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 inherently have the property of 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 transformation tiling frameworks often choose hyperplanes that enforce pipelined start-up which not only leads to load imbalance, but also reduces the number of tiles that can be executed in parallel. 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 of a statement in any program with affine data accesses. We then provide a sound and complete approach to find such hyperplanes for stencil computations. Experimental evaluation on a 12-core Intel Westmere multicore shows that our code is able to outperform a tuned domain-specific stencil code generator by about 4% to 20%. In other cases, we outperform previous compiler techniques by a factor of 2× to 16.4×.

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], [34], [36] is the key transformation used to exploit data locality and parallelism from stencil computations.

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 [17], [2], [26], [3]. 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 [21], [12], [3]. 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 [35], [20] 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. [31] does study this in a more specific context that we intend to here. A comparison of compiler-based and domain-specific 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, and

for \( t = 1; t <= T; t++ \) {
for \( i = 1; i < N+1; i++ \) {
    point1[i] = 0.125*(point0[i+1]-2.0*point0[i]+point0[i-1]);
}
for \( i = 1; i < N+1; i++ \) {
    point0[i] = point1[i];
}
}

Fig. 1. Stencil: 1d heat equation
• 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 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. 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 a single statement of the form as

\[
value_p[t + 1] = f(value_p[t], value_{neighbors_of}(p)[t], \ldots)
\]  

(1)

The \( t \) loop which is the outermost loop (\( t > \) lower bound) forms the face in the iteration space that allows concurrent start.

B. Conic and strict conic combination

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

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

(2)

If \( \lambda_i > 0 \), i.e. if all \( \lambda \)s are strictly positive, we call (2) a strict conic combination of \( \vec{x}_1, \vec{x}_2, \ldots, \vec{x}_n \)

C. Dependences and tiling hyperplanes

Let \( S_1, S_2, \ldots, S_n \) be the statements of a program. The Data Dependence Graph, \( DDG = (S, E) \) 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.

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:

\[
\phi_{S_i} (\vec{x}) = \vec{h} \cdot \vec{x} + h_0
\]

where \( \vec{h} \) consists of scaling and rotation components and \( h_0 \) is the translation or the constant shift component and \( \vec{x} \) is an iteration of \( S_i \).

Prior research [22], [3] provides conditions for a hyperplane to be a valid tiling hyperplane: it should satisfy the following: 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 each edge \( e \) from \( S_i \) to \( S_j \):

\[
\phi_{S_j}(\vec{t}) - \phi_{S_i}(\vec{s}) \geq 0, \quad (\vec{s}, \vec{t}) \in P_e, \forall e \in E \]  

(3)

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 [3].

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 \( e \):

\[
v(\vec{p}) - (\phi_{S_k}(\vec{t}) - \phi_{S_i}(\vec{s})) \geq 0, \quad <\vec{s}, \vec{t}> \in P_e, \forall e \in E
\]  

(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 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 \( (i=0; i<T; i++) \)

for \( (i=2; j<N-3; j++) \)

\[
B[i+1][i] = (B[i][i-2] + B[i+2][i+2])/3.0; \quad // \ S1
\]

Fig. 2. Example program

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 be written as

\[
\phi_{e} \begin{pmatrix}
1 \\
0
\end{pmatrix} \geq \begin{pmatrix}
0 \\
0
\end{pmatrix}
\]  

(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, (4) ends up choosing (1,0) and (2,1)
D. Outer parallelism and concurrent start

Consider a 3-dimensional iteration space with loops \(x, y,\) and \(z\). Let the dependences be \((1,0,0)\) and \((0,1,0)\). Note that there exists an opportunity to start concurrently along the \(z\) axis.

Concurrent start is possible in the above iteration space because dependences span only two dimensions. In other words, there exists one degree of outer parallelism or communication-free parallelism. 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.

\[F' = T_R F\]

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

As mentioned before, \(F'\) 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:

\[
\begin{pmatrix}
 c_1 \\
 c_2
\end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 2 & 1 \end{pmatrix}^{-1} = \begin{pmatrix} 1 & 0 \\ -2 & 1 \end{pmatrix}
\]

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, it is very important to note that every inter-tile dependence is carried by only one of the tiling hyperplanes and this inter-tile dependence has zero component along any other tiling hyperplane.

\[F' = T_R F\]

Tile dependence \(f_k\) carried by any hyperplane \(h_k\) can be found in a different way. It has zero components along all the other hyperplanes. For an \(n\) dimensional space, in the intersection of the orthogonal subspaces of the other \(n - 1\) hyperplanes gives only two unit vectors which are negatives of each other. Among these two unit vectors, the tile dependence \(f_k\) is the one that is carried by the hyperplane \(h_k\), i.e., among the two unit vectors the one which has positive component along the hyperplanes \(h_k\) is the inter-tile dependence carried by \(h_k\).

Consider Figure 7. 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 with \((2,1)\) 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, i.e., there would be a pipelined startup and drain phase. Decreasing the tile size would decrease 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 startup 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 5). 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 face. In this section, we provide conditions for which tiling hyperplanes of any statement allow concurrent start along a given face of its iteration space. Lemma 1 introduces the constraints in terms of inter-tile dependences. Lemma 2 and Lemma 3 map Lemma 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

**LEMMA 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,

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

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.

**LEMMA 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.

\[
k\vec{f} = \lambda_1\vec{h}_1 + \lambda_2\vec{h}_2 + \ldots + \lambda_n\vec{h}_n \quad (6)
\]

**Proof (sufficient):**

Consider the following expression:

\[
\lambda_1(\vec{h}_1 \cdot \vec{c}) + \lambda_2(\vec{h}_2 \cdot \vec{c}) + \ldots + \lambda_n(\vec{h}_n \cdot \vec{c}) \quad (7)
\]

where \( \vec{c} \) is any tile-dependence

As inter-tile dependences are satisfied only by the hyperplane, let \( \vec{h}_1 \) be the only hyperplane which satisfies \( \vec{c} \). Therefore \( \vec{h}_1 \cdot \vec{c} = 0 \). As we constrain all the \( \lambda \)'s to be strictly positive, (7) will always be positive. Thus, by choosing all \( \lambda \) such that

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

we ensure \( \vec{f} \cdot \vec{c} \geq 1 \) for all inter-tile dependences With Lemma 1, this proves sufficiency. \( \square \)

**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., (6) does not hold good. Without loss of generality, we can assume that \( k \in \mathbb{Z}^+ \), but there exist no \( \lambda \)'s which are all strictly positive integers.
Now, consider the inter-tile dependence $c_k$ carried by $\vec{h}_k$. Every tile dependence is carried by only one of the chosen hyperplanes. Therefore, the sum (7) $\leq 0$ because $\vec{h}_k \cdot \vec{c}_k$ is strictly positive, but $\lambda_k \leq 0$ and all other terms $\vec{h}_k \cdot \vec{c}_k$ are all zero. Therefore, $\vec{f} \cdot \vec{c}_k$ will also be zero or negative, which means concurrent start is prohibited along $\vec{f}$. This is a contradiction. □

For the face $\vec{f}$ that allows concurrent start, let $\vec{f}'$ be its counterpart in transformed space.

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

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

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

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

Therefore, Equation 8 becomes

$$\vec{f}' \cdot T^{-1} \geq \vec{I} \tag{9}$$

This can be viewed in a different manner. We know the inter-tile dependences introduced in original iterations space can be approximated as

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

From Lemma 1 we have,

$$\vec{f}' \cdot T^{-1} \geq \vec{I}$$

**B. 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 the same face $\vec{f}$ that allows concurrent start
- Transitive dependences are all carried by this face

**IV. APPROACH FOR FINDING HYPERPLANES ITERATIVELY**

Keeping the stencil-specific properties introduced in II-A, we introduce an iterative scheme which is implemented on top of the Pluto algorithm [3] to find hyperplanes that satisfy properties introduced in the previous section.

**A. Iterative scheme**

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

For a given statement, let $\vec{f}$ be the face along which we would like to start concurrently, $HP$ be the set of hyperplanes already found, and $n$ be the dimension 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 $\vec{f} \cup HP$, as opposed to just $HP$

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.,

$$\vec{h}_n \leq \lambda \vec{f} + \lambda_1 (-\vec{h}_1) + \lambda_2 (-\vec{h}_2) + \cdots + \lambda_{n-1} (-\vec{h}_{n-1}), \tag{9}$$

In Algorithm 1, we only show our additions to the iterative algorithm proposed in [3].

**Algorithm 1 Finding tiling hyperplanes that allow concurrent start**

1. Initialize $HP = \emptyset$
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 $f \cup HP$
5. Add the cost function constraints and minimize for the optimal cost.
6. $HP = h \cup HP$
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 ($HP$)

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 the Equation (6). Trivially, our scheme of finding the hyperplanes is sound, i.e., whenever our scheme gives a set of hyperplanes as output, the output is always correct. □

Now, it remains to be shown that whenever the scheme reports no solution, there does not exist any valid set of hyperplanes.

**Proof: (completeness)**
We prove this by contradiction. Suppose there exists a valid
set of hyperplanes but our scheme failed to find them. The scheme can fail in two places.

**Case 1: While finding the first \( n-1 \) hyperplanes**

Let the algorithm fail while finding \( k^{th} \) hyperplane, \( 1 \leq k \leq n-1 \). It means, there is no valid hyperplane that is linearly independent of the already found \( k-1 \) hyperplanes and face \( \vec{f} \). Therefore, rank of any set of valid hyperplanes can be at the most \( k \). But as \( k \leq n-1 \), it contradicts the fact that there exists a valid set of \( n \) linearly independent hyperplanes.

If our algorithm fails in this step, it means that the highest possible rank of any set of valid tiling hyperplanes is less than the dimensionality of the iteration space of the statement. Hence, there cannot be a solution.

**Case 2: While finding the last hyperplane**

Suppose our scheme fails while trying to find the last hyperplane \( h_n \). It means that for any hyperplane strictly inside the cone formed by the face allowing concurrent start and negatives of the already found hyperplanes there always exists a dependence \( \vec{d} \) such that \( h_n \cdot \vec{d} \leq -1 \).

By Equation (9), we have

\[
kh_n \cdot \vec{d} = \lambda \vec{f} \cdot \vec{d} + \lambda_1(-h_1 \cdot \vec{d}) + \lambda_2(-h_2 \cdot \vec{d}) + \ldots + \lambda_{n-1}(-h_{n-1} \cdot \vec{d})
\]

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

\[
\vec{f} \cdot \vec{d} \geq 1, \forall \vec{d}
\]

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

Our algorithm can fail to find \( h_n \) in only two cases. The first one when it is not statically possible to choose \( \lambda_i \)'s because the dependences are not uniform and \( \vec{f} \cdot \vec{d} \) is independent of the program parameters (Figure 9). In this case, only point-wise concurrent start is possible, but tile-wise concurrent start is not possible. A \( \lambda \) 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).

### B. Example

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.

The transformation computed by Algorithm 1 would be:

\[
\begin{bmatrix}
1 & 1 & 0 \\
1 & 0 & 1 \\
1 & -1 & -1 \\
\end{bmatrix}
\]

**V. EXPERIMENTAL EVALUATION**

We implement our approach on top of the publicly available source-to-source polyhedral tool chain: Pluto [24]. It itself uses the Cloop [7] library for code generation, and PIP [23] to solve for coefficients of hyperplanes. PrimeTile [14] 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 [32] representative of state-of-the-art among domain-specific works.

All benchmarks use double-precision float-point computation. \texttt{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.

### A. Hardware Setup

<table>
<thead>
<tr>
<th></th>
<th>Intel Xeon E5645</th>
<th>AMD Opteron 6136</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microarchitecture</td>
<td>Westmere-EP</td>
<td>Magny-Cours</td>
</tr>
<tr>
<td>Clock</td>
<td>2.4 GHz</td>
<td>2.4 GHz</td>
</tr>
<tr>
<td>Cores / Socket</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>Total cores</td>
<td>12</td>
<td>16</td>
</tr>
<tr>
<td>L1 cache / Core</td>
<td>32 KB</td>
<td>128 KB</td>
</tr>
<tr>
<td>L2 cache / Core</td>
<td>512 KB</td>
<td>512 KB</td>
</tr>
<tr>
<td>L3 cache / Socket</td>
<td>12 MB</td>
<td>12 MB</td>
</tr>
<tr>
<td>Peak GFLOPs</td>
<td>57.6</td>
<td>153.6</td>
</tr>
</tbody>
</table>

**TABLE I**

<table>
<thead>
<tr>
<th>Details of Architectures Used for Experiments</th>
<th>Intel Xeon E5645</th>
<th>AMD Opteron 6136</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compiler Flag</td>
<td>-O3 -fp-model precise</td>
<td>-O3 -fp-model precise</td>
</tr>
<tr>
<td>Linux Kernel</td>
<td>2.6.32</td>
<td>2.6.35</td>
</tr>
</tbody>
</table>
We use two hardware configurations as shown in Table I to evaluate the effectiveness of our framework in exploiting concurrent startup. Concurrent startup is expected to translate into better load balance and scalability of generated code.

- **Intel Xeon E5645**: Intel Xeon E5645 6-core processor in a dual socket configuration. Each core runs at 2.4 GHz with a max turbo frequency of 2.67 GHz. Each processor has 32 KB of L1 data cache, 256 KB of L2 cache per core and a unified L3 cache of 12 MB. The peak performance achievable out of this configuration is 57.6 GFLOPs.

- **AMD Opteron 6136**: AMD Opteron 6136 8-core processor in a dual socket configuration. Each core runs at 2.4 GHz. Each processor has 128KB of L1 Data cache, 512KB of L2 cache per core and a unified L3 cache of 12MB. This configuration provides a peak performance of 153.6 GFLOPs.

### B. 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 [11] 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.


- **APOP**: APOP [16] is a one dimensional 3-point stencil that calculates the price of the American put stock option.

### C. Results

Our scheme consistently outperforms Pluto in all the benchmarks. We perform better than Pochoir on Heat 2D 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 II. The speedup factors are when running all of them on 12 cores. The “icc-par” scheme mentioned in the table is icc with “-parallel” flag added to the compiler flags in Table I.

Heat 1D shows the effect of concurrent start for the tiles enabled by our scheme. Figure 12 shows the load balance we achieve with increasing core count. Pluto generated code performs at around 2 GFLOPs and does not scale with increase in the number of cores beyond 2 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. 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 since the tile schedule is parallel to the face that allows concurrent start. We achieve 54.82% of the theoretical machine peak compared to 66.2% for Pochoir.

For Heat 2D, we perform better than both Pochoir and Pluto (Figure 13). Performance of Pluto generated code saturates after 8 cores for the same reasons mentioned for 1d-heat. Our scheme performs 12.7% better than Pochoir when using 12 cores on the Intel configuration. To characterize scalability of our scheme beyond 12 cores, we tested the three schemes on the AMD Opteron configuration as well. Our scheme performed better than Pochoir here as well (Figure 13(b)).

On the Heat 3D benchmark, both Pluto and our code performs very poorly while Pochoir scales well. Pochoir achieves about 21% of the machine peak for this benchmark. CPU utilization is as expected with our technique while Pluto suffers from load imbalance here. However, our performance here is bad due to poor cache locality, in particular, the lack...
of any particular attention to the tile/thread mapping. With complementary techniques described in [31], we believe this can be overcome, and this is the subject of ongoing work.

The ‘Game of Life’ does not use any floating-point operations in the stencil. Performance is thus reported directly via running time in Figure 14. 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 outperform Pochoir and Pluto (Figure 17). The variations in performance appear to be due to cache locality effects caused by a tile/thread mapping that is not conscious of locality at all. Throughout, all cores were busy though. Again, integrating the orthogonal techniques presented in [31] will likely address this issue.

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 [35], [27], [20], [5], and the other on building domain-specific stencil compilers or domain-specific optimization studies [18], [10], [30], [31], [6], [29], [33], [32]. Among the compiler techniques, the polyhedral compiler works have been implemented in some production compilers [13], [28], [4], and are also available publicly as tools [24], [25]. 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 [32]. Such systems have the opportunity to provide much higher performance than compiler-based ones owing to the greater amount of knowledge they have about computation.
### Summary of Performance

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Original (icc)</th>
<th>pochoir</th>
<th>pluto-cur</th>
<th>Our</th>
<th>icc-par</th>
<th>pochoir</th>
<th>pluto-cur</th>
<th>Our</th>
<th>icc-par</th>
<th>pochoir</th>
<th>pluto-cur</th>
<th>Our</th>
</tr>
</thead>
<tbody>
<tr>
<td>1d-heat</td>
<td>5.73s</td>
<td>1.94s</td>
<td>4.90s</td>
<td>2.41s</td>
<td>2.61s</td>
<td>171.7ms</td>
<td>3.42s</td>
<td>208.9ms</td>
<td>12.52</td>
<td>0.82</td>
<td>16.41</td>
<td></td>
</tr>
<tr>
<td>2d-heat</td>
<td>7.65s</td>
<td>4.93s</td>
<td>6.76m</td>
<td>3.42m</td>
<td>8.44m</td>
<td>26.53s</td>
<td>54.73s</td>
<td>84.57s</td>
<td>17.57</td>
<td>1.20</td>
<td>2.01</td>
<td></td>
</tr>
<tr>
<td>3d-heat</td>
<td>13.23m</td>
<td>8.71m</td>
<td>9.99m</td>
<td>5.01m</td>
<td>12.30m</td>
<td>50.52s</td>
<td>41.99ms</td>
<td>138.9s</td>
<td>5.08</td>
<td>0.19</td>
<td>1.47</td>
<td></td>
</tr>
<tr>
<td>3d7pt</td>
<td>1.61s</td>
<td>849.2ms</td>
<td>1.36s</td>
<td>4.90s</td>
<td>1.60s</td>
<td>156.6ms</td>
<td>1.23s</td>
<td>836.9ms</td>
<td>12.64</td>
<td>0.82</td>
<td>2.79</td>
<td></td>
</tr>
<tr>
<td>1d-heat</td>
<td>1.60s</td>
<td>156.6ms</td>
<td>1.23s</td>
<td>836.9ms</td>
<td>1.20s</td>
<td>1.81s</td>
<td>995.3ms</td>
<td>143.3ms</td>
<td>11.23</td>
<td>1.27</td>
<td>6.95</td>
<td></td>
</tr>
</tbody>
</table>

### Fig. 17. 3D 7-point Stencil

Pochoir [32] 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. [31] 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. Most importantly, ours is a end-to-end automatic compiler framework to generate code for these as opposed to CATS, which is more of a customized optimization technique.

Efficient vectorization of stencils is challenging. Henretty et al. [15] 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 transformation within the tile.

Krishnamoorthy et al [20] 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 [20] 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 startup whenever possible. The 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 $16.4 \times$ on 12 cores over a set of benchmarks. Our implementation will be available in a future release of Pluto shortly.

### References


