Tiling for Dynamic Scheduling

Ravi Teja Mullapudi
Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560012, India
ravi.mullapudi@csa.iisc.ernet.in

Uday Bondhugula
Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560012, India
uday@csa.iisc.ernet.in

ABSTRACT

Tiling is a key transformation used for coarsening the granularity of parallelism and improving locality. It is known that current state-of-the-art compiler approaches for tiling affine loop nests make use of sufficient, i.e., conservative conditions for the validity of tiling. These conservative conditions, which are used for static scheduling, miss tiling schemes for which the tile schedule is not easy to describe statically. However, the partial order of the tiles can be expressed using dependence relations which can be used for dynamic scheduling at runtime. Another set of opportunities are missed due to the classic reason that finding valid tiling hyperplanes is often harder than checking whether a given tiling is valid.

Though the conservative conditions for validity of tiling have worked in practice on a large number of codes, we show that they fail to find the desired tiling in several cases – some of these have dependence patterns similar to real world problems and applications. We then look at ways to improve current techniques to address this issue. To quantify the potential of the improved techniques, we manually tile two dynamic programming algorithms – the Floyd-Warshall algorithm, and Zuker's RNA secondary structure prediction and report their performance on a shared memory multicore. Our 3-d tiled dynamically scheduled implementation of Zuker's algorithm outperforms an optimized multi-core implementation GTfold by a factor of 2.38. Such a 3-d tiling was possible only by reasoning with more precise validity conditions.

Keywords

Tiling, Polyhedral model, Automatic parallelization

1. INTRODUCTION

On modern architectures, memory latency and bandwidth have become a major limiting factor in achieving good performance. The cost of moving data from main memory is orders of magnitude higher than the cost of computation. This disparity between communication and computation costs has prompted a paradigm shift in algorithm design. To achieve good scalability on current and future architectures, algorithms must account for communication costs. Models for cache behavior or the I/O complexity of algorithms have existed for a long time [2, 13, 1, 20]. However, designing algorithms for better locality even with simple memory models is a challenging task. Another facet of architectures today is the increasing parallelism they offer. Exploiting parallelism while maintaining locality makes the task even more daunting. There have been many recent works in designing algorithms which achieve good bounds for IO complexity and parallelism [4, 10, 6].

Tiling [19, 28] is a traditional transformation that has been used to improve locality and coarsen the granularity of computation. State-of-the-art polyhedral compilers like Pluto [25] automatically parallelize and tile affine loop nests while minimizing communication. Although current approaches find effective tiling transformations for a number of affine loop nests, there are cases where desired transformations are missed. The validity of tiling is almost always ensured by only tiling bands of loops on which dependences have non-negative components. Such a condition is easily expressed and linearized for optimization purposes to find valid tiling hyperplanes using machinery from integer linear programming. However, there are tiling strategies which do not satisfy the "non-negative dependence components only" property, but partition the iteration space into tiles which eventually have a valid schedule, i.e., there is no cycle in the dependence graph of tiles. In this paper, we look at such scenarios and see how current techniques could be improved with more precise conditions for the validity of tiling.

Our paper is organized as follows. We first give a brief overview of the state-of-the-art techniques for performing tiling in the polyhedral framework, in particular, the Pluto algorithm. We show its limitations by giving concrete examples where the Pluto algorithm misses good partitions, and then discuss techniques which might help finding the missed tiling. We discuss our ideas for finding better partitions and give directions for further exploration. We experimentally demonstrate the performance benefits one can expect, by manually implementing tiling transformations on two dynamic programming algorithms – Floyd-Warshall All Pairs Shortest Paths and Zuker's optimal RNA folding algorithm. We conclude by pointing out other shortcomings of current approaches which need to be addressed to match performance of manually tuned versions.

2. BACKGROUND

The polyhedral model is a mathematical framework to primarily represent and transform affine loop nests. Affine data accesses are those array accesses where the index expressions can be represented as affine functions of loop iterators and program parameters. An affine loop nest is a sequence of arbitrarily nested loops with affine array accesses and affine loop bounds. A polyhedral repre-

IMPACT 2014
Fourth International Workshop on Polyhedral Compilation Techniques
Jan 20, 2014, Vienna, Austria
In conjunction with HiPEAC 2014.

http://impact.gforge.inria.fr/impact2014

sentation can easily be extracted from such nests. $S = \{s_1, s_2, \ldots, s_n\}$ is the set of statements in an affine loop nest, n being the number of statements. The execution of each statement s can be captured by a polyhedron whose dimension is the number of loops surrounding the statement plus the number of program parameters: we call this the domain of s and denote it by D^s. Each point in D^s is a dynamic instance of statement s, represented by the iteration vector $\vec{i_s}$ containing values of surrounding loop iterators from outermost to innermost.

The generalized dependence graph (GDG) is a multi-graph which captures dependences between statements of an affine loop nest. The vertices of the GDG are statements of an affine loop nest. Edges denote dependence relations between statements. The set of edges in the GDG is denoted by E. Dependence polyhedra are a compact and accurate representation of dependences between dynamic instances of two statements. If e is an edge from statement s to statement t in the GDG, the dependence polyhedron P_e is an integer set that captures when a particular $\vec{i_t}$ depends on $\vec{i_s}$.

A hyperplane is an $n-1$ dimensional affine subspace of an n dimensional space. It partitions the n dimensional space into two half-spaces. A hyperplane for a statement s is of the following form:

$$\phi_s(\vec{i_s}) = \vec{h} \cdot \vec{i_s} + h_0. \quad (1)$$

h_0 is the translation or the constant shift component. The $\vec{h}$ itself can be used to represent a family of hyperplanes to which it is normal. Hyperplanes are used to describe affine schedules or partitions. Polyhedral approaches use hyperplanes to specify tiling directions or tile shapes [19, 3].

3. LIMITATIONS OF CURRENT METHODS

Automatic approaches for tiling iteration spaces have to verify that the tiling transformation does not violate dependences in the original program. These validity constraints are used either while finding tiling hyperplanes or while checking if a given tiling is valid. The validity constraint proposed by Irigoin and Triolet [19] ($HD \geq 0$) only allows for non-negative dependence components along the hyperplane normals. H is a matrix whose rows are normals to faces of the hyperplanes. D is a matrix whose columns are data dependence vectors. Although the condition is conservative, it results in tiling with the desirable properties of being valid independent of the choice of tile sizes and origin.

The validity condition in the book by Xue [29] ($\lfloor HD \rfloor \succcurlyeq \vec{0}$) checks for lexicographic non-negativity of inter-tile dependences. Satisfaction of this condition naturally implies that the tiles can be scanned in the lexicographic order with respect to directions given by the tiling hyperplanes themselves. This is clearly less conservative and thus more powerful since it allows tile-space dependences to have negative components as long as they are lexicographically positive. However, such a condition makes it harder to find or arrive at a tiling that satisfies it. It specifies validity for the chosen tile sizes and tile origin. It is hard to linearize and incorporate this condition into the hyperplane search itself. It can nevertheless be used easily to verify validity given the tiling hyperplanes and tile sizes.

3.1 Pluto algorithm

The Pluto algorithm [7] for automatic parallelization of affine loop nests uses an extension of validity constraints proposed by Irigoin and Triolet [19] while searching for valid tiling hyperplanes. It is also equivalent to the time partitioning constraint used by Lim and Lam [22], and in another context, similar to the forward communication property used by Griebl [15, 14]. We now briefly describe the Pluto algorithm and show the kind of cases where it misses valid tiling opportunities.

The Pluto approach iteratively finds statement-wise scheduling hyperplanes, level by level. A dependence is considered satisfied at a level if it has a positive component along the hyperplane normal at that level. Validity constraints shown in (2) are added for all dependences that have not been satisfied by tilable bands found up to that level. This makes sure that the dependences have non-negative components along all yet to be found hyperplanes. The dependences satisfied by previous levels are only removed when no hyperplane is found satisfying these constraints, i.e., a new tile band needs to be created. For a dependence edge $e \in E$, from statement s to t, the validity constraint is given by:

$$\phi_t(\vec{i_t}) - \phi_s(\vec{i_s}) \geq 0 \, , \langle \vec{i_s}, \vec{i_t} \rangle \in P_e \quad (2)$$

Constraints are also added to ensure that the current set of statement-wise hyperplanes are linearly independent of hyperplanes found for previous levels. Linear independence constraints guarantee that the hyperplanes found result in a one-to-one mapping. At each level, there might be multiple hyperplanes that satisfy the constraints. A cost function that can be encoded is shown in (3).

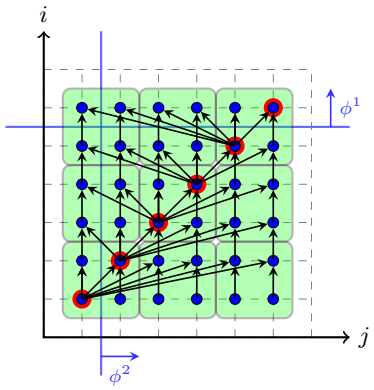
$$\delta_e(\vec{i_s}, \vec{i_t}) = \phi_t(\vec{i_t}) - \phi_s(\vec{i_s}) \, , \langle \vec{i_s}, \vec{i_t} \rangle \in P_e, e : s \rightarrow t \quad (3)$$

δ_e is a factor involved in the reuse distance or communication volume, and minimizing its maximum value across all relevant dependence edges has proved to be often effective. Details on how the cost function is minimized can be found in [7]. Once the statement-wise tiling hyperplanes are found, they are grouped into tilable bands and the bands are tiled. A set of hyperplanes belongs to a tilable band at depth k if all the dependences not satisfied until level k have non-negative components along hyperplanes in the band, i.e., the loops corresponding to the hyperplanes can be permuted without violating any dependences.

For the iteration space shown in Example 1, the Pluto algorithm will find the scheduling hyperplanes $\phi^1_{s_1,s_2} = (1,0)$ and $\phi^2_{s_1,s_2} = (0,1)$. However, it does not lead to the tiling shown in Example 1 since the loops representing hyperplanes ϕ^1 and ϕ^2 are not permutable. So they do not belong to a tilable band. Figure 1 shows an interesting feature of the tiling using hyperplanes ϕ^1 and ϕ^2. The tile size in the ϕ^1 direction cannot be one more than the tile size in ϕ^2 direction, otherwise the tiling becomes invalid. Such restrictions are not captured by current approaches. The dependence graph for the tiles in Example 1 is shown on the right. It is not possible to give a single affine multi-dimensional schedule in loop and tile iterators for the dependence graph.

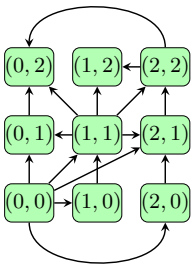
The iteration space for Example 2 is similar to the first one except that there is an anti-dependence. The Pluto algorithm fails to 2-d tile this iteration space as well, but given the structure of the dependences it would seem 2-d tiling is unfeasible. The tiling shown in Example 2 has a valid schedule; it should be noted that the tiles along the diagonal are split into two pieces. The tile dependence graph for the tiling is shown on the right. The graph is acyclic, hence the tiles have a valid schedule.

In both the examples, the 2-d tiling shown will lead to better reuse of array A and task granularity for dynamic scheduling. The tiling shown for the two examples demonstrates two key aspects for improving current approaches. In Example 1 the scheduling hyperplanes computed by Pluto are good, but the condition for using them to tile is too strict. Tiling for Example 2 shows how splitting can help tile spaces with complex dependences. The traditional validity constraints $HD \geq 0$ and $\lfloor HD \rfloor \succcurlyeq \vec{0}$ will report the tiling shown in the examples to be invalid since the tile dependences are

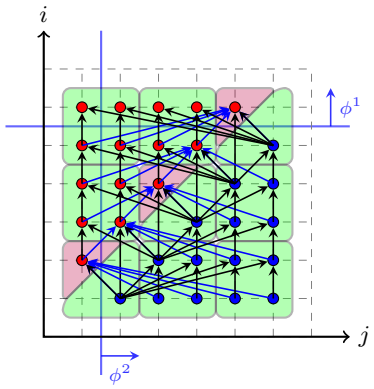



```
for (i = 0; i < N; i++){
   for ( j = 0; j < N; j++) {
      A[j] = A[j] + B[i]; s1
      if (i == j)
         B[i+1] = A[j]; s2
   }
}
```

$\phi^1_{s_1,s_2} = (1,0), \phi^2_{s_1,s_2} = (0,1)$

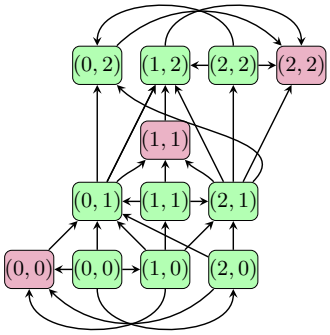


Example 1: Tiling enabled by relaxing validity constraints. The iteration space for the loop nest is shown on the left; Iterations of s_2 are shown as larger points along the diagonal. The tile dependence graph is shown on the right




```
for (i = 0; i < N; i++){
   for ( j = 0; j < N; j++) {
      if (j > i)
         A[j] = A[j] + A[i]; s1
      if (j < i)
         A[j] = A[j] + A[i]; s2
   }
}
```

$\phi^1_{s_1,s_2} = (1,0), \phi^2_{s_1,s_2} = (0,1)$



Example 2: Tiling enabled by selective splitting of tiles. The iteration space for the loop nest is shown on the left. Flow dependences and anti-dependences are shown in different colors, and transitively covered anti-dependences are not shown. The tile dependence graph is shown on the right

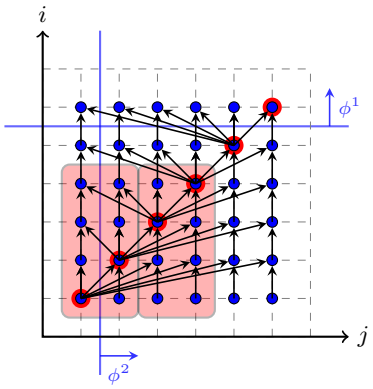


Figure 1: Tiling becomes invalid due to the choice of the tile sizes along each hyperplane, the tiles shown depend on each other

negative even in the lexicographical sense. The main challenge in coming up with such tiling schemes is ensuring that the tile dependence graph is cycle-free. We elaborate on this when we discuss our method for relaxing the validity criteria.

3.2 Other techniques

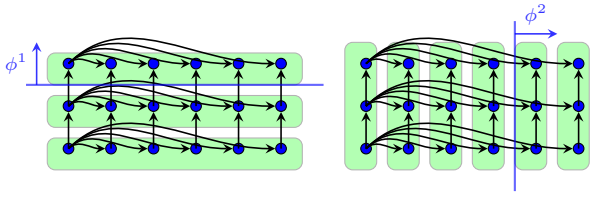
Index Set Splitting [16] (ISS) partitions the iteration space and enables finding different schedules for the partitions. The rationale is that some dependences might be sparse in the iteration space, but constrain the schedule for the entire iteration space. Example 3 shows such a dependence. Partitioning the iteration space to isolate such sparse dependences, and computing different schedules for the partitions might expose more parallelism. Although the objective of ISS is to extract more parallelism and is orthogonal to tiling, the techniques for partitioning can potentially be used to break cycles in the tile dependence graph. To compute the partitions, ISS [16] uses composition and transitive closure of dependence relations, which might not be affine or even decidable in the general case [26]. This leads to approximations which might span the entire iteration space rendering the technique ineffective.

Folding [30] techniques try to make non-uniform dependences shorter or uniform, and have been used to enable time tiling of stencils on periodic domains [24]. The iteration space in Example 1 can be folded along the diagonal, and the folded iteration space can be 2-d tiled. However, it might not always be feasible to find a fold that enables tiling.

Although the techniques we have seen help in enabling tiling in specific cases like stencils, a general approach for both uniform and non-uniform dependences is lacking. In the next section, we discuss our ideas on building a general approach.

4. DIRECTIONS

Coming up with static affine schedules for the tile dependence graphs in both Example 1 and 2 is not very straightforward. There is no set of hyperplanes that can generate a valid schedule for the tiles. Index set splitting techniques can be used to partition the iteration space, and affine schedules can be computed for each partition. However, once a tiling is found and proved to be valid, inter-tile dependence relations can be computed. Such dependence relations can be used to build a task graph for dynamic scheduling at runtime [5]. Dynamic scheduling of tasks has been shown to be more effective than static scheduling for at least certain codes [8, 9, 5]. This is primarily due to better load balance and use of point-to-point synchronization. Also, dynamic scheduling with exact dependence relations exposes more parallelism than static scheduling approaches. This is illustrated in Example 3. Static scheduling results in sequential execution of iterations along ϕ^2, due to the non-uniform dependence. However, dynamic scheduling with exact dependences will allow for parallel execution of iterations along ϕ^2. The downside of using dynamic scheduling is the overhead incurred in using a task scheduling runtime when the task granularity is fine, i.e., when tile sizes are small. A careful choice of tile size is required to balance parallelism, locality, and runtime overhead.



Example 3: Iteration space where a sparse dependence constrains the static schedule

4.1 Tiling validity revisited

Given a set of hyperplanes $\{\phi^1, \ldots, \phi^k\}$ and tile sizes τ_i for ϕ^i, k dimensional tiles can be formed by aggregating hyperplane instances along each ϕ^i separated by τ_i. The subscript for ϕ^k is dropped for convenience when referring to any statement-wise tiling hyperplane at level k.

When ϕ^k_s is used as a tiling hyperplane with tile size τ_k, the resulting tiles can be characterized by their coordinates that we call tile coordinates. If T^k_s is such a tile coordinate associated with ϕ^k_s, then the tile itself is characterized by addition of the following constraints to the statement's domain:

$$\tau_k * T^k_s \leq \phi^k_s(\vec{i_s}) \leq \tau_k * (T^k_s + 1) - 1. \quad (4)$$

DEFINITION 1 (VALID TILING). *A set of hyperplanes ϕ^1, ϕ^2, $\ldots$, ϕ^k with tile size τ_i for ϕ^i, is a valid tiling of an iteration space if the dependence graph of k-dimensional tiles formed by the hyperplanes with their respective tile sizes is cycle-free.*

The above is well-known. The non-existence of a cycle implies that the tiles can be scheduled, if not statically, dynamically with each tile executed atomically. Given dependence relations between the tiles, their transitive closure can be computed to check if there is a cycle in the tile dependence graph. As pointed out earlier, computing the exact transitive closure on parametric relations (at compile time) is undecidable in the general case. If transitive closure can be computed exactly, validity of tiling can be determined precisely. While finding partitions or tiling in an iterative fashion, the following Theorem 1 that we state below can be used. To the best of our knowledge, this result does not appear in prior art.

THEOREM 1. *$\{\phi^1, \ldots, \phi^k\}$ with tile size τ_i for ϕ^i is a valid tiling of an iteration space, if $\{\phi^1, \ldots, \phi^{k-1}\}$ is a valid tiling and ϕ^k is a valid one-dimensional tiling of each $k-1$ dimensional tile formed by $\{\phi^1, \ldots, \phi^{k-1}\}$*

PROOF. Let ϕ^k with size τ_k be a valid tiling of each $k-1$ dimensional tile formed by $\{\phi^1, \ldots, \phi^{k-1}\}$, and $\{\phi^1, \ldots, \phi^{k-1}\}$ be a valid tiling of the entire iteration space. If $\{\phi^1, \ldots, \phi^k\}$ is an invalid tiling of the iteration space, there must be a cycle between the k dimensional tiles. The k dimensional tiles that form a cycle cannot all be from the same $k-1$ dimensional tile. The k dimensional tiles in the cycle should be from at least two distinct $k-1$ dimensional tiles formed by $\{\phi^1, \ldots, \phi^{k-1}\}$; so there exists a cycle between $k-1$ dimensional tiles formed by $\{\phi^1, \ldots, \phi^{k-1}\}$. This leads us to a contradiction as $\{\phi^1, \ldots, \phi^{k-1}\}$ is a valid tiling. Figure 2 illustrates the contradiction. □

A consequence of the above theorem is that it is suitable for use in algorithms that iteratively find tiling hyperplanes. At level k, checking for the validity of k-dimensional tiling within each $k-1$ dimensional tile defined by the previous $k-1$ hyperplanes will suffice. In addition, though the computation of a transitive closure or its approximation is an expensive operation, computing it on the dependence relations localized to partitions of an iteration space can improve accuracy.

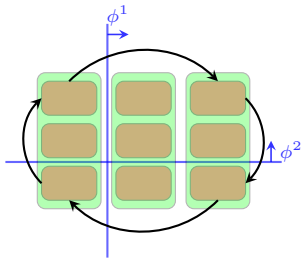


Figure 2: 2-d tiles (formed by ϕ^1, ϕ^2) from different 1-d tiles (formed by ϕ^1) can have a dependence cycle only if the 1-d tiles have a dependence cycle

4.2 Iterative tiling for dynamic scheduling

We first introduce notation to represent tile dependences in a restricted space. The inter-tile dependence polyhedron between k-dimensional tiles formed by $\langle \phi^1, \ldots, \phi^k \rangle$ due to e is denoted by P^k_e. It is computed by projecting out dimensions inner to ϕ^k (for both source and target statements) from the dependence polyhedron for edge e in the transformed space [5]. Each point in the polyhedron P^k_e is of the form $\langle \langle T^1_s, \ldots, T^k_s \rangle, \langle T^1_t, \ldots, T^k_t \rangle \rangle$ where e is from statement s to statement t.

DEFINITION 2 (RESTRICTED TILE DEPENDENCE). *Q^k_e is a subset of P^k_e restricted to the same $k-1$ dimensional tile defined by the $k-1$ tiling hyperplanes outer to ϕ^k. Q^k_e thus captures dependences between only those k-dimensional tiles which are in the same $k-1$ dimensional tile formed by $\langle \phi^1, \ldots, \phi^{k-1} \rangle$. If e is from statement s to statement t, then*

$$Q^k_e = P^k_e \wedge \left(\bigwedge_{1 \leq l \leq k-1} T^l_s = T^l_t \right).$$

Given statement-wise scheduling hyperplanes that Pluto finds, Algorithm 1 provides an iterative approach, which constructs a

valid tiling for dynamic scheduling. The main phases in the algorithm are those of validity checking and tiling correction. For the cycle checking phase, an approximate or exact transitive closure of tile dependence relations can be used. If the absence of cycles at level k cannot be proved, the tile size for that level is halved and tiling at that level is retried. When the tile size drops to one, the dimension is considered not tilable and we do not tile it. Note that both the validity checking step and the tiling correction step can be more accurate and sophisticated. We are currently investigating better approximations for cycle checking and methods to incorporate splitting in the correction step. Splitting can not only be used to break cycles in the tile dependence graph, but also improve the parallelism as shown in split tiling [17].

Algorithm 1: Iterative tiling for dynamic scheduling

Input : Statement-wise scheduling hyperplanes $\phi_s^1, \phi_s^2, \ldots, \phi_s^{maxdim}$ for all $s \in S$; Tile sizes $\tau_1, \tau_2, \ldots, \tau_{maxdim}$; Dependence Polyhedra P_e for all $e \in E$ the set of edges in the GDG.

1 **for** k = *1* **to** $maxdim$ **do**
2 **while** $\tau_k > 1$ **do**
   // **1. Check validity of tiling at level** k**.**
3 **for each** $e \in E$ **do**
4 Compute Q_e^k for dependence $e\colon s \to t$.
   // C is the set of tiles that might be in a cycle
5 C = CycleCheck(Q_e^k for each edge $e \in E$)
6 **if** $C = \emptyset$ **then**
   // Tiling is valid, move to next level
7 **break**
   // **2. Attempt to correct tiling**
8 $\tau_k = \lfloor \tau_k/2 \rfloor$

4.3 A conservative cycle check

Computing transitive closure or even its approximation can be quite expensive, using a simpler conservative method for cycle checking can reduce the computation costs. Also the transitive closure can only be computed when the tile dependence relations are known. So it cannot be used as a validity criteria while searching for tiling hyperplanes. Due to Theorem 1, we need to check for cycles only within each $k-1$ dimensional tile.

Tiles formed by ϕ^k in each $k-1$ dimensional tile can be viewed as points on a one-dimensional line. T_s^k gives the tile coordinates of statement s on this line. Inter-tile dependences can now be thought of as going forward or backward along the line. An inter-tile dependence $\langle T_s^k, T_t^k \rangle$ is called a *backward dependence* if T_t^k depends on T_s^k and $T_t^k < T_s^k$. Similarly, it is called a *forward dependence* if T_t^k depends on T_s^k and $T_t^k > T_s^k$. A cycle can be formed among points on this line only if there is at least one point T_s^k for which one of the following conditions holds true:

1. There is a *backward dependence* $\langle T_s^k, T_t^k \rangle$ and a *forward dependence* $\langle T_{s'}^k, T_{t'}^k \rangle$ such that $T_{t'}^k = T_s^k$ where $s, t, s', t' \in S$. We call this backward violation (BV).

2. There is a *forward dependence* $\langle T_s^k, T_t^k \rangle$ and a *backward dependence* $\langle T_{s'}^k, T_{t'}^k \rangle$ such that $T_{t'}^k = T_s^k$ where $s, t, s', t' \in S$. We call this forward violation (FV).

We denote the set of k dimensional tiles in the same $k-1$ dimensional tile that satisfy conditions BV and FV by B^k and F^k respectively. If the sets B^k and F^k are empty, there is no cycle between the k dimensional tiles. The conditions BV and FV should be checked for every pair of tile dependences within each $k-1$ dimensional tile strip. This translates to computing the sets B^k and

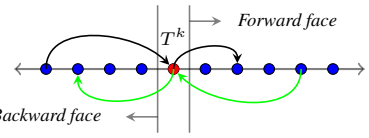


Figure 3: Representing inter-tile dependences at level k as a line graph, the forward dependences are shown above the line and the backward ones below.

F^k considering every pair of restricted tile dependence polyhedra at level k. Given two restricted tile dependence polyhedra Q_e^k and $Q_{e'}^k$ where $e, e' \in E$, $B_{e,e'}^k$ and $F_{e,e'}^k$ are tiles that satisfy BV and FV respectively for inter-tile dependences e, e' within each $k-1$ dimensional tile, then:

$$B_{e,e'}^k = \Big\{ \langle T_s^1, \ldots, T_s^k \rangle \mid \exists T_t^l, \exists T_{s'}^l, \exists T_{t'}^l, (T_s^k \geq T_t^k + 1) \wedge (T_{s'}^k \leq T_{t'}^k - 1) \wedge_{1 \leq l \leq k} (T_{t'}^l = T_s^l) \wedge \langle \langle T_s^1, \ldots, T_s^k \rangle, \langle T_t^1, \ldots, T_t^k \rangle \rangle \in Q_e^k, e : s \to t \langle \langle T_{s'}^1, \ldots, T_{s'}^k \rangle, \langle T_{t'}^1, \ldots, T_{t'}^k \rangle \rangle \in Q_{e'}^k, e' : s' \to t' \Big\}$$

$$F_{e,e'}^k = \Big\{ \langle T_s^1, \ldots, T_s^k \rangle \mid \exists T_t^l, \exists T_{s'}^l, \exists T_{t'}^l, (T_s^k \leq T_t^k - 1) \wedge (T_{s'}^k \geq T_{t'}^k + 1) \wedge_{1 \leq l \leq k} (T_{t'}^l = T_s^l) \wedge \langle \langle T_s^1, \ldots, T_s^k \rangle, \langle T_t^1, \ldots, T_t^k \rangle \rangle \in Q_e^k, e : s \to t \langle \langle T_{s'}^1, \ldots, T_{s'}^k \rangle, \langle T_{t'}^1, \ldots, T_{t'}^k \rangle \rangle \in Q_{e'}^k, e' : s' \to t' \Big\}$$

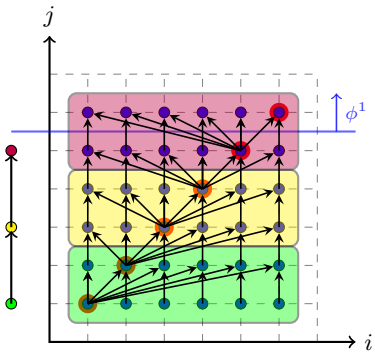
Algorithm 2 uses $B_{e,e'}^k$ and $F_{e,e'}^k$ to compute the set of tiles that possibly participate in cycles. This algorithm can be used as an over-approximation for the cycle checking step in line 7 of Algorithm 1. Algorithm 2 also gives the tiles that satisfy BV or FV – this information can be used by the correction step to split or merge tiles. In Example 2, the output of Algorithm 2 will be the diagonal tiles.

Algorithm 2: Approximate cycle detection

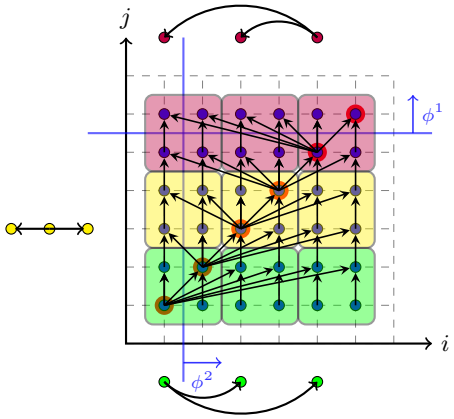
Input : Tile dependence polyhedra at level k Q_e^k for all $e \in E$ the set of edges in the GDG.
Output: Set of tiles that might be part of a cycle
// B^k set of tiles that satisfy B at level k.
// F^k set of tiles that satisfy F at level k.

1 $B^k = \emptyset, F^k = \emptyset$
2 **for each** *pair* $\langle Q_e^k, Q_{e'}^k \rangle$ $e, e' \in E$ // e can be equal to e'
3 **do**
4 Compute $B_{e,e'}^k, F_{e,e'}^k$ using Q_e^k and $Q_{e'}^k$
5 $B^k = B_{e,e'}^k \cup B^k$
6 $F^k = F_{e,e'}^k \cup F^k$
7 **return** $B^k \cup F^k$

Figure 4 shows the working of Algorithm 1 on Example 1. We choose equal tile sizes in both directions. The conservative cycle checking we propose will be able to prove validity of tiling in this case. In the second phase shown in Figure 4b, there is no tile for which either condition BV or FV is satisfied.



(a) ϕ^1 partitions the iteration space into tiles. The tile dependence graph is shown on the left



(b) Tiles formed by ϕ^2 within each tile formed by ϕ^1. The tile dependence graphs for the bottom row of tiles shown below, top row of tiles above and the middle row of tiles to the left

Figure 4: Working of Algorithm 1 on Example 1, given hyperplanes ϕ^1 and ϕ^2 with the same tile sizes for both

4.4 Finding partitioning hyperplanes

Algorithm 1 uses scheduling hyperplanes computed by Pluto to construct a tiling for dynamic scheduling. Although this already improves the space of tiling solutions, using scheduling hyperplanes computed with conservative validity conditions itself limits the solution space. Instead, a valid tiling can be constructed iteratively using an approach similar to the first phase of Pluto. This might result in a better partitioning than the two-phased approach, since the partitioning hyperplanes can have the effect of index set splitting. An approach would be to use a more powerful validity condition for an edge $e \in E$ from statement s to statement t in place of the one used by Pluto. Let R_e^k be the updated dependence polyhedron for edge e obtained by adding tiling constraints given by (4) for $\phi^1, \phi^2, \ldots, \phi^{k-1}$ have been added to P_e. Then, we can use the following tiling validity constraint for ϕ^k:

$$\phi_t^k(\vec{i_t}) - \phi_s^k(\vec{i_s}) \geq 0\,,$$
$$\langle \vec{i_s}, \vec{i_t} \rangle \in \left(R_e^k \wedge \left(\bigwedge_{1 \leq l \leq k-1} T_s^l = T_t^l \right) \right). \qquad (5)$$

The above in effect only considers dependences that do not go across two distinct tiles formed when previously found hyperplanes are tiled. This directly follows from Theorem 1. All hyperplanes from previous levels are known and tile sizes are compile-time constants. Constraints for linear independence with previously found hyperplanes are added, similar to the Pluto approach, and its cost function remains unchanged.

The choice of tile size at an outer level can effect validity at inner levels. In Algorithm 1 and the modified hyperplane search, only the tile size at the current level is altered in an attempt to find a valid tiling. Keeping the tile size choices at outer levels fixed might lead to a weaker tiling transformation. This is a limitation of our approach. A method for backtracking and correcting tile sizes at outer levels is needed for further improvement. However, such backtracking methods can lead to a combinatorial explosion. Good heuristics for backtracking need to be developed to make the method robust to tile size choices. The validity condition in (5) is still the traditional condition used by Pluto. Linearizing the approximate cycle checking constraints and using them in hyperplane search is an interesting avenue for future work.

5. APPLICATIONS

Examples 1 and 2 are artificially constructed iteration spaces which highlight the shortcomings of current tiling approaches. The dependence patterns in the example loop nests are exhibited by real algorithms like Floyd-Warshall's All-Pairs Shortest-Paths [12] and Zuker's [31] optimal RNA secondary structure prediction.

5.1 Floyd-Warshall's All-Pairs Shortest-Paths

Listing 1 shows the loop nest for the All-Pairs Shortest-Paths dynamic programming algorithm. Automatic loop parallelization of this code fails due to the loop carried dependences along inner loops i and j. A full array expansion corresponding to the Floyd-Warshall recurrence is shown in Listing 2. This code exposes more parallelism and can be 3-d tiled using Algorithm 1, but the increase in storage requirement is unacceptable.

```
for(k = 0; k < N; k++) {
   for(i = 0 ; i < N; i++) {
      for(j = 0; j < N; j++) {
        D[i][j] = MIN(D[i][k] + D[k][j], D[i][j]);
      }
   }
}
```

Listing 1: Kernel for Floyd-Warshall's All-Pairs Shortest-Paths

```
for(k = 0; k < N; k++) {
  for(i = 0 ; i < N; i++) {
    for(j = 0; j < N; j++) {
      D[k+1][i][j] = MIN(D[k][i][k] + D[k][k][j], D[k][i
          ][j]);
    }
  }
}
```

Listing 2: Kernel for Floyd-Warshall's All-Pairs Shortest-Paths with full array expansion

Listing 3 shows how to selectively copy the parts of D that are read by the next iteration of k, to reduce storage requirement. Now, the i and j loops can be 2-d tiled and marked parallel – this is the tiling found with traditional validity constraints. The dependence patterns in the 3-d iteration space formed by loops shown in Listing 3 are similar to Example 1. One can visualize this by projecting the 3-d iteration space on the plane $j = 0$ or $i = 0$. Algorithm 1 which uses our improved validity criteria can find 3-d tiling for both

the kernels in Listing 2 and Listing 3. The 3-d tiling of Listing 3 is similar to the 2-d tiling in Example 1 – it allows reuse of D along the k loop as well.

```
for(k = 0; k < N; k++) {
   for(i = 0 ; i < N; i++) {
      for(j = 0; j < N; j++) {
         /*C[0][*] and R[0][*] are initialized
           to D[*][0] and D[0][*] respectively*/
         D[i][j] = MIN(C[k][i] + R[k][j], D[i][j]);
         if (i == k + 1)
            R[k+1][j] = D[i][j];
         if (j == k + 1)
            C[k+1][i] = D[i][j];
      }
   }
}
```

Listing 3: Using additional arrays R and C to remove false dependences in the All-Pairs Shortest-Paths kernel

```
for(k = 0; k < N; k++) {
   for(i = 0 ; i < N; i++) {
      for(j = 0; j < N; j++) {
         if (i < k && j < k)
            D[i][j] = MIN(D[i][k] + D[k][j], D[i][j]);
         if (i > k && j < k)
            D[i][j] = MIN(D[i][k] + D[k][j], D[i][j]);
         if (i < k && j > k)
            D[i][j] = MIN(D[i][k] + D[k][j], D[i][j]);
         if (i > k && j > k)
            D[i][j] = MIN(D[i][k] + D[k][j], D[i][j]);
      }
   }
}
```

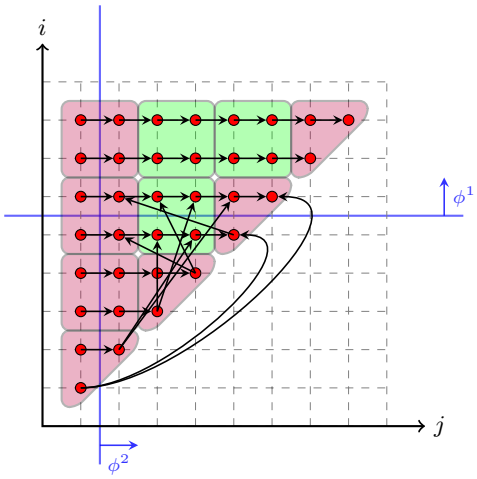
Listing 4: All-Pairs Shortest-Paths kernel after removing spurious writes

Instead of performing the selective copy transformation and using additional arrays, properties of the algorithm can be used to eliminate some of the spurious writes. The array $D[i][j]$ denotes the distance from node i to j. For the code in Listing 1, the writes to $D[i][j]$ when $i = k$ or $j = k$ will not change the value of $D[i][j]$, $MIN(D[i][j] + D[i][i], D[i][j]) = D[i][j]$ and $MIN(D[i][j] + D[j][j], D[i][j]) = D[i][j]$ since the distance of a node to itself cannot be negative. Listing 4 shows the code with the spurious writes removed. This is very similar to the iteration space shown in Example 2. Our current methods will not be able to perform 3-d tiling, but the tiles that need to be split can be identified using Algorithm 2.

5.2 Zuker's optimal RNA secondary structure prediction

Zuker's optimal RNA secondary structure prediction is a complex dynamic programming algorithm. A full description of the recurrence equations and their analysis is beyond the scope of this paper. Lavenier et al. [21] provide an excellent overview of the algorithm and its mapping to a GPU architecture. Our approach is quite similar to theirs. There are two loop nests in the algorithm with $\mathcal{O}(n^3)$ and $\mathcal{O}(k^2n^2)$ complexities where n is the RNA string size and k is a parameter. A low value of k is used in practice making the $\mathcal{O}(n^3)$ loop more interesting for larger sequences. We focus on the $\mathcal{O}(n^3)$ loop nest and 3-d tile it. The $\mathcal{O}(k^2n^2)$ loop nest also has significant reuse and is the subject of future investigation. We illustrate the key ideas of our tiling transformation using a simpler loop nest shown in Example 4.

```
for (i = 0; i < N; i++){
   for ( j = 0; j < i+1; j++) {
      A[i] = A[i] + A[i-j]; s1
   }
}
```



Example 4: Tiling enabled by merging of tiles; complete dependences are shown only for the iterations in the third row of tiles along i to avoid cluttering the diagram

The code shown in Example 4 does a sum(+) reduction for each i and stores it in $A[i]$. The uniform dependences in the ϕ^2 direction form a reduction chain. Operations along each chain can be reordered. The tiling shown in Example 4 is invalid due to cyclic dependences between the left and right tile in each row; the diagram shows this for the third row of tiles. However, due to the reduction along ϕ^2, the left and right tile can be merged into one thereby eliminating cycles. The tiles in the middle are executed in the reduction order followed by the merged tile in each row. This 2-d tiling improves reuse of array A in both i and j directions. Note that Algorithm 2 can identify the left and the right tiles since they have both dependences going out and coming in from the same face. The $\mathcal{O}(n^3)$ loop in Zuker's algorithm has the same dependence patterns as Nussinov's algorithm shown in Listing 5. The innermost loop k computes a reduction for each i, j and stores it in $S[i][j]$. A projection of the iteration space for Nussinov's algorithm on the plane $i = 0$ or $j = 0$ will show dependence patterns similar to Example 4. Similar to Example 4, the 3-d tile with the lowest k and the tile with the highest k have cyclic dependences and they are merged.

```
for (i = N-1; i >= 0; i--) {
   for (j = i+1; j < N; j++) {
      for (k = 0; k < j-i; k++) {
         S[i][j] = MAX(S[i][k+i] + S[k+i+1][j], S[i][j]);
      }
      S[i][j] = MAX(S[i][j], S[i+1][j-1] +
                           can_pair(RNA[i],RNA[j]));
   }
}
```

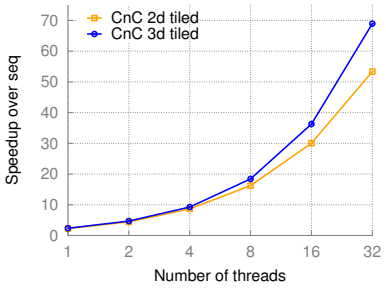
Listing 5: Nussinov's Algorithm

5.3 Performance evaluation

We have used Intel Concurrent Collections (CnC) to implement 3-d tiling similar to that proposed by Venkataraman et al. [27] and

Lavenier et al. [21] for the for All-Pairs Shortest-Paths algorithm and Zuker's algorithm respectively. These implementations quantify the performance benefit one can expect from good tiling transformations. In the CnC programming model, the data required for a task is explicitly specified. This allows for the same program with minor modifications to run seamlessly both on shared and distributed memory. We have only focused on high level tiling transformations in our implementations. An important transformation which has not been discussed in the paper is data tiling. We tiled the data to match the iteration space tiling in our implementations. This is not an additional transformation that we use to improve performance, but the programming model (CnC) requires the user to specify data required for the computation. Specifying data dependences at the granularity of each element would have led to very poor performance. However, the data tiling transformation does improve locality and we have not analyzed its individual impact on performance. Both the 3-d and 2-d tiled implementation of the All-Pairs Shortest-Paths algorithm are part of samples distributed with Intel CnC [11].

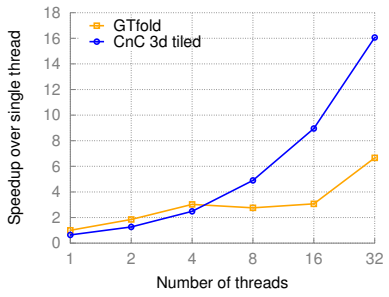
We did a scaling study of the 3-d tiled implementations on a shared memory multi-core architecture. The experimental setup is a four socket machine with an AMD Opteron 6136 (2.4 GHz, 128 KB L1, 512 KB L2, 6 MB L3 cache) in each socket. The memory architecture is NUMA. So, numactl is used to bind threads and pages appropriately for all our experiments. The compiler used for the experiments is Intel C compiler (icc) 13.0.1. All benchmarks are compiled with "-O3 -ansi-alias -ipo" optimization flags.



`floyd` – *seq* time is 231s

Figure 5: Speedup over a naive sequential implementation

The graph `floyd` in Figure 5 shows scaling for both 3-d tiled and 2-d tiled implementations of Floyd-Warshall's All-Pairs Shortest-Paths. The sequential version of `floyd` we used as our baseline is shown in Listing 1. Both the 2-d tiled and 3-d tiled CnC versions scale well but 3-d tiling outperforms 2-d tiling as the number of threads increase. The main factor driving the performance gains is good temporal locality of data in the tiled implementations, i.e., reuse along the k loop of code in Listing 1. Even in the 2-d tiled case where only the i and j loops are tiled, the runtime schedules the tiles that access the same data very close in both space (on the same processor) and time. So, even 2-d tiling shows super-ideal speedup. The 3-d tiling performs even better since the inter-tile reuse in the case of 2-d tiling is converted into intra-tile reuse. Moreover, the number of tasks the runtime system has to handle dramatically reduces when moving from 2-d to 3-d tiling. We used tile sizes 256×256 and $128 \times 128 \times 128$ for the 2-d and 3-d tiled experiments respectively. The tile sizes used give the best performance for both implementations. For a problem size of $N = 4096$, this translates to $4096 \times 16 \times 16$ and $32 \times 32 \times 32$ tasks for the 2-d and 3-d tiled implementations.



`zuker` – *single thread* time is 253s

Figure 6: Speedup over single thread GTfold

GTfold [23] is an optimized multi-core implementation of RNA secondary structure prediction algorithms. We optimized the Zuker's algorithm implementation in GTfold using a 3-d tiling transformation. The graph `zuker` in Figure 6 shows scaling of GTfold and our 3-d tiled CnC implementation of Zuker's algorithm. Our CnC implementation outperforms the GTfold implementation [18] by 2.38×. The GTfold implementation does not perform tiling to improve temporal locality. They optimize memory layout of the arrays to improve spatial locality. Our implementation tiles the innermost k loop in Listing 5 which gives better temporal reuse of S. Note that our implementation performs worse than GTfold for a single thread but scales better as the number of threads increases. Due to the complex 3-d tiling transformation, task kernels become complex. We believe this coupled with runtime overheads for task and data management lead to lower single thread performance.

6. CONCLUSIONS

We have shown that validity constraints for tiling used by current approaches are too conservative, and miss desirable tiling opportunities. We proposed validity constraints which are less restrictive, and gave an iterative approach to construct tilings using improved validity constraints. The iterative method improves the space of tiling solutions, but still does not cover cases which require tile splitting. Use of index set splitting to break cycles in the tile dependence graph can enable tiling in a wider range of scenarios. Even if a complex tiling is found, the approximate cycle checking we propose might not be able to prove its validity. Better approximations for transitive closure or alternate geometric methods may be required to establish validity. We have experimentally shown that tiling transformations driven by more accurate validity conditions are essential for good scaling on some problems. Our 3-d tiled implementation of Zuker's algorithm outperformed an optimized multi-core implementation, GTfold, by a factor of 2.38.

Acknowledgments

We would like to thank the program committee and chairs of IMPACT 2014 for their detailed reviews and comments.

7. REFERENCES

[1] A. Aggarwal, B. Alpern, A. Chandra, and M. Snir. A model for hierarchical memory. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 305–314. ACM, 1987.

[2] A. Aggarwal, J. Vitter, et al. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[3] 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.

[4] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz. Minimizing communication in numerical linear algebra. *SIAM Journal on Matrix Analysis and Applications*, 32(3):866–901, 2011.

[5] 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.

[6] G. E. Blelloch, R. A. Chowdhury, P. B. Gibbons, V. Ramachandran, S. Chen, and M. Kozuch. Provably good multicore cache performance for divide-and-conquer algorithms. In *Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 501–510. Society for Industrial and Applied Mathematics, 2008.

[7] 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 *Compiler Construction*, pages 132–146. Springer, 2008.

[8] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra. Dague: A generic distributed dag engine for high performance computing. *Parallel Computing*, 38, 2012.

[9] E. Chan, E. S. Quintana-Orti, G. Quintana-Orti, and R. Van De Geijn. Supermatrix out-of-order scheduling of matrix operations for smp and multi-core architectures. In *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 116–125. ACM, 2007.

[10] R. A. Chowdhury and V. Ramachandran. Cache-efficient dynamic programming algorithms for multicores. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 207–216. ACM, 2008.

[11] Intel Concurrent Collections for C++ 0.9. http://software.intel.com/en-us/articles/intel-concurrent-collections-for-cc.

[12] R. W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345–, June 1962.

[13] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Foundations of Computer Science, 1999. 40th Annual Symposium on*, pages 285–297. IEEE, 1999.

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

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

[16] M. Griebl, P. Feautrier, and C. Lengauer. Index set splitting. *International Journal of Parallel Programming*, 28(6):607–631, 2000.

[17] T. Grosser, A. Cohen, P. H. Kelly, J. Ramanujam, P. Sadayappan, and S. Verdoolaege. Split tiling for GPUs: automatic parallelization using trapezoidal tiles. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, pages 24–31. ACM, 2013.

[18] GTfold:Scalable Multicore Code for RNA Secondary Structure Prediction. http://gtfold.sourceforge.net/.

[19] F. Irigoin and R. Triolet. Supernode partitioning. In *ACM SIGPLAN Principles of Programming Languages*, pages 319–329, 1988.

[20] H. Jia-Wei and H. Kung. I/O complexity: The red-blue pebble game. In *Proceedings of the thirteenth annual ACM symposium on Theory of computing*, pages 326–333. ACM, 1981.

[21] D. Lavenier, G. Rizk, S. Rajopadhye, et al. GPU accelerated RNA folding algorithm. *GPU Computing Gems*, 2011.

[22] A. Lim and M. S. Lam. Maximizing parallelism and minimizing synchronization with affine transforms. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 201–214, 1997.

[23] A. Mathuriya, D. A. Bader, C. E. Heitsch, and S. C. Harvey. Gtfold: a scalable multicore code for RNA secondary structure prediction. In *Proceedings of the 2009 ACM symposium on Applied Computing*, pages 981–988. ACM, 2009.

[24] N. Osheim, M. Strout, D. Rostron, and S. Rajopadhye. Smashing: Folding space to tile through time. In *Languages and Compilers for Parallel Computing*, pages 80–93. Springer, 2008.

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

[26] W. Pugh and E. Rosser. Iteration space slicing and its application to communication optimization. In *Proceedings of the 11th international conference on Supercomputing*, pages 221–228. ACM, 1997.

[27] G. Venkataraman, S. Sahni, and S. Mukhopadhyaya. A blocked all-pairs shortest-paths algorithm. *Journal of Experimental Algorithmics (JEA)*, 8:2–2, 2003.

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

[29] J. Xue. *Loop tiling for parallelism*. Springer, 2000.

[30] Y. Yaacoby and P. R. Cappello. Converting affine recurrence equations to quasi-uniform recurrence equations. In *VLSI Algorithms and Architectures*, pages 319–328. Springer, 1988.

[31] M. Zuker and P. Stiegler. Optimal computer folding of large RNA sequences using thermodynamics and auxiliary information. *Nucleic acids research*, 9(1):133–148, 1981.