Automatic Tiling of "Mostly-Tileable" Loop Nests

David Wonnacott
Computer Science Dept.
Haverford College
Haverford, Pa, USA 19041
davew@cs.haverford.edu

Tian Jin
Haverford College
Haverford, Pa, USA 19041
tjin@haverford.edu

Allison Lake
Haverford College
(Class of 2014)
allison.lake1@gmail.com

ABSTRACT

Polyhedral compilation techniques have proven to be a powerful tool for optimization of dense array codes. In particular, their ability to tile imperfectly nested loops has provided dramatic speedups by countering limits of memory or network bandwidth. Unfortunately, certain codes, including RNA secondary-structure prediction codes, cannot be tiled effectively using the standard tiling algorithm used with polyhedral techniques.

We have developed a more general variant of polyhedral tiling. It can be applied to loop nests that we consider "mostly tileable". Mostly-tileable loop nests are nests for which classic tiling is prevented by an asymptotically insignificant number of iterations. Our tiling algorithm follows directly from this definition: we peel the problematic iterations of the loop nest and apply classic tiling only to the remaining iterations.

We have applied our algorithm by hand to Nussinov's algorithm for RNA secondary-structure prediction, and more recently developed a ISCC script that derives the transformation from the dependence structure of the code and then performs the transformation. The optimized code is dramatically faster than the un-tiled code, or codes in which only the outer loops are tiled. In future work, we plan to apply our technique to the more challenging codes such as Zuker et. al.'s more recent algorithms for RNA secondary-structure prediction, and seek other important "mostly-tileable" loop nests.

We have not yet explored the performance impact of these tilings on multi-core systems.

1. INTRODUCTION

Polyhedral compilation techniques have proven to be a powerful tool for optimization of dense array codes. In particular, their ability to tile imperfectly nested loops has provided dramatic speedups by countering limits of memory or network bandwidth. However, there are significant challenges to applying tiling to some important loop nests, such as the dynamic programming core of Nussinov's algorithm for prediction of the secondary structure of RNA [7]. While a number of authors have developed approaches to tiling this or similar codes ([1], [3], [10], [11], [2], [12], [8], [4], [9]), we know of no formulation that is suitable for automatically producing the transformation we use from the dependence structure of the program (as would be used in an automatic optimizing compiler). In fact, [3] specifically mentions the challenge of deducing the correct PLS ("piecewise linear schedule") as "an open problem, and beyond the scope of this paper".

1.1 Nussinov's Algorithm, and the Dependence Structure Thereof

RNA secondary structure prediction is an important ongoing problem in bioinformatics. RNA is a single-stranded nucleic acid with four nucleotide base subunits. It serves as an intermediate in protein synthesis from DNA and can catalyze various biological reactions. Hydrogen bonding among its nucleotide bases, namely the classical Watson-Crick-Franklin A-U and C-G base pairs, causes RNA to fold on itself, forming a roughly two-dimensional secondary structure that is crucial to understanding the biological activity of the molecule. These patterns of self-complementary base pairing are often evolutionarily conserved, and thus prediction of secondary structure can be useful when attempting to identify a particular type of RNA, understand its 3-dimensional structure, or search a database for homologous RNA's with conserved secondary structure.

One of the first attempts at predicting RNA secondary structure in a computationally efficient way is the base pair maximization approach, developed by Nussinov in 1978 [7]. Given an RNA sequence $x_1 \ldots x_n$, the $O(n^3)$ Nussinov algorithm finds the secondary structure with the maximum number of base pairs. This problem definition is a drastic simplification of the biophysical factors affecting RNA secondary structure formation but serves as a useful starting point for more sophisticated algorithms. The forward algorithm works by computing the maximum number of base pairs for subsequences $x_i \ldots . x_j$, starting with subsequences of length 1 and building upwards, storing the result of each subsequence in a dynamic programming array. The output of the forward algorithm is the maximum number of base pairs possible for $x_1 \ldots . x_n$, i.e. the entire sequence, and a subsequent trace-back algorithm allows for the output of the list of base pairs in the optimal secondary structure.

IMPACT 2015
Fifth International Workshop on Polyhedral Compilation Techniques
Jan 19, 2015, Amsterdam, The Netherlands
In conjunction with HiPEAC 2015.

http://impact.gforge.inria.fr/impact2015

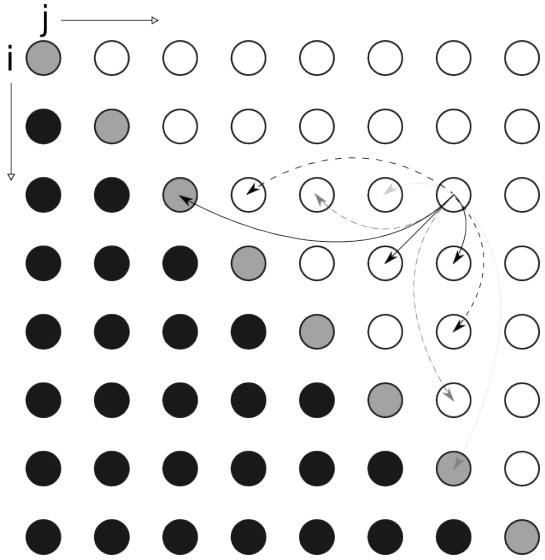

Figure 1: Dependences of One $N(i,j)$ Computation

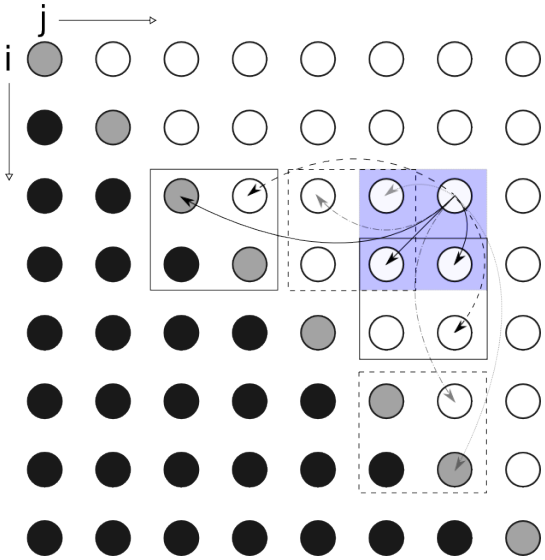

Figure 2: Tiling the Computations of N

Specifically, if $\delta(i,j) = 1$ if(x_i, x_j) match and $i < j-1$, or 0 otherwise, then $N(i,j)$ (the maximum number of base-pair matches of $x_i \ldots x_j$) is defined over the region $1 \leqslant i < j \leqslant n$ as $N(i,j) =$

$$\max(N(i+1, j-1) + \delta(i,j), \max_{i \leqslant k < j}(N(i,k) + N(k+1,j)))$$

and zero elsewhere. The equation leads directly to a triply-nested loop to compute N with the k loop innermost. Each value of $N(i,j)$ is computed in iteration (i,j) of the outer loops, before it is used in later iterations of i and j.

This algorithm is an example of "nonserial polyadic dynamic programming" (in the terminology of [5], as discussed in [12]). For large data sets, the performance of such algorithms is limited by memory bandwidth. A number of authors have explored the challenges of scheduling these codes (see Section 4). Our work lies not in the development of a superior schedule, or in replication of the performance results that result from such schedules, but rather in the automation of the loop tiling that is critical to this scheduling process. The remainder of this section describes the challenges to tiling Nussinov's algorithm and other NPDP codes.

The dependence structure of the recurrence in Nussinov's algorithm allows significant freedom in the scheduling of the iterations of the i and j loops. Figure 1 illustrates the iterations of the (i,j) loop nest, with each white circle representing one iteration (i,j), i.e. all computations needed to produce $N(i,j)$. Grey and black circles represent the "zero elsewhere" elements of $N(i,j)$, arranged in the same pattern as the elements that must actually be computed.

The arrows of Figure 1 illustrate the dependence structure among the iterations of i and j, with dependence "weathervane" arcs pointing into the direction of data flow. With all dependences pointing left and down in the figure, we can schedule the iterations of (i,j) with any wavefront that moves upward and to the right.

Note that the number of values used in the computation of element $N(i,j)$, as well as the length of the dependence arcs, grows with the number of values of k, i.e., as the point (i,j) that we are considering moves away from the diagonal. Thus, for large data sets, the computation of a single value of $N(i,j)$ can touch many memory cells.

Note also the interesting pattern of accesses for the values of k needed for $N(i,j)$. This pattern is illustrated in Figure 1, where two arrows of matching line style (solid, dashed, faint, etc.) point to the two values combined for one given value of k, i.e., one execution of the equation $N(i,k)+N(k+1,j)$. Note the heaviest lines reach the $N(i,j)$ values that are *farthest left* and *closest below*, and the lightest lines reach those that are *closest on the left* and *farthest below*. As we shall see, this pattern creates fundamental challenges for the classical tiling algorithm, which requires that the iteration space be divided into tiles that can be serialized.

1.2 Tiling Nussinov's Algorithm

Thanks to the simple pattern of dependences shown in Figure 1, standard loop tiling techniques can be used to tile the (i,j) loop nest. Tiling the k loop as well is another matter. Consider the 2x2 group of elements of $N(i,j)$ that are highlighted in blue in Figure 2. Tiling only the outer two loops corresponds to computing each of these four $N(i,j)$ values in turn. For small data sets, this may create significant reuse, since e.g. the two topmost iterations will each read the same row of iterations to the left, and similarly the rightmost two will read from the same column of values, producing reuse in both directions. However, for large problem sizes, the necessary rows and columns of values may not remain in cache between different (i,j) iterations, resulting in poor locality. As others have noted, the solution lies in tiling the k loop as well.

By tiling the k loop, we collect all uses for similar values of k (in groups we call k-tiles). In terms of Figure 2, one k-tile corresponds to performing all updates of the blue-shaded

iterations using the values surrounded by solid lines; another k-tile corresponds to updates from the values in the dashed boxes. Unlike the rows and columns to the left of and below the blue box, these k-tiles do not grow with the problem size, and thus locality is preserved for large problem sizes.

Unfortunately, this tiling violates a premise of standard tiling algorithms. Any "atomic" ordering of the resulting tiles will result in a violation of the program's dependences; there is *no* legal serial schedule in which we can execute these tiles. Consider what happens if we attempt to perform all updates to the blue region from the solid-boxed iterations before any from the dashed-box iterations: when we follow the solid dependence arcs to update the upper-right element, we read the iteration directly below it, *which has not yet been updated based on the dashed-box iterations.* Similarly, if we perform all updates from the dashed-box iterations before any solid-box iterations, we read an unfinished value when we follow the faintest dependence arcs.

For larger problem sizes, there will be some k-tiles that are not caught up in this problem. Furthermore, the number of "problematic" iterations of the k loop grows with the tile size, but not with the problem size, whereas the number of non-problematic iterations grows with the problem size. Thus, for a given tiling, and as the problem size grows, the iteration space is dominated by non-problematic iterations. We call this a *mostly-tileable* loop nest: without the asymptotically-small number of problematic iterations, the (i, j, k) nest could be tiled. However, any tiling algorithm that requires the entire set of tiles to be serializable must fail to find this tiling.

2. TILING THE TILEABLE ITERATIONS

At least two approaches have been developed (often independently by separate authors) for getting around the limits of the classical tiling algorithm. One approach (used e.g., by Gautam [3] and by Rizk [8]) essentially removes the "problematic" iterations and applies classical tiling to the remainder. An alternative is to group non-consecutive values of k into tiles, executing the "middle" values of k (i.e., those near $\frac{i+j}{2}$) into the first tile, and then working "outward" to reach $k = i$ and $k = j - 1$ together in the last tile. Prior work has demonstrated the performance advantages of these schedules, but not how to automatically derive them from the dependence structure of the original sequential algorithm. Gautam et al. [3] specifically mention the derivation of this split of the iteration space as a remaining open problem.

Our algorithm for splitting the "problematic" iterations follows immediately from our definition of mostly-tileable loops. The steps of our algorithm are listed in Figure 3. For affine-control loops, the steps of the algorithm can be performed by polyhedral tools such as Verdoolaege's "Integer Set Library", a.k.a. "ISL" [13]. We have not integrated the algorithm into a compiler, but we have used ISL's ISCC text interface to execute it on Nussinov's algorithm. The ISCC scripts are shown in the appendix, with with parenthesized references in Figure 3 serving to relate the details in the appendix (mostly from Figure 7) to the high-level algorithm.

The details of Nussinov's algorithm and its dependences are given in Figure 4 of the appendix; Figure 5 illustrates the

1. We perform instance-wise dependence analysis in the polyhedral model (these are entered by hand, as variables with names starting `DEP`, in our example).

2. We define a standard rectangular tiling transformation, and apply it to the dependences, *without* yet considering its legality (see the `T_TILE` transformation and dependence relations with names starting `DEP6`).

3. We identify all "problematic" iterations — those that read from the tile being updated — by taking the intersection of each dependence relation with a relation whose source and sink iterations share the same values of all but the innermost loop (this intersection produces the iterations in the `PEELED` set, which we put back into the original iteration space as `PEELED_3`); the remaining iterations are not problematic (the `UNPEELED_3` set).

4. The full transformation executes the non-problematic iterations using any legal schedule for these (tileable) iterations, using standard techniques (in our example, we simply applied the original code's $(-i, j)$ schedule to the tiles by applying the `T_CORE` transformation to `UNPEELED_3`), after which the problematic iterations are executed without tiling the innermost loop (applying `T_PEEL` to `PEELED_3`).

5. As with any other transformation, the legality can be tested by testing if any dependence is mapped back in time (we take the intersection between the dependences in the transformed space and the `BACK_IN_TIME6` relation; the script that runs ISL will fail if any of these results is non-empty).

6. If the non-problematic set does not grow more quickly than the problematic set, the transformation is legal but probably not worthwhile. We believe that symbolic volume computation could be used to test for this situation, but have not explored doing so.

Figure 3: Algorithm — Tiling Mostly-Tileable Nests

use of ISCC's "codegen" feature to generate a simple untiled version of Nussinov's Algorithm. Figure 6 illustrates the tiling of only the i and j loops, which requires only standard tiling techniques. Note that, before applying ISCC to this file, the symbolic constant `TS` must be replaced with a known integer (our Makefile uses a sed script to do this automatically).

Figure 7 gives our algorithm for tiling a mostly-tileable loop nest. Note that the important definitions (e.g., of `PEELED`, i.e., the set of problematic iterations, and the transformation itself) require information about the dependences, but are not otherwise specific to Nussinov's algorithm or even to the domain of nonserial polyadic dynamic programming.

As we have noted, the code produced by our algorithm is not novel. However, we believe we are the first to have demonstrated its derivation from the program dependence structure, and are optimistic that other codes may benefit from this algorithm for tiling "mostly-tileable" loop nests.

3. EXPERIMENTAL EVALUATION

We have performed a simple confirmation that the code produced by our algorithm does, in fact, run more quickly than the original code and the code produced by tiling only the i and j loops of Nussinov's algorithm. For extensive exploration of the performance impact of various schedules of the resulting tiles, on a variety of sequential and parallel systems, see the works cited in Section 4.

In our experiments, we executed three versions of the Nussinov algorithms: "untiled", i.e. the original code, "tiled-ij", the code in which only the two outer loops were tiled, and "tiled", the fully-tiled code resulting from our algorithm of Figure 3. We executed each on three different classes of machine, in all cases using only a single core, and executing tiles, and iterations of tiles, in the same fashion as the original schedule (of $(-i, j, k)$). We experimented with randomly generated RNA strands of lengths 500, 2200 and 5000. Problem size 500 was chosen because it reflects the scenario where dramatic speed-up is not to be expected, since the entire upper triangle of the N matrix requires about 1 Megabyte of storage and will thus fit in the outermost cache level on each processor. Problem size 2200 was chosen as it is the average length of RNA strand in human body. Problem size 5000 (roughly the size of the longest human mRNA) was chosen to illustrate any additional advantages for larger instances. Tile size was chosen to be 40, 50 and 60 empirically as suggested by experimenting the tile size on the completely tiled version of Nussinov.

The results are shown in Tables 1, 2, and 3. Each entry in the tiled_ij and tiled columns has three performance results, for tile sizes from 40, 50, and 60. The speed up factor is shown in parenthesis next to each set of performance results. To fit margins, the CPU's are abbreviated I7-0 (for Intel Core i7-860), A10 (for AMD A10-5800K), and I7-3 (for Intel Core i7-3770K).

CPU	Untiled	Tiled-ij	Tiled
I7-0	310	310/310/314 (1x)	63/63/63 (5x)
A10	190	190/190/200 (0.9x)	65/64/64 (3x)
I7-3	220	230/230/230 (0.9x)	47/47/46 (4x)

Table 1: Run Time (in seconds) of Three Versions of Nussinov's Algorithm, on Problem Size 5000

CPU	Untiled	Tiled-ij	Tiled
I7-0	19	19/19/19 (1x)	5.4/5.3/5.3 (4x)
A10	15	15/15/15 (1x)	5.5/5.5/5.5 (3x)
I7-3	14	14/14/14 (1x)	3.9/3.9/3.9 (4x)

Table 2: Run Time (in seconds) of Three Versions of Nussinov's Algorithm, on Problem Size 2200

CPU	Untiled	Tiled-ij	Tiled
I7-0	0.075	0.077/0.080/0.078	0.070/0.074/0.070
A10	0.093	0.098/0.085/0.099	0.069/0.070/0.081
I7-3	0.052	0.054/0.052/0.054	0.056/0.056/0.055

Table 3: Run Time (in seconds) of Three Versions of Nussinov's Algorithm, on Problem Size 500

Problem sizes 2200 and 5000 illustrate cases where the efficiency of cache reuse becomes a dominant factor in the performance of the algorithm. They clearly illustrate the value of tiling all three loops: a slight decrease in performance occurs for tiled_ij, in comparison with untiled version. We can attribute this inefficiency to the fact that in tiled_ij version of Nussinov algorithm, there is no significant improvement on cache reusing because k value of most of the iterations is non-trivial and exceeds the cache capacity which means moving forward to the next element in the same i-j tile will result in cold misses in virtually all the elements with different k values. In contrast, we see a dramatic increase of performance for the fully tiled version because, by limiting the size of the tile to a reasonable value in terms of the cache size, iterating along the k-dimension, the tiled-version program essentially reads from almost the same elements within a tile.

For problem size 500, the data move only among the different levels of cache, not to RAM, and neither approach to tiling produces significant speedup.

We also applied our transformation in a hand-written C code, and in the AlphaZ program transformation system [14]. Table 4 shows the performance of our C and AlphaZ codes, on the Intel Core i7-3770K system, for the problem sizes used above. We explored only one tile size for each problem, using 40 for the larger problem sizes and 50 for $n = 500$, since our AlphaZ implementation requires that the problem size be an integer multiple of tile size. Untiled performance numbers and speedup factors are shown in parenthesis after each value in the table. With the increased performance of the codes in this table comes an increase in the importance of tiling; tiling is even relevant for $n = 500$.

We were initially surprised to see that the AlphaZ system produced code that was significantly faster than the ISL code or our hand-written C code. Further exploration revealed that this was, at least in part, due to the fact that our AlphaZ code transformed not only the *iteration space* of the computation, but also the *storage map* — instead of a two-dimensional array like the one shown in Figure 1, it uses a four-dimensional array (a two-dimensional set of tiles). This alignment of storage with iterations should ensure that cache bandwidth is not wasted on partial cache lines that cross tile boundaries. We tested this hypothesis with a second hand-tuned C code, in which we used a four-dimensional array, and got tiled execution times comparable to AlphaZ (0.026, 2.1, and 20 seconds, respectively). We would also expect the choice of optimal storage mapping to be an important factor in preventing false sharing in multi-core implementations, though we have not (yet) tested this hypothesis.

Access To Experimental Data:. The codes used in the experiments described herein, and transcripts of the command-line sessions that produced the results, are available from the Haverford College Computer Science Technical Report web site, `http://cs.haverford.edu/TechReports`.

4. RELATED WORK

The problem of tiling nonserial polyadic dynamic programming (NPDP) has been approached in a number of ways. Prior work generally focuses on development of a tiled algorithm for NPDP, rather than automatic deduction of such

Version	Tiled Run Time (vs. Untiled) for Codes in AlphaZ and C n=500, TS=50	n=2200, TS=40	n=5000, TS=40
Hand-Tiled C	0.030 (vs. 0.041)	2.1 (vs. 13., ˜6x)	25 (vs. 210, ˜8x)
AlphaZ	0.028 (vs. 0.046)	1.6 (vs. 20., ˜12x)	19 (vs. 320, ˜17x)

Table 4: Run Time (in seconds) of Tiled Codes in C and AlphaZ, on Intel i3770K

an algorithm from the dependence pattern. It frequently focuses on parallel, rather than single-core, performance, and includes a wider variety of example codes. Much of it includes detailed modeling or analysis of factors such as tile size selection. Note that, by identifying the set of iterations for which standard polyhedral tiling can be applied, we hope to leverage the vast body of work that addresses these modeling and tuning issues, and focus primarily on novel issues such as the importance of storage mapping.

Almeida et al. [1] give an algorithm for tiling triangular reductions, including Nussinov's. They include performance models for PRAM and BSP, and give an algorithm for deducing tile size. Their algorithm focuses on two specific traversals of the tiled space (horizontal and vertical).

Gupta et al. [3] explore techniques for scheduling reductions, including for string parenthesization (which has dependences similar to Nussinov's algorithm). They explore the use of piecewise linear schedules (such as the one we produce), but list the automatic selection of such schedules as a remaining open problem outside the scope of their work.

Tan et al. ([10], [11], [12]) give techniques for parallel tiled execution of NPDP codes via a transform of the equations to shift the iteration space. We believe the algorithm produced by our transformation is essentially the same as their algorithm. They include a detailed performance model.

Chowdhury et al. [2] also investigate cache optimization of dynamic programming codes. We believe their algorithm is similar, though perhaps not identical, to those given above. Their presentation is somewhat different, focusing on applying the results of each updated tile across the iterations to the right and above that tile, and this makes a direct comparison somewhat challenging.

Rizk et al. [8] apply the same piecewise-linear schedule used above (and in our work), to produce efficient GPU code for RNA folding. Like earlier work, they do not show how to derive the schedule from the program's dependence structure.

Jacob et al. [4] tile the iteration space of the RNA folding algorithm of Zuker et al., to allow them to realize the algorithm on FPGA chips and produce dramatic speedups. Like Tan et al., they simplify the dependence structure by transforming the recurrence equation. To fit the FPGA model, they apply their technique to the entire dynamic programming matrix, rather than just the triangular part of the n^3 iteration space that contains useful computation, a situation they describe as "suboptimal" (though it still produces an impressively fast result).

Stivala et al. [9] provide a lock-free algorithm for parallel dynamic programming for "multicore processor architectures with shared memory". However, they do not focus on analysis of cache performance.

Mullapudi and Bondhugula [6] have also explored automatic techniques for tiling codes that lie outside the domain of standard tiling techniques. Their approach involves dynamic scheduling of tiles, rather than the generation of a static schedule; it can be applied to Nussinov's algorithm. At this time, we do not have a precise characterization of the relative domains of the two techniques.

5. CONCLUSIONS

Achieving peak 'performance for Nussinov's RNA secondary-structure prediction algorithm requires tiling of all loops, if the full dynamic programming matrix is significantly larger than the cache size. A number of techniques have been developed for tiling this and other instances of nonserial polyadic dynamic programming, but previous work has focused on development of an efficient algorithm for this problem domain, rather than a transformation to deduce the efficient algorithm from the dependence structure of the program.

We have provided a definition of "mostly-tileable" loop nests; an algorithm for tiling such nests follows directly from this definition, and is readily implemented with polyhedral tools, as illustrated by our example implementation with ISCC.

Our ISCC implementation is significantly slower than our hand-written C codes, though the ISCC codes exhibit an overall pattern of speedups that is similar to those of the tiled vs. untiled hand-written C codes. We plan to explore this difference in performance, and also to explore the performance of these codes when running small problems simultaneously on a multi-core system, when bandwidth between the (shared) L3 cache and the cores may become a more significant problem.

In the longer-term, we plan to apply our algorithm to other examples of nonserial polyadic dynamic programming, including the more biologically accurate RNA folding algorithms of Zuker et al.. We also plan to investigate whether our techniques can be applied to important examples outside of the NPDP domain. We plan to continue to investigate storage remapping in combination with iteration space transformation, and hope to explore the possibility of integrating our algorithm more fully into the AlphaZ tiling system.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. CCF-1218827.

We would like to thank Sanjay Rajopadhye for suggesting this problem, and for his advice as we developed our approach.

6. REFERENCES

[1] F. Almeida, R. Andonov, D. Gonzalez, L. M. Moreno, V. Poirriez, and C. Rodriguez. Optimal tiling for the rna base pairing problem. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '02, pages 173–182, New York, NY, USA, 2002. ACM.

[2] 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*, SPAA '08, pages 207–216, New York, NY, USA, 2008. ACM.

[3] G. Gupta, S. Rajopadhye, and P. Quinton. Scheduling reductions on realistic machines. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '02, pages 117–126, New York, NY, USA, 2002. ACM.

[4] A. C. Jacob, J. D. Buhler, and R. D. Chamberlain. Rapid rna folding: Analysis and acceleration of the zuker recurrence. In *Proceedings of the 2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, FCCM '10, pages 87–94, Washington, DC, USA, 2010. IEEE Computer Society.

[5] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Benjamin/Cummings Publishing Company Redwood City, CA, 1994.

[6] R. T. Mullapudi and U. Bondhugula. Tiling for dynamic scheduling. In *IMPACT 2014: 4rd International Workshop on Polyhedral Compilation Techniques*, Jan. 2014.

[7] R. Nussinov, G. Pieczenik, J. R. Griggs, and D. J. Kleitman. Algorithms for loop matchings. *SIAM Journal on Applied mathematics*, 35(1):68–82, 1978.

[8] G. Rizk, D. Lavenier, and S. Rajopadhye. *GPU accelerated RNA folding algorithm*, chapter 14. Morgan Kauffman, 2010. in GPU Computing Gems 4, editor: W-M. Hwu.

[9] A. Stivala, P. J. Stuckey, M. Garcia de la Banda, M. Hermenegildo, and A. Wirth. Lock-free parallel dynamic programming. *J. Parallel Distrib. Comput.*, 70(8):839–848, Aug. 2010.

[10] G. Tan, S. Feng, and N. Sun. Locality and parallelism optimization for dynamic programming algorithm in bioinformatics. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[11] G. Tan, S. Feng, and N. Sun. Cache oblivious algorithms for nonserial polyadic programming. *J. Supercomput.*, 39(2):227–249, Feb. 2007.

[12] G. Tan, N. Sun, and G. R. Gao. Improving performance of dynamic programming via parallelism and locality on multicore architectures. *IEEE Trans. Parallel Distrib. Syst.*, 20(2):261–274, Feb. 2009.

[13] S. Verdoolaege. Isl: An integer set library for the polyhedral model. In *Proceedings of the Third International Congress Conference on Mathematical Software*, ICMS'10, pages 299–302, Berlin, Heidelberg, 2010. Springer-Verlag.

[14] T. Yuki, V. Basupalli, G. Gupta, G. Iooss, D. Kim, T. Pathan, P. Srinivasa, Y. Zoũ, and S. Rajopadhye. Alphaz: A system for analysis, transformation, and code generation in the polyhedral equational model. Technical report, Technical Report CS-12-101, Colorado State University, 2012.

Appendix: ISCC Implementation of Our Algorithm

Figures 4, 5, 6, and 7, give the ISCC programs used in our experiments. These files, and the hand-written C codes, can be downloaded from the Haverford College Computer Science Tech Report web site (`http://cs.haverford.edu/TechReports`).

```
#
# Iteration space and dependence info. for Nussinov's RNA algorithm
#  (maximize the number of matching base pairs)
#
# visualized as i counting down the page, j counting from left to right
#  (we'll need to execute coming back _up_ the page, to preserve dependences)
#

# 'join'= use two sub-folds at k, i.e. N(i, k) and N(k+1, j) to N(i, j)
IS_JOIN := [size] -> { join[i,j,k]    : 0<=i<size && i<j<size && i<=k<j };

# 'add' = add a new base to each end of a group, i.e. i to i+1 and j to j-1
IS_ADD  := [size] -> {  add[i,j,size] : 0<=i<size && i<j<size };


IS_ALL :=  IS_JOIN + IS_ADD;

# some "weathervane" dependences, i.e., pointing from use back to definition
DEP_ADD    := (unwrap (IS_ADD  cross IS_ADD)) *
 ([size] -> { add[i,j,size] -> add[i',j',size] : i'=i+1 && j'=j-1 });

# two "join" dependences, first from i,k, then from k+1,j:
DEP_JOIN_i := (unwrap (IS_JOIN cross IS_ADD)) *
         ([size] -> { join[i,j,k]   -> add[i',j',size] : i'=i   && j'=k   });
DEP_JOIN_j := (unwrap (IS_JOIN cross IS_ADD)) *
 ([size] -> { join[i,j,k]   -> add[i',j',size] : i'=k+1 && j'=j   });

# Some things to check for bad dependences (assuming size>0):
BACK_IN_TIME3 := [size] -> { [a, b, c] -> [a', b', c'] : size > 0 &&
    ((a < a') || (a=a' && b<b') || (a=a' && b=b' && c<c')) };
BACK_IN_TIME5 := [size] -> ...
BACK_IN_TIME6 := [size] -> ...
```

Figure 4: Description of Iteration Space and Dependence Structure using ISCC

```
# Re-generate the untiled Nussinov algorithm, as a check:
# Execute the "i" loop "upwards" on the diagram,
#  from max i (at the bottom) to min i (at the top),
#  i.e., to obey dependeces

T_UP3 := [size] -> {
join[i,j,k]->[size-1-i,j,k];
 add[i,j,k]->[size-1-i,j,k];
};

codegen (T_UP3 * IS_ALL);
```

Figure 5: Re-generation of Untiled Version of Nussinov's Algorithm using ISCC

```
# execute the "i" loop "upwards" on the diagram,
#  from max i (at the bottom) to min i (at the top),
#  i.e., to obey dependeces
T_UP3  := [size] -> {
join[i,j,k]->join[size-1-i,j,k];
 add[i,j,k]->add[size-1-i,j,k];
};

# Tiling, DEFINED IN TERMS OF ORIGINAL i/j
# note this puts in tile boundaries but do not reorder
T_TILE_ij := [size] -> { join[i,j,k] -> join[ib,id,jb,jd,k] :
    0<=id<TS && i=TS*ib+id &&
    0<=jd<TS && j=TS*jb+jd;
    add[i,j,size] -> add[ib,id,jb,jd,size] :
    0<=id<TS && i=TS*ib+id &&
    0<=jd<TS && j=TS*jb+jd;
};

ISCC_dename := [size] -> { join[ib,id,jb,jd,k] -> [ib,id,jb,jd,k];
                    add[ib,id,jb,jd,k] -> [ib,id,jb,jd,k] };

# to check some stuff, for the whole transformation
TRANS := (T_UP3 . T_TILE_ij . ISCC_dename);

# See if any dependence gets mapped back-in-time by our transformation:
#  if any of the following are non-null, our Makefile will stop compiling:
DEP_ADD5 := (TRANS^-1) . DEP_ADD . TRANS;
DEP_ADD5 * BACK_IN_TIME5;

DEP_JOIN_i5 := (TRANS^-1) . DEP_JOIN_i . TRANS;
DEP_JOIN_i5 * BACK_IN_TIME5;

DEP_JOIN_j5 := (TRANS^-1) . DEP_JOIN_j . TRANS;
DEP_JOIN_j5 * BACK_IN_TIME5;

# if we didn't get stopped by non-null relations above, we'll use this code:
codegen( TRANS * IS_ALL );
```

Figure 6: Tiling Only The (i, j) Loops of Nussinov's Algorithm using ISCC

```
# (Note: No T_UP3 here because that's part of our final transformation)

# Tiling, DEFINED IN TERMS OF ORIGINAL i/j/k (with i counting "down the page")
#  once again, we define tile boundaries but do not reorder yet
T_TILE := [size] -> { join[i,j,k] -> join[ib,id,jb,jd,kb,kd] :
  0<=id<TS && i=TS*ib+id &&  0<=jd<TS && j=TS*jb+jd &&  0<=kd<TS && k=TS*kb+kd;
    add[i,j,size] -> add[ib,id,jb,jd,size,0] :
  0<=id<TS && i=TS*ib+id &&  0<=jd<TS && j=TS*jb+jd;
};

# convert dependences into 6-dimensional desired tile space
DEP6_ADD    := T_TILE^-1 . DEP_ADD    . T_TILE;
DEP6_JOIN_i := T_TILE^-1 . DEP_JOIN_i . T_TILE;
DEP6_JOIN_j := T_TILE^-1 . DEP_JOIN_j . T_TILE;
# define the problem with reordering,
#   i.e. iterations that read and write to same (i,j) tile
PEELME := [size] -> {
join[ib,id,jb,jd,kb,kd]->add[ib',id',jb',jd',kb',kd']: ib=ib' && jb=jb';
 add[ib,id,jb,jd,kb,kd]->add[ib',id',jb',jd',kb',kd']: ib=ib' && jb=jb';
};
# Find the "problematic" set to be peeled off and executed later,
#  i.e. those where a dependence matches the "PEELME" conditions
PEELED := (dom (DEP6_JOIN_i * PEELME)) +
          (dom (DEP6_JOIN_j * PEELME)) +
  (dom (DEP6_ADD    * PEELME));


# For easier thinking, put that set back into the original 3-D space:
PEELED_3   := domain ((T_TILE^-1 * PEELED)^-1);
UNPEELED_3 := IS_ALL - PEELED_3;
# Transform the "core" iterations and the "peeled" iterations,
#  putting all "core" first, and tiling i, j, and k there:
T_CORE := ([size] -> { join[ib,id,jb,jd,kb,kd] ->[-ib,jb,1,kb,-id,jd,kd];
                       add[ib,id,jb,jd,size,0]->[-ib,jb,1,size,-id,jd,0]; });
#  and all "peeled" iterations after the core, tiling only i and j there:
T_PEEL := ([size] -> { join[ib,id,jb,jd,kb,kd] ->[-ib,jb,2,-id,jd,kb,kd];
                  add[ib,id,jb,jd,size,0] ->[-ib,jb,2,-id,jd,size,0]; });

# Check for any dependences getting sent back in time:
T_PEEL_THE_RIGHT_ONES := (T_TILE . T_PEEL) * PEELED_3;
T_CORE_THE_RIGHT_ONES := (T_TILE . T_PEEL) * UNPEELED_3;
T_SHAZAM := T_PEEL_THE_RIGHT_ONES + T_CORE_THE_RIGHT_ONES;
DEP_ADD6    := (T_SHAZAM^-1) . DEP_ADD    . T_SHAZAM;
DEP_JOIN_i6 := (T_SHAZAM^-1) . DEP_JOIN_i . T_SHAZAM;
DEP_JOIN_j6 := (T_SHAZAM^-1) . DEP_JOIN_j . T_SHAZAM;
DEP_ADD6 * BACK_IN_TIME6;
DEP_JOIN_i6 * BACK_IN_TIME6;
DEP_JOIN_j6 * BACK_IN_TIME6;

# And generate the code:
codegen( ((T_TILE . T_CORE)*UNPEELED_3)  + ((T_TILE . T_PEEL)*PEELED_3) );
```

Figure 7: Tiling All Loops of Nussinov's Algorithm, After Peeling Problematic Ones, using ISCC