Palkowski and Bielecki *BMC Bioinformatics* (2019) 20:208
https://doi.org/10.1186/s12859-019-2785-6

BMC Bioinformatics

RESEARCH ARTICLE Open Access

# Tiling Nussinov's RNA folding loop nest with a space-time approach

Marek Palkowski* and Wlodzimierz Bielecki

## Abstract

**Background:** An RNA primary structure, or sequence, is a single strand considered as a chain of nucleotides from the alphabet AUGC (adenine, uracil, guanine, cytosine). The strand can be folded onto itself, i.e., one segment of an RNA sequence might be paired with another segment of the same RNA sequence into a two-dimensional structure composed by a list of complementary base pairs, which are close together with the minimum energy. That list is called RNA's secondary structure and is predicted by an RNA folding algorithm. RNA secondary structure prediction is a computing-intensive task that lies at the core of search applications in bioinformatics.

**Results:** We suggest a space-time tiling approach and apply it to generate parallel cache effective tiled code for RNA folding using Nussinov's algorithm.

**Conclusions:** Parallel tiled code generated with a suggested space-time loop tiling approach outperforms known related codes generated automatically by means of optimizing compilers and codes produced manually. The presented approach enables us to tile all the three loops of Nussinov's recurrence that is not possible with commonly known tiling techniques. Generated parallel tiled code is scalable regarding to the number of parallel threads – increasing the number of threads reduces code execution time. Defining speed up as the ratio of the time taken to run the original serial program on one thread to the time taken to run the tiled program on P threads, we achieve super-linear speed up (a value of speed up is greater than the number of threads used) for parallel tiled code against the original serial code up to 32 threads and super-linear speed up scalability (increasing speed up with increasing the thread number) up to 8 threads. For one thread used, speed up is about 4.2 achieved on an Intel Xeon machine used for carrying out experiments.

**Keywords:** RNA folding, Loop tiling, Space-time tiling, Nussinov's algorithm, Parallel computing

## Background

Ribonucleic acid (RNA) molecule is one of the most important molecules in the biological systems. RNA is typically produced as a single stranded molecule, which then folds intramolecularly to form a number of short base-paired stems. This base-paired structure is called the secondary structure of the RNA. The dynamic programming approach to RNA secondary structure prediction relies on the fact that structures can be recursively decomposed into smaller components. In each of the decomposition steps, only a single loop (or stacking of two consecutive base pairs) needs to be evaluated.

Nussinov proposed a dynamic programming algorithm for RNA folding in 1978 [1], which maximizes the number of non-crossing matchings between complimentary bases of an RNA sequence of length $N$.

Let $X = x_1, x_2, \ldots, x_N$ be an RNA sequence, where $x_i \in \{G(guanine), A(adenine), U(uracil), C(cytosine)\}$ is a nucleotide. Nussinov's dynamic programming recurrence for $N \times N$ matrix $S$ is given below.

$$S(i,j) = \max_{1 \le i < j \le N} \begin{cases} S(i+1, j-1) + \sigma(i,j) \\ \max_{i \le k < j} (S(i,k) + S(k+1,j)), \end{cases}$$

Here, $S(i,j)$ defines the maximum number of base-pair matches of $x_i, \ldots, x_j$ over the region $1 \le i < j \le N$ and $\sigma(i,j)$ is a function, which returns 1 if $(x_i, x_j)$ is an $AU$, $GC$, or $GU$ pair and 0 otherwise.

*Correspondence: mpalkowski@wi.zut.edu.pl
West Pomeranian University of Technology, Faculty of Computer Science, Zolnierska 49, 71-210 Szczecin, Poland

Listing 1 represents the triply nested affine loops with two statements accessing the two-dimensional array $S$ implementing Nussinov's algorithm.

**Listing 1 Nussinov's loop nest**

```
1  for (i = N-1; i >= 0; i--) {
2    for (j = i+1; j < N; j++) {
3      for (k = 0; k < j-i; k++) {
4       S[i][j] = max(S[i][k+i] + ↙
            ↳ S[k+i+1][j], S[i][j]); // S1
5      }
6     S[i][j] = max(S[i][j], S[i+1][j-1] + ↙
          ↳ sigma(i,j));    // S2
7    }
8  }
```

Fast Nussinov implementations for shared memory architectures must ensure both aspects of code parallelism and cache optimization. Cache optimization is not found or limited to the memory layout optimization to improve spatial locality in popular parallel implementations of RNA folding, for example, GTFold [2], UNAfold [3] or RNAfold [4], which, however, implement energy minimization.

There are several manual or empirical approaches in literature improving data locality of serial or multi-threaded RNA folding code, e.g. [5–10], dedicated to various hardware platforms including GPUs and FPGAs.

Li et al. [5] suggested a cache efficient version of Nussinov's recurrence by using the lower triangle of matrix $S$ to store the transpose of the computed values in the upper triangle of $S$. As new $S_{i,j}$s are computed, they are stored in both $S_{i,j}$ and $S_{j,i}$ for $j \leq i$. The sum $S_{i,k} + S_{k+1,j}$ is computed as $S_{i,k} + S_{j,k+1}$. Hence, Li's modifications accelerate rapidly code execution because reading values in a row is more cache efficient than reading values in a column [5].

Zhao and Sahni developed three cache-efficient algorithms without increasing the memory requirement, *ByRow*, *ByRowSegment*, and *ByBox* for Nussinov's RNA folding [6]. They showed that presented techniques based on a simple LRU cache model give better run time and energy performance than Li's approach. Unfortunately, no parallel code is presented by the authors.

The effectiveness of automatic tiling and parallelization of loop nests depends a lot of their dependence patterns. The dependences of a loop nest can be classified into two categories: uniform and non-uniform. The dependences are uniform only when the distances between dependent loop nest statement instances in the iteration space are uniform, i.e., these distances are expressed by constants; otherwise they are non-uniform. A set of distance vectors represents distances between dependent loop nest statement instances calculated as the difference between the iteration vectors representing the destinations and sources of dependences.

For uniform dependences, the corresponding dependence graph is regular while for non-uniform dependences it is irregular. Automatic tiling and parallelization of loop nests with non-uniform dependences by means of affine transformations is much difficult than those exposing uniform dependences. The reason is that for such dependences, in general, constraints formed to extract affine transformations (to be next applied to tile and parallelize loops) are parametric and non-linear, this considerably increases the computational complexity of extracting affine transformations. Even when affine transformations can be found, they do not guarantee efficient loop tiling and parallelization: only some loops from all ones in the loop nest can be tiled and/or parallelized.

The Nussinov kernel involves mathematical operations over affine control loops whose iteration space can be represented by the polyhedral model [11]. However, the Nussinov RNA folding acceleration is still a challenging task for modern compilers because that code is within nonserial polyadic dynamic programming (NPDP), which is a particular family of dynamic programming with non-uniform data dependences, and it, as mentioned above, is more difficult to be optimized [7].

Optimizing compilers usually apply loop tiling to generate cache-efficient code on multicore architectures that maximizes data reuse in deep memory hierarchies and reduces synchronization cost [12]. Loop tiling transformations allow for improving data locality and generate coarse-grained parallel code that leads to improving code performance [13]. The most popular techniques of tiling are based on the affine-transformation framework (ATF), which is implemented in several tools [12].

Pluto [14] is the most popular state-of-the-art source-to-source polyhedral code generator that transforms C programs to parallel coarse-grained code with enhanced data locality. Pluto uses a scheduling algorithm, which tries to find affine transformations allowing the most efficient tiling. The main purpose is to minimize the amount of inter-tiles communications and ensure parallelism among tiles. The Pluto schedule is optimal, it reduces the number of dependences crossing tile boundaries. Unfortunately, Pluto fails to generate a band of loops where all multiple consecutive loops may be interchanged while respecting the legality in the case of rectangular tiling for NPDP kernels. Pluto serializes the innermost loop of Nussinov's RNA folding, which is a key of cache locality optimization [15]. As a consequence, Pluto fails to generate 3-D tiles that prevents achieving maximal code locality and performance.

Mullapudi and Bondhugula introduced a dynamic tiling technique for Zuker's RNA secondary structure prediction [11]. Their technique overcomes some limitations of the affine transformation framework. Generated tiles are of the 3-D dimension and they can be scheduled only at run-time, i.e.,

that technique does not allow for generation of any static code.

Wonnacott et al. suggested 3-D tiling of "mostly-tileable" loop nests representing Nussinov's RNA secondary structure prediction [16]. This approach extracts non-problematic statement instances in the loop nest iteration space, i.e., those that can be safely tiled by means of well-known techniques. The reminding statement instances should be run serially to preserve all the dependences available in the loop nest. Unfortunately, the approach allows for generation of only serial code.

The tiling technique presented in paper [15] transforms (corrects) original rectangular tiles into target ones, which are valid under lexicographic order. Tile correction is performed by means of the transitive closure of loop dependence graphs. Loop skewing is used to parallelize code. We achieved higher speed-up of generated tiled code in comparison with that produced with state-of the-art source-to-source optimizing compilers. However, the correction technique generates irregular tiles, some of them can be too large, this does not allow us to achieve maximal code locality and performance [17].

In this paper, we present a novel approach of space-time tiling for accelerating Nussinov's RNA folding, which allows for generation of tiled code with the following features:

- the dimension of generated tiles is 3-D,
- generated code can be easily parallelized by means of the skewing technique,
- target parallel code is scalable regarding to the number of threads and the length of an RNA sequence,
- generated parallel code is regular and compact, it outperforms known automatically and manually generated related codes.

The concept of space-time tiling is the following. Scrutinizing the Nussinov loop nest, we discover that dependences along both axis $i$ and $j$ spread in only the forward direction, i.e., the two corresponding elements of all dependence distance vectors are non-negative (taking into account that the value of index $i$ is decremented). Using this fact, we split the iteration spaces of the loop nest statements into two groups of sub-spaces of a fixed width, which intersect axes $i$ and $j$ and are in parallel with planes $(j,k)$ and $(i,k)$, respectively. Figure 1 presents such sub-spaces for statement S1. Blue lines depict planes situated in parallel with plane $(j,k)$ while red ones present planes located in parallel with plane $(i,k)$. Each sub-space of the first and second groups is represented with identifiers $id_1$ and $id_2$, respectively; integers in brackets are the identifiers of sub-spaces; name $SPACEi(j)$ states for

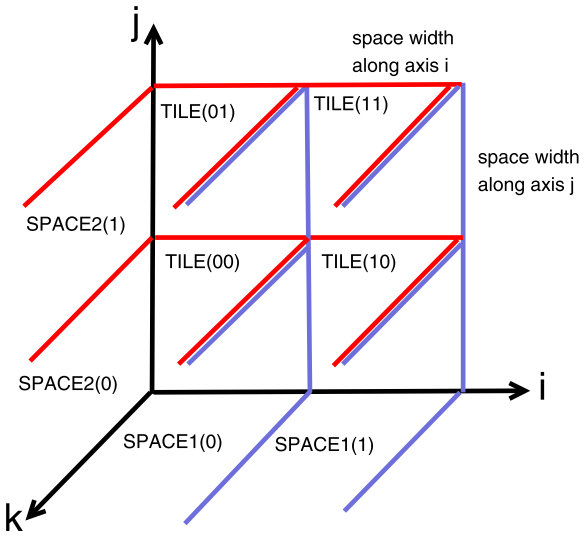


**Fig. 1** Spaces and tiles

the sub-space belonging to group $i, i = 1,2$ and its identifier is $j = 0,1$.

Then we form tiles as the intersection of sets representing the sub-spaces mentioned above, see Fig. 1. Each such a tile is represented with an identifier $(id_1, id_2)$, where $id_1$ and $id_2$ are the identifiers of the corresponding sub-spaces, in Fig. 1, they are shown in brackets. Such tiles are valid under lexicographical order because inter-tile dependences are spread in only the forward directions regarding to both $id_1$ and $id_2$, i.e, the both corresponding elements of all dependence distance vectors are non-negative.

The size of generated tiles is limited along axes $i$ and $j$ with the width of sub-spaces, but it is not limited along axis $k$ that reduces code locality – see Fig. 1. To split each tile generated as presented above into sub-tiles along axis $k$, we can find any valid schedule of statement instances allowing for forming time partitions, which are to be enumerated serially, while statement instances of each time partition can be run in parallel. Let us consider Fig. 2. It represents the iteration space of a single tile provided that the sub-space width is equal to 3. Suppose that dependences within that tile are characterized with the following set of distance vectors.

$\{(0,0,1),(0,1,0),(1,0,-1)\}$.

Distance vectors allow us to generate all dependences in the iteration space shown in Fig. 2. To find the destination of a dependence with a given dependence source within the iteration space, we add a distance vector to this source and if the resulting iteration is within the iteration space, we conclude that there is the dependence in that iteration space.

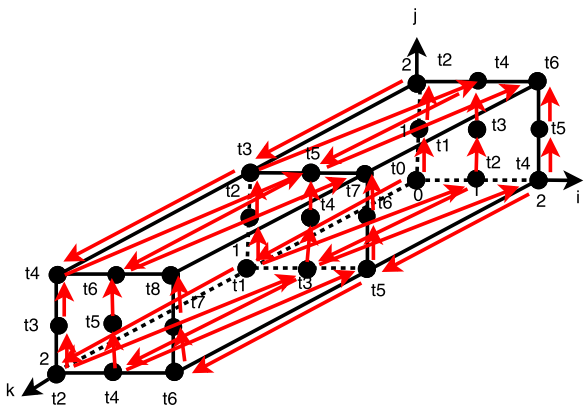


**Fig. 2** Time slices

In Fig. 2, we show with red arrows some dependences corresponding to the distance vectors above. Valid time partitions, *ti*, can be represented with the following sets, each including iterations, which can be run in parallel.

$t0 := \{ (0,0,0) \}$,
$t1 := \{ (0,0,1), (0,1,0) \}$,
$t2 := \{ (0,0,2), (0,1,1), (0,2,0), (1,0,0) \}$,
$t3 := \{ (0,1,2), (0,2,1), (1,0,1), (1,1,0) \}$,
$t4 := \{ (0,2,2), (1,0,2), (1,1,1), (1,2,0), (2,0,0) \}$,
$t5 := \{ (1,1,2), (1,2,1), (2,0,1), (2,1,0) \}$,
$t6 := \{ (1,2,2), (2,0,2), (2,1,1), (2,2,0) \}$,
$t7 := \{ (2,1,2), (2,2,1) \}$,
$t8 := \{ (2,2,2) \}$.

Each time partition comprises independent iterations, which can be executed in parallel while time partitions should be enumerated in lexicographical order. In Fig. 2, labels $ti, i = 0, 1, 2, \ldots, 8$ mark a time partition comprising the corresponding iteration.

Next we combine time partitions into time slices each including a fixed number of time partitions. Provided that the number of time partitions in a slice is equal to 3, we get the following time slices, $TIMEi, i = 1, 2, 3$.

$TIME1 := t0 \cup t1 \cup t2 = \{ (0,1,2), (0,2,1), (1,0,1), (0,0,1), (1,1,0), (0,1,0), (0,0,0) \}$,

$TIME2 := t3 \cup t4 \cup t5 = \{ (0,2,2), (1,1,2), (0,1,2), (1,0,2), (1,2,1), (0,2,1), (1,1,1), (2,0,1), (1,0,1), (1,2,0), (2,1,0), (1,1,0), (2,0,0) \}$,

$TIME3 := t6 \cup t7 \cup t8 = \{ (2,2,2), (1,2,2), (2,1,2), (2,0,2), (2,2,1), (2,1,1), (2,2,0) \}$.

Enumerating time slices is valid under lexicographic order because before executing the next slice all data necessary are already calculated in the previous slices. So, within a single tile, we form sub-tiles represented with time slices whose execution in serial order increases code locality because instead of execution of one larger tile, multiple smaller sub-tiles will be executed.

To generate target tiles, we apply the intersection operation to sets representing sub-spaces and time slices. In the following section, we prove that for the Nussinov loop nest, resulting tiles are valid under lexicographical order and demonstrate how parallel target code can be generated.

## Methods

In compiler research, polytopes and related mathematical objects have been successfully applied to represent and manipulate an important class of compute- and data-intensive programs in an approach that has become known as the *polyhedral model*. It formalizes analyzing, parallelizing, and transforming program fragments consisting of (sequences of) arbitrarily nested loops (like dynamic programming loops), where the loop bounds, statements conditions and array accesses are affine combinations of symbolic constants and loop iterators.

The polyhedral method treats each iteration of a loop statement within the loop nest as an integral point inside mathematical objects called polyhedra that contains all iterations of the statement. A convex polyhedron can be formally defined as the set of solutions to a system of linear inequalities of the form $Mx \leq b$.

The polyhedral model of a loop nest includes i) a set representing an iteration space for each statement, ii) access relations (read and write) for each array available in the loop nest body, iii) relations describing a global schedule for each statement – a discrete time when a statement instance is executed according to the original iteration execution order.

A mathematical representation of a set is the following.

$S := PARAMS \rightarrow \{NAME(I) \mid constraints\}$, where $S$ is the set name, "$PARAMS \rightarrow$" means that the *constraints* include parameters *PARAMS*, each parameter is an arbitrary integer whose value defines an upper loop bound, $NAME(I)$ is the named tuple with name *NAME* and vector $I$ whose elements are loop indices or expressions including loop indices; *constraints* are comprised of affine equations and inequalities including vector $I$ and parameters *PARAMS* combined through the conjunction ($\wedge$), disjunction ($\vee$), projection ($\exists$), and negation ($\neg$) operators. For example, $S := N \rightarrow \{S1(i,j) \mid 1 <= i <= N \wedge 1 <= i <= N\}$ denotes a parametric set (regarding to parameter $N$) with name $S$, named tuple $S1(i,j)$, and constraints $1 <= i <= N \wedge 1 <= i <= N$.

Relations are defined in a similar way as sets, except that a single tuple is replaced with a pair of tuples separated by the arrow sign " $\rightarrow$ ", i.e, the mathematical representation of a relation is the following.

$R := PARAMS \rightarrow \{NAME1(I) \rightarrow NAME2(J) \mid constraints\}$,

where $R$ is the relation name, $NAME1(I)$ and $NAME2(J)$ are the named tuples with names $NAME1$ and $NAME2$ including vectors $I$ and $J$, respectively. For example, $R := N \rightarrow \{S1(i1,j1) \rightarrow S2(i1,j2) \mid 1 <= i1, i2 <= N \wedge 1 <=$

$i1, j2 <= N\}$ denotes the parametric relation with name $R$, tuples $S1(i1, j2), S2(i2, j2)$ and the constraints $1 <= i <= N1 <= i1, i2 <= N \wedge 1 <= i1, j2 <= N$. A relation maps elements represented with the first tuple to elements of the second one.

To extract the polyhedral model for a C program (all its three components), the PET (Polyhedral Extraction Tool) tool can be applied [18]. To extract dependences available in a loop nest, we use the polyhedral model returned with PET and apply the iscc calculator [19] to implement calculations on polyhedral sets and relations in a way presented in paper [20]. Iscc is an interactive interface to the barvinok counting library [21] and PET.

To optimize code, we use the following operations on relations and sets: intersection ($\cap$), union ($\cup$), difference (-), domain (dom $R$), range (ran $R$), relation application ($S' = R(S)$: $e' \in S'$ iff exists $e$ s.t. $e \rightarrow e' \in R, e \in S$) [21].

Given the loop nest, the iteration $i$ is lexicographically less than iteration $j$, denoted as $i \prec j$, if the following conditions are true $i_1 < j_1 \vee \exists k \geq 1 : i_k < j_k \wedge i_t = j_t, for\ t < k$.

The iteration domain of the Nussinov loop nest in Listing 1) is represented with the following parametric set comprising all the statement instances executed for statements S1 and S2.

$Iteration\ Domain := N \rightarrow \{S1(i,j,k) \mid 0 \leq i \leq N--1 \wedge i+1 \leq j \leq N--1 \wedge 0 \leq k \leq j-i-1; S2(i,j) \mid 0 \leq i \leq N--1 \wedge i+1 \leq j \leq N--1\}$,

where $S1(i,j,k), S2(i,j)$ are the tuples defining the iteration domain of the first and second statements of the Nussinov loop nest, respectively; the constraints of the tuples specify the value range of indices $i, j, k$ in the loop nest.

The relation representing dependences available in the examined loop nest is the following.

$R := N \rightarrow \{ S1(i,j,k) \rightarrow S2(i', i+j-i') \mid 0 \leq k < -i+j \wedge i' \geq -1+i \wedge i' \geq 0 \wedge -N+i+j < i' \leq i \} \cup N \rightarrow \{ S1(i,j,k) \rightarrow S1(i,j,k') \mid i \geq 0 \wedge j < N \wedge k \geq 0 \wedge k < k' < -i+j \} \cup N \rightarrow \{ S1(i,j,k) \rightarrow S1(i,j', -i+j) \mid i \geq 0 \wedge 0 \leq k < -i+j \wedge j < j' < N \} \cup N \rightarrow \{ S1(i,j,k) \rightarrow S1(i', j, -1+i-i') \mid j < N \wedge 0 \leq k < -i+j \wedge 0 \leq i' < i \} \cup N \rightarrow \{ S2(i,j) \rightarrow S1(i,j', -i+j) \mid i \geq 0 \wedge j > i \wedge j < j' < N \} \cup N \rightarrow \{ S2(i,j) \rightarrow S1(i', j, -1+i-i') \mid i < j < N \wedge 0 \leq i' < i \} \cup N \rightarrow \{ S2(i,j) \rightarrow S2(-1+i, 1+j) \mid i > 0 \wedge i < j \leq -2+N \}$,

where "$N \rightarrow$ " means that $N$ is a parameter, i.e., an arbitrary constant whose value defines loop upper bounds in the relation constraints. Relation $R$ is represented with a union ($\cup$) of simpler relations included in curly braces. Each simple relation includes two named tuples, for example, S1(...)→S2(...) and constraints. Elements of tuples are loop iterators or affine expressions whose components are loop iterators. The left tuple defines dependence sources while the right one states for the corresponding dependence destinations. Name S1 of the left tuple means that dependence sources are originated with statement S1 while S2 marks that dependence destinations are descended with statement S2. Each constraint includes affine inequalities whose elements are loop iterators and parameter $N$, for example , $0 \leq k < -i+j$, multiple inequalities are combined by means of the conjunction operator ($\wedge$).

Using relation $R$, we can discover the maximal number of rectangular sub-space types to be formed, this number is defined with the maximal number of the outer loops for which all elements of all distance vectors (formed as the differences between the range and domain of relation $R$) are non-negative. Such subspaces can be enumerated in lexicographical order because there is no cycle among them (the two corresponding elements of all distance vectors are non-negative).

Because statements S1 and S2 have different iteration spaces, we should form a global iteration space common for both statements S1 and S2. With this purpose, we apply the global schedule of the statements of the examined loop nest. It is a part of the loop nest polyhedral model and represented with a relation, which maps an iteration vector of a statement to a corresponding multidimensional timestamp, i.e., a discrete time when the statement instance has to be executed in the common iteration space. PET returns the following global schedule for the examined loop nest.

$SCHED\_GLOB := N \rightarrow \{ S2(i,j) \rightarrow (i,j,1,0) \} \cup N \rightarrow \{ S1(i,j,k) \rightarrow (i,j,0,k) \}$.

Next, we form relation, *R_GLOB*, by means of replacing each named tuple of relation $R$ with the tuple resulting due to applying relation *SCHED_GLOB* to named tuples $S_1$ and $S_2$. That relation describes dependences in the common iteration space and it is as follows.

$R\_GLOB := N \rightarrow \{ (i,j,0,k) \rightarrow (i,j,k',1,0) \mid i \geq 0 \wedge j < N \wedge k \geq 0 \wedge k < k' < -i+j \} \cup N \rightarrow \{ (i,j,0,k) \rightarrow (i', i+j-i', 1, 0) \mid 0 \leq k < -i+j \wedge i' \geq -1+i \wedge i' \geq 0 \wedge -N+i+j < i' \leq i \} \cup N \rightarrow \{ (i,j,0,k) \rightarrow (i,j',0,-i+j) \mid i \geq 0 \wedge 0 \leq k < -i+j \wedge j < j' < N \} \cup N \rightarrow \{ (i,j,0,k) \rightarrow (i', j, 0, -1+i-i') \mid j < N \wedge 0 \leq k < -i+j \wedge 0 \leq i' < i \} \cup N \rightarrow \{ (i,j,1,0) \rightarrow (i,j',0,-i+j) \mid i \geq 0 \wedge j > i \wedge j < j' < N \} \cup N \rightarrow \{ (i,j,1,0) \rightarrow (i', j, 0, -1+i-i') \mid i < j < N \wedge 0 \leq i' < i \} \cup N \rightarrow \{ (i,j,1,0) \rightarrow (-1+i, 1+j, 1, 0) \mid i > 0 \wedge i < j \leq -2+N \}$.

Then we apply the *deltas* operator of the iscc calculator to relation *R_GLOB* and get the following distance vectors in the global (common) iteration space.

$N \rightarrow \{ (i,-i,1,k) \mid -1 \leq i \leq 0 \wedge 2-N-2i \leq k \leq 0 \} \cup N \rightarrow \{ (i,0,0,k) \mid i < 0 \wedge -N-2i < k < -i \} \cup N \rightarrow \{ (0,j,0,k) \mid j > 0 \wedge 0 < k < N-j \} \cup N \rightarrow \{ (0,j,-1,i_3) \mid j > 0 \wedge 0 < i_3 < N-j \} \cup N \rightarrow \{ (i,0,-1,-1-i) \mid 2-N \leq i < 0 \} \cup N \rightarrow \{ (-1,1,0,0) \mid N \geq 4 \}$.

Taking into account that the value of iterator $i$ is decremented, we conclude that only the third element of each distance vector is negative. This allows us to state that sub-spaces located in parallel with plane $(j,k)$ and intersecting axis $i$ as well as sub-spaces located in parallel with plane $(i,k)$ and intersecting axis $j$, can be enumerated in lexicographical order because the both elements of all the corresponding dependence distance vectors are non-negative.

We split the iteration spaces of statements S1 and S2 into sub-spaces of width 16 (any other constant width can be chosen). For statement S1, sets $SPACE_{11}$ and $SPACE_{12}$ below represent sub-spaces intersecting axes $i$ and $j$, respectively.

$SPACE_{11} := (ii,N) \rightarrow \{\,\mathrm{S1}(i,j,k) \mid ii \geq 0 \wedge N > 0 \wedge i \geq 0 \wedge -16 - 16ii + N \leq i < -16ii + N \wedge i \leq -2 + N \wedge i < j < N \wedge 0 \leq k < -i + j\,\}$,

$SPACE_{12} := (jj,N) \rightarrow \{\,\mathrm{S1}(i,j,k) \mid jj \geq 0 \wedge N > 0 \wedge 0 \leq i \leq -2+N \wedge j > 16jj+i \wedge i < j < N \wedge j \leq 16+16jj+i \wedge 0 \leq k < -i+j\,\}$.

For statement S2, sets $SPACE_{21}$ and $SPACE_{22}$ below represent sub-spaces intersecting axes $i$ and $j$, respectively.

$SPACE_{21} := (ii,N) \rightarrow \{\,\mathrm{S2}(i,j) \mid ii \geq 0 \wedge N > 0 \wedge i \geq 0 \wedge -16-16ii+N \leq i < -16ii+N \wedge i \leq -2+N \wedge i < j < N\,\}$,

$SPACE_{22} := (jj,N) \rightarrow \{\,\mathrm{S2}(i,j) \mid jj \geq 0 \wedge N > 0 \wedge 0 \leq i \leq -2+N \wedge j > 16jj+i \wedge i < j < N \wedge j \leq 16+16jj+i\,\}$.

Variables $ii$ and $jj$ are the parametric identifiers of sub-spaces. The intersection of the union of sets $SPACE_{11}$, $SPACE_{12}$ and the union of sets $SPACE_{21}$, $SPACE_{22}$ results in tiles whose size is limited along axes $i$ and $j$ with the width of sub-spaces (16), but the size of those tiles is not limited along axis $k$. Fig. 1 presents such sub-spaces and tiles for statement S1, integers in brackets are the identifiers of sub-spaces and tiles. Blue lines depict planes situated in parallel with plane $(j,k)$ while red ones present planes located in parallel with plane $(i,k)$.

For large $N$, the entire data associated with each such a tile cannot be held at cache, this leads to decreasing code locality. To improve code locality, we form time slices each including a constant number of time partitions. Each time partition holds statement instances that can be run in parallel for a given schedule. For a time slice, the number of statement instances within axes $k$ is limited with a constant number of time partitions within that slice. Let us suppose that a time slice is represented with a parametric set, *TIME*, while a set representing the results of the intersection of above mentioned sets is named as *SPACE*. Then the intersection of sets *TIME* and *SPACE* results in a parametric set describing tiles whose size is limited along all axes: $i,j$, and $k$. Choosing the proper width of sub-spaces and the proper number of time partitions within a time slice, we may obtain tiles for which the entire data associated with each of them can be held at cache, this can improve significantly code locality.

To form time partitions, we apply the loop skewing transformation [22]. It is a convenient method to implement the wavefront method of executing a loop nest in parallel, which creates a "wave" that passes through the iteration space. Skewing changes the iteration vectors for each iteration by adding the outer loop index value to the inner one, for example, for a loop nest of depth 2 with iterators $i$ and $j$, iteration $(i,j)$ becomes relabeled as $(i,i+j)$. If all distance vectors of a new loop nest with iterators $i$ and $i+j$ comprise only non-negative elements, interchanging those iterators, i.e., forming new iterators $i+j$ and $i$ allows for generation of a new loop nest, where the outermost loop $i+j$ is serial while $i$ is parallel. In general, for a loop nest of depth $d$, a new loop nest with iterators $(i_1+i_2+\ldots+i_d, i_1, i_2, \ldots, i_{d-1})$ is valid if all distance vectors comprise only non-negative elements; the first loop is serial while the reminding ones are parallel [22].

For the Nussinov loop nest, we first form the following schedule.

$SCHED := N \rightarrow \{\,\mathrm{S1}(i,j,k) \rightarrow (-i+j,k) \mid N > 0 \wedge 0 \leq i \leq -2+N \wedge i < j < N \wedge 0 \leq k < -i+j\,\} \cup N \rightarrow \{\,\mathrm{S2}(i,j) \rightarrow (-i+j,j) \mid N > 0 \wedge 0 \leq i \leq -2+N \wedge i < j < N\,\}$.

That schedule maps each instance of statements S1 and S2 to two-dimensional time $(-i+j,k)$ and $(-i+j,j)$, respectively. To check whether that schedule is valid, we apply the way suggested in paper [23], which envisages checking whether the following inequality $\delta((SCHED^{-1}) \,.\, R \,.\, SCHED) \geq 0$ is true, where $R$ is the relation describing all the dependences available in the examined loop nest, it is presented above; "." is the iscc join (composition) operator of two relations; $\delta$ is the *deltas* iscc operator that maps a relation to the differences between image and domain elements.

Using such a checking, we conclude that relation *SCHED* above is valid. By means of relation *SCHED*, we form set *TIME* representing time slices each including 16 time partitions (any other constant value can be chosen). With this purpose, we first calculate the inverse relation, $SCHED^{-1}$, of relation *SCHED*.

$SCHED^{-1} := N \rightarrow \{\,(i0,i1) \rightarrow \mathrm{S10}(i,i0+i,i1) \mid N > 0 \wedge i_0 > 0 \wedge 0 \leq i_1 < i0 \wedge 0 \leq i < N - i0 \wedge i \leq -2+N\,\} \cup N \rightarrow \{\,(i_0,i_1) \rightarrow \mathrm{S12}(-i0+i1,i1) \mid N > 0 \wedge i_0 > 0 \wedge i0 \leq i_1 < N \wedge i_1 \leq -2+N+i0\,\}$.

In the relation above, variables $i0, i1$ represent two-dimensional time. To form set *TIME*, we i) make $i0$ to be the parameter of set *TILE*, ii) make the right tuple and the constraints of relation $SCHED^{-1}$ to be the tuple and constraints of set *TIME*, iii) introduce parameter $tt$ of set *TIME* and add the constraints of the form $i1 \geq 16tt \wedge 0 \leq i1 \leq 15 + 16tt$ to the constraints of set *TIME*, those constraints mean that the width of a time slice is 16 (any other constant value can be chosen) and time partitions within a time

slice are dependent on parameter $tt$. This results in the following set.

$TIME := (i0, tt, N) \rightarrow \{ S1(i, i0 + i, i1) \mid i0 > 0 \wedge N > 0 \wedge 0 \leq i < -i0 + N \wedge i \leq -2 + N \wedge i1 \geq 16tt \wedge 0 \leq i1 \leq 15 + 16tt \wedge i1 < i0 \} \cup (i0, tt, N) \rightarrow \{ S2(i, i0 + i) \mid i0 > 0 \wedge N > 0 \wedge i \geq 0 \wedge -i0 + 16tt \leq i < -i0 + N \wedge i \leq 15 - i0 + 16tt \wedge i \leq -2 + N \}$.

We calculate set *TILE*, which represents target tiles, as follows

$TILE := TIME\ .\ (SPACE_{11} \cup SPACE_{12})\ .\ (SPACE_{21} \cup SPACE_{22}) = (N, jj, ii, i0, tt) \rightarrow \{ S2(i, i0 + i) \mid jj \geq 0 \wedge N > 0 \wedge ii \geq 0 \wedge i0 > 16jj \wedge 0 < i0 \leq 16 + 16jj \wedge i \geq -16 + N - 16ii \wedge i \geq 0 \wedge -i0 + 16tt \leq i \leq 15 - i0 + 16tt \wedge i < N - 16ii \wedge i < N - i0 \wedge i \leq -2 + N \} \cup (jj, N, ii, i0, tt) \rightarrow \{ S1(i, i0 + i, k) \mid jj \geq 0 \wedge N > 0 \wedge ii \geq 0 \wedge i0 > 16jj \wedge 0 < i0 \leq 16 + 16jj \wedge i \geq -16 + N - 16ii \wedge 0 \leq i < N - 16ii \wedge i < N - i0 \wedge i \leq -2 + N \wedge k \geq 16tt \wedge 0 \leq k \leq 15 + 16tt \wedge k < i0 \}$.

To find out what is the number of statement instances within a tile represented with the set above, we applied the iscc *card* operator to set *TILE*, which calculates the number of elements within a set. The analysis of the result returned with that operator allows us to conclude that the size of each tile is not parameterized, i.e., the number of elements within each tile does not depend on the parameter $N$ defining the upper bounds in the Nussinov loop nest as it takes place when the tile correction technique [15] is applied to generated target tiles.

Let us re-write set *TILE* in the following form.

$TILE := (N, jj, ii, i0, tt) \rightarrow \{ S1(i, i0 + i, k) \mid constraints_1;\ S2(i, i0 + i) \mid constraints_2 \}$.

To generate parallel code on the tile level, we apply the skewing transformation $(ii + jj)$ to form the following schedule allowing for parallel code generation.

$SCHED\_PAR := N \rightarrow \{ S1(i, i0 + i, k) \rightarrow (ii + jj, jj, i0, tt, i, i0 + i, k) \mid constraints_1; S2(i, i0 + i) \rightarrow (ii + jj, jj, i0, tt, i, i0 + i) \mid constraints_2 \}$,

where $constraints_1$ and $constraints_2$ are the constraints of set *TILE* above.

That schedule maps each instance of statements S1 and S2 to a time partition whose all tiles can be executed in parallel. To check the validity of that schedule we again apply the technique presented in paper [23] and explained above to relation *SCHED_PAR*, and confirm that the schedule is valid.

Applying the iscc *codegen* operator to relation *SCHED_PAR*, we generate pseudocode and postprocess it to the parallel tiled code presented in Listing 2. In that code, the outermost loop $c0$ enumerates serially time partitions including tiles, loop $c1$ scans tiles to be executed in parallel for each time partition, loops $c3, c4$, and $c6$ enumerate serially statement instances within each tile, *pragma omp parallel for* makes loop $c1$ parallel [24].

## Results

We conducted experiments on RNA secondary structure prediction using a machine with 2 × Intel Xeon processors E5-2695 v2, 2.4 GHz, 12 cores/24 threads, 256KB L2 Cache and 30MB L3 Cache, and 128 GB RAM. All programs were compiled with using of the Intel C++ Compiler (*icc* 17.0.1) with the -O3 flag of optimization. The code parallelism is presented with the OpenMP programming interface [24].

At the address traco.sourceforge.net/nuss.tar.gz, all source codes used for carrying out experiments can be found as well as a program allowing us to run each parallel program for a prepared sequence in the FASTA format and obtain a target Nussinov table.

To carry out experiments, we used randomly generated RNA strands of length from 2500 to 15000. Papers [5, 6, 15] show that cache efficient code performance does not change based on strings themselves, but it depends on the size of a string.

**Listing 2** Parallel tiled code generated with the space-time approach and implementing Nussinov's algorithm.

```
for( c0 = 0; c0 <= floord(N - 2, 8); c0 += 1)
  #pragma omp parallel for
  for( c1 = (c0 + 1) / 2; c1 <= min(c0, (N - 1) / 16); c1 += 1)
    for( c3 = 16 * c0 - 16 * c1 + 1; c3 <= min(min(N - 1, 16 * c1 + 15), 16 * c0 - 16 * c1 + 16); c3 += 1) {
      for( c4 = 0; c4 <= c0 - c1; c4 += 1)
        for( c6 = max(-N + 16*c1 + 1, -N + c3 + 1); c6 <= min(0, -N + 16*c1 + 16); c6++) {
          for( c10 = 16 * c4; c10 <= min(c3 - 1, 16 * c4 + 15); c10 += 1)
            S[-c6][(c3-c6)] = MAX(S[-c6][c10-c6] + S[c10-c6+1][(c3-c6)], S[-c6][(c3-c6)]);
          if (c1 + c4 == c0 && 16 * c0 + c6 + 15 >= 16 * c1 + c3)
            S[-c6][(c3-c6)] = MAX(S[-c6][(c3-c6)], S[-c6+1][(c3-c6)-1] + sigma[-c6][(c3-c6)] );
        }
      for( c4 = max(c0 - c1 + 1, -c1 + (N + c3) / 16 - 1); c4 <= min((N - 1) / 16, -c1 + (N + c3 - 1) / 16); c4 += 1)
        for( c6 = max(max(-N + 16 * c1 + 1, -N + c3 + 1), c3 - 16 * c4 - 15); c6 <= min(-N + 16 * c1 + 16, c3 - 16 * c4); c6 += 1)
          S[-c6][(c3-c6)] = MAX(S[-c6][(c3-c6)], S[-c6+1][(c3-c6)-1]+sigma[-c6][(c3-c6)]);
    }
```

**Table 1** Execution times of the examined codes depending on the thread number for $N = 5000$

| $N = 5000$ | Original | Tile correction | Tile correction | Pluto (ATF) | Space-Time Tiling | Li (Transpose) |
|---|---|---|---|---|---|---|
| Threads | | 16×16×16 | 1×128×16 | 16×16×1 | 16×16×16 | |
| 1 | 243.4322 | 132.3891 | 77.4809 | 130.748 | 57.7063 | 66.56 |
| 4 | | 57.5215 | 19.5745 | 64.9885 | 21.634 | 21.24 |
| 8 | | 38.4351 | 11.8201 | 32.2481 | 12.563 | 14.05 |
| 16 | | 18.455 | 7.8852 | 18.662 | 7.1172 | 12.39 |
| 32 | | 17.3659 | 7.9378 | 23.1456 | 6.8839 | 12.44 |
| 48 | | 16.0771 | 8.7662 | 19.0443 | 5.7779 | 13.72 |

We compared the performance of code generated with the presented approach with that of i) Pluto parallel tiled code (based on affine transformations), ii) tiled code based on the correction technique [15, 17], and iii) the Li manual cache efficient implementation [5] of Nussinov's RNA folding. The tile size 16×16×1 for Pluto code [14] was chosen empirically (Pluto does not tile the most inner loop) as the best among many sizes examined. For the code generated with the presented approach, the tile size 16×16×16 was chosen from many different tile sizes, examined by us, as one exposing the highest code performance. We used this tile size also for tiled code generated with the tile correction technique based on the transitive closure of dependence graphs. We experimented with tiled code based on correction also for the best tile size demonstrated in paper [17], 1×128×16.

The results in Table 1, including the execution time of the examined programs and Fig. 3 graphically presenting the corresponding code speed up against the original code, demonstrate that only the code generated with the suggested approach is scalable up to 48 threads, i.e., increasing the number of threads reduces the time of code execution. The results in Table 1 show that space-time tiled code implementing Nussinov's algorithm with the tile size [16×16×16] outperforms all examined implementations for a larger number of threads (equal or greater than 16) for $N$=5000.

Table 2 and Fig. 4 present execution time and speed up for various RNA sequence lengths, respectively. We can see that the presented approach allows for obtaining cache efficient tiled code, which outperforms the other examined implementations for each length.

## Discussion

In paper [17], we showed that tile correction allows for generation code of the best performance when the outermost loop is serial. We revealed that the best tile size (for tile correction) is not optimal for the code generated with the suggested approach in that case, tiling all loops is a better solution.

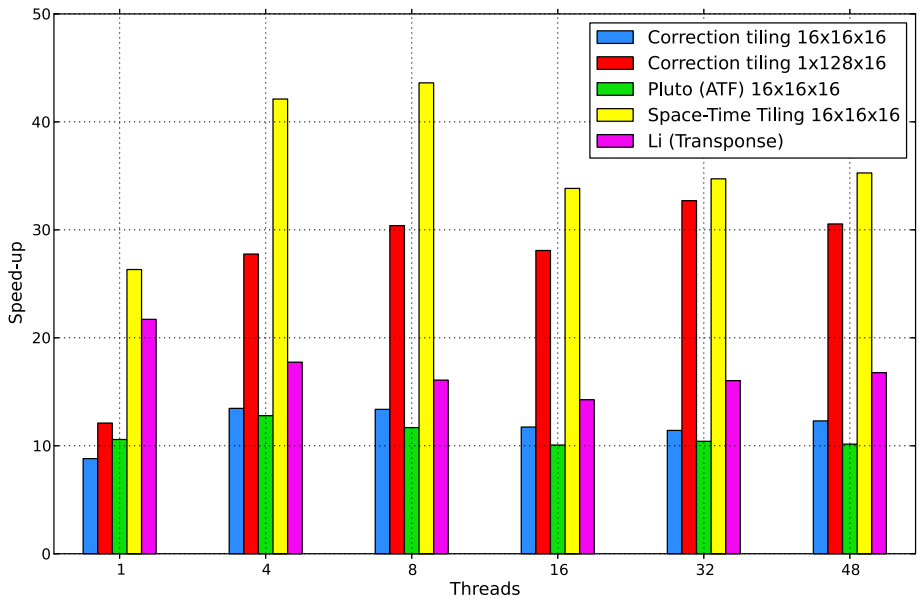


**Fig. 3** Speed up of the examined codes implementing Nussinov's RNA folding depending on the number of threads

**Table 2** Execution times of the examined codes depending on the RNA sequence length for 48 threads used

| 48 Threads | Original | Tile correction | Tile correction | Pluto (ATF) | Space-Time Tiling | Li (Transpose) |
|---|---|---|---|---|---|---|
| Seq. length | | 16×16×16 | 1×128×16 | 16×16×1 | 16×16×16 | |
| 2500 | 21.06 | 2.39 | 1.74 | 1.99 | 0.80 | 0.97 |
| 5000 | 243.43 | 18.08 | 8.77 | 19.04 | 5.78 | 13.72 |
| 7500 | 812.10 | 60.72 | 26.72 | 69.49 | 18.62 | 50.50 |
| 10000 | 1709.43 | 145.66 | 60.86 | 169.76 | 50.52 | 119.82 |
| 12500 | 3450.01 | 302.01 | 105.50 | 331.21 | 99.35 | 215.20 |
| 15000 | 5863.83 | 476.65 | 192.01 | 577.76 | 166.29 | 349.76 |

Space-time tiled code achieves super-linear speed up (greater than the number of threads used) of serial and parallel tiled code against the original serial code up to 32 threads. However super linear speed up scalability (increasing speed-up with increasing the thread number) is observed only from one to eight threads. Code regularity and lack of parametric large tiles allow us to use the entire power of the multi-processor machine with the maximal number of threads while the speed up of Li's code and that based on tile correction is significantly limited in this case.

The transposition of the Nussinov array [5] is effective only for short RNA strands with the length equal to 2500. For longer sequences (greater than 7500), only the performance of code based on applying transitive closure with the best tile size is comparable, but it is still worse than the performance of code generated with the approach presented in this paper.

Summing up, we may conclude that the parallel tiled code generated by means of the presented space-time approach is the fastest implementation among all examined ones including the manual generated Li code. Code regularity and fixed tiles are dominant factors in achieving high code performance and scalability in comparison to our previous implementation based on tile correction. We achieved the best code performance for each RNA sequence length using all cores/threads on the studied Intel Xeon machine.

## Conclusion

In this paper, we introduced a space-time tiling approach for the loop nest implementing Nussinov's folding. It allows us to generate parallel tiled code, which outperforms known related codes generated automatically by means of affine transformations, tile correction based on the transitive closure of dependence graphs, and manually

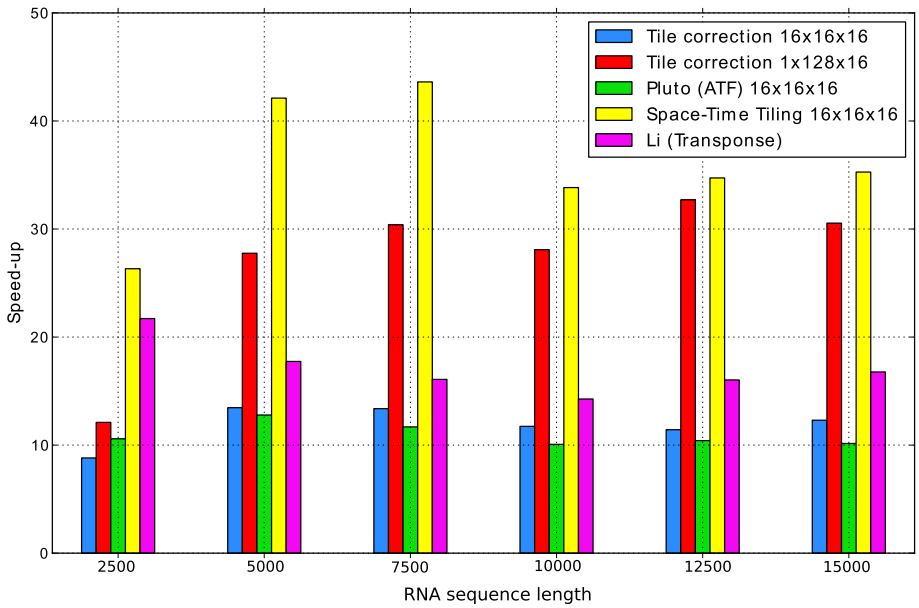


**Fig. 4** Speed up of the examined codes depending on the RNA sequence length for 48 threads used

generated Li's transposition code. The presented approach enables us to tile all three loops of Nussinov's recurrence that is not possible with commonly known optimizing compilers based on affine transformations. The approach generates 3-D tiles for the Nussinov loop nest.

The results of an experimental study allow us to conclude that the generated code based on the space-time approach i) outperforms known related codes, ii) is scalable regarding to the number of parallel threads (execution time decreases with increasing the thread number up to 48 threads); iii) allows for achieving super-linear speed up (greater than the number of threads used) of serial and parallel tiled code against the original serial code up to 32 threads and super-linear speed up scalability (increasing speed up with increasing the thread number) up to 8 threads, for one thread, speed up is about 4.2.

The presented code optimization can be applied to other dynamic programming kernels, for example, DNA sequence alignment or energy minimization for RNA folding. In the future, we plan to apply space-time tiling to Zuker's algorithm allowing for energy minimization. It is much complex than the algorithm examined in this paper – there are four nested loops and multiple complicated statements within the corresponding loop nest. However that code is still within the polyhedral model, so space-time tiling can be applied to that code to tile all loops.

#### Abbreviations
ATF: Affine transformation framework; FPGA: field-programmable gate array; NPDP: Nonserial polyadic dynamic programming

#### Acknowledgements
Not applicable.

#### Funding
No specific funding was received for this study.

#### Availability of data and materials
At the address traco.sourceforge.net/nuss.tar.gz, all source codes used for carrying out experiments can be found as well as a program allowing us to run each parallel program for a prepared sequence in the FASTA format and obtain a target Nussinov table.

#### Authors' contributions
MP verified the main concept of the presented technique, analysed related work, implemented the approach and carried out the experimental study. WB proposed the main concept of the presented technique, checked the correctness of the presented technique, participated in its implementation and the analysis of the results of the experimental study. All authors read and approved the final manuscript.

#### Ethics approval and consent to participate
Not applicable.

#### Consent for publication
Not applicable.

#### Competing interests
The authors declare that they have no competing interests.

## Publisher's Note


Received: 25 June 2018 Accepted: 1 April 2019
Published online: 24 April 2019

#### References
1. Nussinov R, Pieczenik G, Griggs JR, Kleitman DJ. Algorithms for Loop Matchings. SIAM J Appl Math. 1978;35(1):68–82.
2. Mathuriya A, Bader DA, Heitsch CE, Harvey SC. Gtfold: A scalable multicore code for rna secondary structure prediction. In: Proceedings of the 2009 ACM Symposium on Applied Computing. SAC '09. New York: ACM; 2009. p. 981–8.
3. Markham NR, Zuker M. In: Keith JM, editor. UNAFold. Totowa: Humana Press; 2008. pp. 3–31.
4. Hofacker IF, Stadler PF. Memory efficient folding algorithms for circular RNA secondary structures. Bioinformatics. 2006;22(10):1172–6. https://academic.oup.com/bioinformatics/article/22/10/1172/236586.
5. Li J, Ranka S, Sahni S. Multicore and GPU algorithms for Nussinov RNA folding. BMC Bioinformatics. 2014;15(8):1. https://doi.org/10.1186/1471-2105-15-S8-S1.
6. Zhao C, Sahni S. Cache and energy efficient algorithms for nussinov's rna folding. BMC Bioinformatics. 2017;18(15):518.
7. Liu L, Wang M, Jiang J, Li R, Yang G. Efficient nonserial polyadic dynamic programming on the cell processor. In: IPDPS Workshops. Anchorage: IEEE; 2011. p. 460–71.
8. Almeida F, et al. Optimal tiling for the rna base pairing problem. In: Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures. SPAA '02. New York: ACM; 2002. p. 173–82. https://doi.org/10.1145/564870.564901.
9. Tan G, Feng S, Sun N. Locality and parallelism optimization for dynamic programming algorithm in bioinformatics. In: SC 2006 Conference, Proceedings of the ACM/IEEE. Tampa: IEEE; 2006. p. 41.
10. Jacob A, Buhler J, Chamberlain RD. Accelerating Nussinov RNA secondary structure prediction with systolic arrays on FPGAs. In: Proceedings of the 2008 International Conference on Application-Specific Systems, Architectures and Processors. ASAP '08. Washington: IEEE Computer Society; 2008. p. 191–6. https://doi.org/10.1109/ASAP.2008.4580177.
11. Mullapudi RT, Bondhugula U. Tiling for dynamic scheduling. In: Rajopadhye S, Verdoolaege S, editors. Proceedings of the 4th International Workshop on Polyhedral Compilation Techniques. Vienna; 2014. http://impact.gforge.inria.fr/impact2014/papers/impact2014-mullapudi.pdf. Accessed 15 Apr 2019.
12. Hammami E, Slama Y. An overview on loop tiling techniques for code generation. In: 2017 IEEE/ACS 14th International Conference on Computer Systems and Applications (AICCSA); 2017. p. 280–7. https://doi.org/10.1109/AICCSA.2017.168.
13. Xue J. Loop Tiling for Parallelism. Norwell: Kluwer Academic Publishers; 2000.
14. Bondhugula U, Hartono A, Ramanujam J, Sadayappan P. A practical automatic polyhedral parallelizer and locality optimizer. SIGPLAN Not. 2008;43(6):101–13.
15. Palkowski M, Bielecki W. Parallel tiled Nussinov rna folding loop nest generated using both dependence graph transitive closure and loop skewing. BMC Bioinformatics. 2017;18(1):290. https://doi.org/10.1186/s12859-017-1707-8.
16. Wonnacott D, Jin T, Lake A. Automatic tiling of 'mostly-tileable" loop nests. In: IMPACT 2015: 5th, At Amsterdam, The Netherlands; 2015. http://impact.gforge.inria.fr/impact2015/papers/impact2015-wonnacott.pdf. Accessed 15 Apr 2019.
17. Palkowski M, Bielecki W. Tuning iteration space slicing based tiled multi-core code implementing nussinov's rna folding. BMC Bioinformatics. 2018;19(1):12.
18. Verdoolaege S, Grosser T. Polyhedral extraction tool. In: In Proceedings of the 2nd International Workshop on Polyhedral Compilation Techniques. Paris; 2012. http://impact.gforge.inria.fr/impact2012/workshop_IMPACT/verdoolaege.pdf. Accessed 15 Apr 2019.
19. Verdoolaege S. Counting affine calculator and applications. In: First International Workshop on Polyhedral Compilation Techniques (IMPACT'11), Charmonix, France; 2011. http://perso.ens-lyon.fr/christophe.alias/impact2011/impact-05.pdf. Accessed 15 Apr 2019.
20. Verdoolaege S. Integer set library - manual, http://isl.gforge.inria.fr/manual.pdf. Technical report. 2011.

21. Verdoolaege S. barvinok: User guide 0.41; 2018. http://barvinok.gforge.inria.fr/barvinok.pdf. Accessed 15 Apr 2019.
22. Wolfe M. Loops skewing: The wavefront method revisited. Int J Parallel Prog. 1986;15(4):279–93.
23. Verdoolaege S, Carlos Juega J, Cohen A, Ignacio Gomez J, Tenllado C, Catthoor F. Polyhedral parallel code generation for cuda. ACM Trans Archit Code Optim (TACO). 2013;9(4):54.
24. OpenMP Architecture Review Board. OpenMP Application Program Interface Version 4.5. 2015. https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf. Accessed 15 Apr 2019.