations in performance with our code are due to the same reasons as those mentioned for 3d-heat.

VI. Related work

A significant amount of work has been done on optimizing stencil computations. These works fall into two categories. One body of works is in the form of developing compiler optimizations [37], [29], [22], [7], and the other on building domain-specific stencil compilers or domain-specific optimization studies [20], [12], [32], [33], [8], [31], [35], [34]. Among the compiler techniques, the polyhedral compiler works have been implemented in some production compilers [15], [30], [6], and are also available publicly as tools [26], [27]. They are in the form of more general transformation frameworks and more or less subsume previous compiler works for stencils. Other significant body of works studies optimization specific to stencils, and a number of these works are very recent. Many of these are publicly available as well such as Pochoir [34]. Such systems have the opportunity to provide much higher performance than compiler-based ones owing to

Benchmark	Performance (1 core)				Performance (12 cores)				Speedup over		
	Original (icc)	pochoir	pluto-cur	Our	icc-par	pochoir	pluto-cur	Our	icc-par	pochoir	pluto-cur
1d-heat	5.73s	1.94s	3.01s	1.77s	2.61s	171.7ms	1.58s	155.8ms	16.75	1.10	10.14
2d-heat	7.65m	4.93m	6.76m	3.42m	8.44m	28.43s	65.31s	25.22s	20.08	1.13	2.59
2d-heat-amd	13.23m	8.71m	9.99m	5.01m	12.30m	50.52s	84.57s	48.11ms	15.34	1.05	1.75
3d-heat	1.61s	849.2ms	1.27s	871ms	1.60s	156.6ms	446ms	135ms	11.85	1.16	3.3
game-of-life	13.50m	9.86m	14.15m	10.60m	13.50m	52.51s	2.98m	64.09s	12.64	0.82	2.79
apop	81.72ms	49.98ms	76.43ms	73.43ms	46.70ms	12.19ms	63.40ms	11.70ms	3.99	1.04	5.42
3d7pt	1.61s	1.05s	1.68s	919.8ms	1.6s	181.9s	995.3ms	143.2ms	11.23	1.27	6.95

TABLE III
SUMMARY OF PERFORMANCE

the greater amount of knowledge they have about computation. Pochoir [34] uses a cache-oblivious tiling mechanism with tiles shaped like trapezoids. Such a tiling mechanism will not suffer from load imbalance. Our scheme on the other hand is a dependence-driven one and is oblivious to the input code being a stencil. Experimental evaluation included a comparison with Pochoir. In addition, though our techniques are implemented in a polyhedral source-to-source compiler, they apply to either body of works.

Strzodka et al. [33] present a technique, cache accurate time skewing (CATS), for tiling stencils, and report significant improvement over Pluto and other simpler manual optimization strategies for 2-d and 3-d domains. Diamond-shaped tiles that are automatically found by our technique are also used by CATS in a particular way. However, their scheme also pays attention to additional orthogonal aspects such as mapping of tiles to threads and is presumably far more cache-friendly than our scheme. We plan to explore those complementary aspects and integrate them into our technique in future. While ours is an end-to-end automatic compiler framework, CATS is more of a customized optimization system.

Efficient vectorization of stencils is challenging. Henretty et al. [17] present a data layout transformation technique to improve vectorization. However, reported improvement was limited for architectures that provide nearly the same performance for unaligned loads as for aligned loads. This is the case with Nehalem, Westmere, and Sandy Bridge architectures from Intel. We did not thus explore layout transformations for vectorization, but instead relied on Intel compiler's auto-vectorization after automatic application of enabling transformations within a tile.

Krishnamoorthy et al [22] addressed concurrent start when tiling stencil computations. However, the approach 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. The key problem with both overlapped tiling and split tiling is the difficulty in performing code generation automatically. No implementation of these techniques currently exists. In addition, our conditions for concurrent start work with arbitrary affine dependences while those in [22] were presented for uniform dependences, i.e., when dependences can be represented by constant distance vectors.

VII. CONCLUSIONS

We have designed and evaluated new techniques for tiling stencil computations. These techniques, in addition to the usual benefits of tiling, provide 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 in some cases by about 4% to 27%. In other cases, we outperform previous compiler techniques by a factor of $2\times$ to $10.14\times$ on 12 cores over a set of benchmarks. Our implementation will be available in a future release of Pluto shortly.

REFERENCES

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[2] C. Ancourt and F. Irigoin. Scanning polyhedra with do loops. In *ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 39–50, 1991.

[3] M. Baskaran, N. Vydyanathan, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan. Compiler-assisted dynamic scheduling for effective parallelization of loop nests on multicore processors. In *ACM SIGPLAN PPoPP*, pages 219–228, 2009.

[4] C. Bastoul. Clan: The Chunky Loop Analyzer. Clan user guide.

[5] U. Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *ETAPS CC*, Apr. 2008.

[6] U. Bondhugula, O. Gunluk, S. Dash, and L. Renganarayanan. A model for fusion and code motion in an automatic parallelizing compiler. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 343–352. ACM, 2010.

[7] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral program optimization system. In *ACM SIGPLAN PLDI*, June 2008.

[8] M. Christen, O. Schenk, and H. Burkhart. Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 676 –687, may 2011.

[9] CLooG: The Chunky Loop Generator. http://www.cloog.org.

[10] A. Cohen, S. Girbal, D. Parello, M. Sigler, O. Temam, and N. Vasilache. Facilitating the search for compositions of program transformations. In *ACM International conference on Supercomputing*, pages 151–160, June 2005.

[11] K. Datta. *Auto-tuning Stencil Codes for Cache-Based Multicore Platforms*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2009.

[12] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. A. Patterson, J. Shalf, and K. A. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *SC*, page 4, 2008.

[13] M. Gardner. *Mathematical Games*. Scientific American, 1970.

[14] M. Griebl, P. Feautrier, and A. Größlinger. Forward communication only placements and their use for parallel program construction. In *LCPC*, pages 16–30, 2005.

[15] T. Grosser, H. Zheng, R. Aloor, A. Simbrger, A. Grolinger, and L.-N. Pouchet. Polly polyhedral optimization in LLVM. In *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT), 2011*, 2011.

[16] A. Hartono, M. Manik, A. Baskaran, C. Bastoul, A. Cohen, S. Krishnamoorthy, B. Norris, and J. Ramanujam. Primetile: A parametric multi-level tiler for imperfect loop nests, 2009.

[17] T. Henretty, K. Stock, L.-N. Pouchet, F. Franchetti, J. Ramanujam, and P. Sadayappan. Data layout transformation for stencil computations on short simd architectures. In *ETAPS International Conference on Compiler Construction (CC'11)*, pages 225–245, Saarbrucken, Germany, Mar. 2011.

[18] J. Hull. *Options, futures, and other derivatives*. Pearson Prentice Hall, Upper Saddle River, NJ [u.a.], 6. ed., pearson internat. ed edition, 2006.

[19] F. Irigoin and R. Triolet. Supernode partitioning. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '88, pages 319–329, 1988.

[20] S. Kamil, K. Datta, S. Williams, L. Oliker, J. Shalf, and K. Yellick. Implicit and explicit optimization for stencil computations. In *ACM SIGPLAN workshop on Memory Systems Perofmance and Correctness*, 2006.

[21] W. Kelly and W. Pugh. A unifying framework for iteration reordering transformations. Technical Report CS-TR-3430, Dept. of Computer Science, University of Maryland, College Park, 1995.

[22] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan. Effective Automatic Parallelization of Stencil Computations. In *ACM SIGPLAN symposium on Programming Languages Design and Implementation*, July 2007.

[23] A. Lim, G. I. Cheong, and M. S. Lam. An affine partitioning algorithm to maximize parallelism and minimize communication. In *ACM ICS*, pages 228–237, 1999.

[24] A. Lim and M. S. Lam. Maximizing parallelism and minimizing synchronization with affine partitions. *Parallel Computing*, 24(3-4):445–475, 1998.

[25] PIP: The Parametric Integer Programming Library. http://www.piplib.org.

[26] PLUTO: A polyhedral automatic parallelizer and locality optimizer for multicores. http://pluto-compiler.sourceforge.net.

[27] POCC: Polyhedral compiler collection. http://pocc.sourceforge.net.

[28] J. Ramanujam and P. Sadayappan. Tiling multidimensional iteration spaces for multicomputers. *Journal of Parallel and Distributed Computing*, 16(2):108–230, 1992.

[29] L. Renganarayanan, M. Harthikote-Matha, R. Dewri, and S. V. Rajopadhye. Towards optimal multi-level tiling for stencil computations. In *IPDPS*, pages 1–10, 2007.

[30] RSTREAM - High Level Compiler, Reservoir Labs. http://www.reservoir.com.

[31] N. Sedaghati, R. Thomas, L.-N. Pouchet, R. Teodorescu, and P. Sadayappan. StVEC: A vector instruction extension for high performance stencil computation. In *20th International Conference on Parallel Architecture and Compilation Techniques (PACT'11)*, Galveston Island, Texas, Oct. 2011.

[32] R. Strzodka, M. Shaheen, D. Pajak, and H.-P. Seidel. Cache oblivious parallelograms in iterative stencil computations. In *ICS*, pages 49–59, 2010.

[33] R. Strzodka, M. Shaheen, D. Pajak, and H.-P. Seidel. Cache accurate time skewing in iterative stencil computations. In *ICPP*, pages 571–581, 2011.

[34] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson. The pochoir stencil compiler. In *SPAA*, pages 117–128, 2011.

[35] J. Treibig, G. Wellein, and G. Hager. Efficient multicore-aware parallelization strategies for iterative stencil computations. *Journal of Computational Science*, 2(2):130 – 137, 2011.

[36] M. J. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[37] D. Wonnacott. Using time skewing to eliminate idle time due to memory bandwidth and network limitations. In *Proceedins of the 14th International Parallel and Distributed Processing Symposium (IPDPS)*, pages 171 –180, 2000.

[38] J. Xue. *Loop tiling for parallelism*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.