Palkowski and Bielecki *BMC Bioinformatics* (2018) 19:12
DOI 10.1186/s12859-018-2008-6

BMC Bioinformatics

RESEARCH ARTICLE Open Access

Tuning iteration space slicing based tiled multi-core code implementing Nussinov's RNA folding

Marek Palkowski* and Wlodzimierz Bielecki

Abstract

Background: RNA folding is an ongoing compute-intensive task of bioinformatics. Parallelization and improving code locality for this kind of algorithms is one of the most relevant areas in computational biology. Fortunately, RNA secondary structure approaches, such as Nussinov's recurrence, involve mathematical operations over affine control loops whose iteration space can be represented by the polyhedral model. This allows us to apply powerful polyhedral compilation techniques based on the transitive closure of dependence graphs to generate parallel tiled code implementing Nussinov's RNA folding. Such techniques are within the iteration space slicing framework – the transitive dependences are applied to the statement instances of interest to produce valid tiles. The main problem at generating parallel tiled code is defining a proper tile size and tile dimension which impact parallelism degree and code locality.

Results: To choose the best tile size and tile dimension, we first construct parallel parametric tiled code (parameters are variables defining tile size). With this purpose, we first generate two nonparametric tiled codes with different fixed tile sizes but with the same code structure and then derive a general affine model, which describes all integer factors available in expressions of those codes. Using this model and known integer factors present in the mentioned expressions (they define the left-hand side of the model), we find unknown integers in this model for each integer factor available in the same fixed tiled code position and replace in this code expressions, including integer factors, with those including parameters. Then we use this parallel parametric tiled code to implement the well-known tile size selection (TSS) technique, which allows us to discover in a given search space the best tile size and tile dimension maximizing target code performance.

Conclusions: For a given search space, the presented approach allows us to choose the best tile size and tile dimension in parallel tiled code implementing Nussinov's RNA folding. Experimental results, received on modern Intel multi-core processors, demonstrate that this code outperforms known closely related implementations when the length of RNA strands is bigger than 2500.

Keywords: RNA folding, Parametric loop tiling, Computational biology, Nussinov's algorithm, Parallel computing, Tile size selection

Background

RNA structure prediction, or folding, is an important ongoing problem that lies at the core of several search applications in computational biology. Algorithms to predict the structure of single RNA molecules find a structure of minimum free energy for a given RNA using dynamic programming. Nussinov's folding algorithm [1] uses the number of base pairs as a proxy for free energy, preferring the structure with the most base pairs.

Nussinov's algorithm is compute intensive due to a cubic time complexity. Fortunately, it involves mathematical operations over affine control loops whose iteration space can be represented by the polyhedral model [2]. Thanks to the simple pattern of dependences, loop tiling techniques can be used to accelerate Nussinov's folding.

Let S be an $N \times N$ Nussinov matrix and $\sigma(i,j)$ be a function which returns 1 if (x_i, x_j) match and $i < j - 1$, or 0

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

otherwise, then the following recursion $S(i,j)$ (the maximum number of base-pair matches of $x_i, ..., x_j$) is defined over the region $1 \leq i < j \leq N$ as

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

and $S(i,j)$ is zero beyond that region.

Listing 1 represents the loop nest implementing Nussinov's algorithm. It consists of triply nested affine loops with two statements accessing to two-dimensional array S.

Listing 1 Nussinov 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  }
```

Loop tiling or blocking is a crucial program transformation, which offers a number of benefits. It is used to improve code locality, expose parallelism, and allow for adjusting parallel code granularity or balance. All those factors impact parallel code performance [3].

In paper [4], we presented loop tiling based on the transitive closure of a dependence graph for Nussinov's algorithm. It is within the iteration space slicing (ISS) framework [5]. The key step in calculating an iteration space slice is to calculate the transitive closure of the data dependence graph of the program; then transitive dependences are applied to the statement instances of interest to produce valid tiles. The idea of tiling, presented in paper [4], is to transform (correct) original rectangular fixed tiles so that all target tiles are valid under lexicographic order. We demonstrated higher speed-up of generated tiled code (for a properly chosen size of original tiles) than that of code produced with state-of the-art source-to-source optimizing compilers. But that paper does not answer what is the best size of original tiles allowing for generation of tiled code demonstrating the maximal speed-up. In general, the number of combinations of possible tile sizes can be very large. For each tile size, it is necessary to generate tiled code, compile and spawn it, and finally carry out code profiling. This can result in very high expenses not allowing for discovering the best tile size in practice.

The goal of this paper is to present an approach allowing us to determine the best tile size maximizing tiled code performance to be applied in practice. This approach is based on parametric tiling.

Parametric tiling is more general, it allows for defining tile size with parameters instead of constants [3]. With fixed size tiling, a separate program must be generated and compiled each time when tile size is changed. In general, this can be very expensive. Thereby, parametric tiling is more flexible and time and cost saving when we deal with code locality analysis and tuning code for target architectures. However, most state-of-the-art compilation tools do not provide parametric tiling, they are able to generate tiled code for only fixed tile size. Parametric tiling is generally known to be non-linear, breaking the mathematical closure properties of the polyhedral model.

To our best knowledge, well-known tiling techniques and optimizing compilers are based on linear or affine transformations [6–8], for example, the-state-of-the-art PluTo compiler [6] generates tiled code applying affine transformations derived. However, PluTo can only generate tiled code when tile size is fixed.

PrimeTile [9] is the first system to generate parametrically tiled code for affine imperfectly nested loops. It uses a level by level approach to generate tiled code, with a prolog, epilog, and a full-tiles loop nest corresponding to each nesting level of the original code. But loop tiling is generated seamlessly in the affine transformation framework.

DynTile [10] utilizes wavefront parallelism in the tiled iteration space corresponding to the convex hull of all the statement domains of the input untiled code. Tiles are scheduled dynamically, i.e., at run time.

PTile [11] is an approach to compile-time generation of code for wavefront parallel tiled execution.

Although DynTile, PTile, and PrimeTile present very effective tiling for stencils, using affine loop transformations, they do not allow us to tile dynamic programming kernels efficiently, in particular, they fail to tile the innermost loop in the code implementing Nussinov's algorithm [2]. We show in this paper that tiling of that loop is crucial to achieve high performance. Furtermore, known techniques of mono-parametric tiling [3] (tile sizes are multiple of the same block parameter) do not guarantee notable locality improvements for Nussinov's algorithm. To our best knowledge, there does not exist any parametric loop tiling scheme for the loop nest implementing Nussinov's algorithm.

Mullapudi and Bondhugula presented dynamic tiling for Zuker's optimal RNA secondary structure prediction [2] to overcome limitations of affine transformations. 3-D iterative tiling for dynamic scheduling is

calculated by means of reduction chains. Operations along each chain find maximum and can be reordered to eliminate cycles. Their approach involves dynamic scheduling of tiles, rather than the generation of a static schedule.

Wonnacott et al. introduced serial 3-D tiling of "mostly-tileable" loop nests of Nussinov's RNA secondary structure prediction in paper [12]. This approach tiles non-problematic iterations (iterations of loops *'i'* and *'j'*) with classic tiling strategies while problematic iterations of loop (*'k'*) are peeled off and executed later. Unfortunately, the paper does not consider any parallel code, tiling is represented with serial code.

In this paper, we present an approach allowing for deriving the best size of original tiles to be used for generation of ISS based tiled code implementing Nussinov's RNA folding.

Methods

Brief introduction

The *polyhedral model* is a mathematical formalism for analyzing, parallelizing, and transforming an important class of compute- and data-intensive programs, or program fragments consisting of (sequences of) arbitrarily nested loops. Loop bounds, statements conditions and array accesses are affine functions in the program.

Within the polyhedral model for analysis and transformation of affine programs, we deal with sets and relations whose constraints need to be affine, i.e., presented with linear expressions and constant terms. Affine constraints may be combined through the conjunction (and), disjunction (or), projection (exists), and negation (not) operators.

An access relation connects iterations of a statement to the array elements accessed by those iterations. Relations are defined in similar way as sets, except that the single space is replaced by a pair of spaces separated by the arrow sign $\rightarrow$. We use the exact dependence analysis proposed by Pugh and Wonnacott [13], where loop dependences are represented with *relations*.

Standard operations on relations and sets are used, such as intersection ($\cap$), union ($\cup$), difference (-), domain (dom R), range (ran R), relation application ($S' = R(S)$: $e' \in S'$ iff exists e s.t. $e \rightarrow e' \in R, e \in S$). The detailed description of these operations is presented in [13].

The *positive transitive closure* of a given lexicographically forward dependence relation R, R^+, is defined as follows [5]:

$$R^+ = \left\{ e \rightarrow e' : \; e \rightarrow e' \in R \vee \exists e'' s.t.\; e \rightarrow e'' \in R \wedge e'' \rightarrow e' \in R^+ \right\}.$$

It describes which vertices e' in a dependence graph (represented by relation R) are connected directly or transitively with vertex e.

In sequential loop nests, the iteration i executes before j if i is *lexicographically less* than j, denoted as $i \prec j$, i.e., $i_1 < j_1 \vee \exists k \geq 1 : i_k < j_k \wedge i_t = j_t, for\ t < k$.

Generation of tiles for the Nussinov loop nest

Let us recap tiled code generation for Nussinov's algorithm presented in [4]. To generate valid 3-D tiled code for the Nussinov loop nest, we adopt the approach presented in paper [14], which is based on the transitive closure of dependence graphs.

The iteration domain of the Nussinov loop nest (see Listing 1) is represented with the following set.

$$Iteration\ Domain = \begin{cases} i : 0 \leq i \leq N-1, \\ j : i+1 \leq j \leq N-1, \\ k : \begin{cases} s1 : 0 \leq k \leq j-i-1, \\ s2 : k = 0. \end{cases} \end{cases}$$

Let vector $\boldsymbol{I} = (i,j,k)^T$ define indices of the Nussinov loop nest, diagonal matrix $\boldsymbol{B} = [b_1, b_2, b_3]$ define tile sizes, vectors $\boldsymbol{II} = (ii, jj, kk)^T$ and $\boldsymbol{II'} = (iip, jjp, kkp)^T$ specify tile identifiers. Each tile identifier is represented with a non-negative integer, i.e., the following constraint $\boldsymbol{II} \geq 0$ has to be satisfied.

First, we form parametric set, $TILE(\boldsymbol{II}, \boldsymbol{B})$, including statement instances belonging to a parametric rectangular tile (parameters are tile identifiers) as follows

$$TILE = \begin{cases} i : N-1-b_1 * ii \geq i \geq max(-b_1 * (ii+1), \\ \quad N-1) \wedge ii \geq 0 \\ j : b_2 * jj + i + 1 \leq j \leq min(b_2 * (jj+1) + 1, \\ \quad N-1) \wedge jj \geq 0 \\ k : \begin{cases} s1 : b_3 * kk \leq k \leq min(b_3 * (kk+1) - 1, \\ \quad j - i - 1) \wedge kk \geq 0 \\ s2 : k = 0. \end{cases} \end{cases}$$

$TILE\_LT$ ($TILE\_GT$) is the union of all the tiles whose identifiers are lexicographically less (greater) than that of $TILE(\boldsymbol{II}, \boldsymbol{B})$:

$TILE\_LT\ (GT) = \{[\boldsymbol{I}] \mid \exists\ \boldsymbol{II'}$ s. t. $\boldsymbol{II'} \prec (\succ)\ \boldsymbol{II} \wedge \boldsymbol{II} \geq 0 \wedge \boldsymbol{B}^*\boldsymbol{II} + \boldsymbol{LB} \leq \boldsymbol{UB} \wedge \boldsymbol{II'} \geq 0$ and $\boldsymbol{B}^*\boldsymbol{II'} + \boldsymbol{LB} \leq \boldsymbol{UB} \wedge \boldsymbol{I}$ in $TILE(\boldsymbol{II'}, \boldsymbol{B})\}$.

For calculating exact relation R^+, where R is the union of all dependence relations extracted for the Nussinov loop nest, we apply the algorithm presented in paper [15]. Next, we calculate the following set

$$TILE\_ITR = TILE - R^{+}(TILE\_GT),$$

which does not include any invalid dependence target, i.e., it does not include any dependence target whose source is within set *TILE_GT.*

The following set

$$TVLD\_LT = (R^{+}(TILE\_ITR) \cap TILE\_LT) - R^{+}(TILE\_GT)$$

includes all the iterations that i) belong to the tiles whose identifiers are lexicographically less than that of set *TILE_ITR*, ii) are the targets of the dependences whose sources are contained in set *TILE_ITR*, and iii) are not any target of a dependence whose source belong to set *TILE_GT*. Target tiles are defined by the following set

$$TILE\_VLD = TILE\_ITR \cup TVLD\_LT.$$

Next, we form set *TILE_VLD_EXT* by means of inserting into the first positions of the tuple of set *TILE_VLD* elements of vector $\boldsymbol{II}$: $ii_1, ii_2, ..., ii_d$. Nonparametric tiled code is generated by means of applying any code generator allowing for scanning elements of set *TILE_VLD_EXT* in lexicographic order, for example, isl AST [16].

In paper [4], we discuss parallelization of ISS based fixed tiled code by means of loop skewing which honors all dependences among generated tiles.

Assumption about good original tile size and tile dimension

The most important step in generating target ISS based tiled code is defining an original tile size and dimension to form set *TILE* according to the approach presented in paper [4]. They impact serial and parallel code locality and performance. It worth noting that in general, target tiles represented with set *TILE_VLD* are different from original rectangular ones defined with set *TILE*. Target tiles can be parametric non-rectangular ones, i.e., the number of statement instances within such tiles depends on parametric upper loop index bounds.

For parametric tiles, it does not guarantee that the data size per a tile is smaller than the capacity of cache, this leads to decreasing code locality. The number of target parametric tiles and the percentage of the iteration space, occupied by them, depend on an original tile size. So, we strive to choose such original tile size which minimizes the percentage of the iteration sub-space occupied with target parametric tiles. Let us note that if for a given loop nest statement, the set $(R^{+}(TILE\_GT) \cap TILE)$ is empty, this means that for this statement, every target tile is the same as the corresponding original one, i.e., target parametric tiles are absent, so we have a good tiling scheme.

An additional file presents sets $(R^{+}(TILE\_GT) \cap TILE)$ for $s1$ when $B = [7, 79, 133]$ and $B = [1, 79, 133]$, respectively [see Additional file 1]. The set $(R^{+}(TILE\_GT) \cap TILE)$ for statement $s2$ is empty.

Scrutinizing the constraints of the set $(R^{+}(TILE\_GT) \cap TILE)$ for statement $s1$ when $B = [7, 79, 133]$ allows us to conclude that most target tiles are different from original ones and they are non-rectangular. For many target tiles, the data size per a target tile can be greater than the cache capacity of a multi-core platform used by us for carrying out experiments (for details, see the next section). So, the 3-D tiling scheme for ISS based tiled code is not desired.

When we tile only the two inner loops, i.e., $B = [1, 79, 133]$, we can derive the following conclusions. A value of parameter $b3$ has the most impact on the percentage of statement instances within non-corrected (rectangular) target tiles because it influences two loop indexes: j and k. For example, if the constraint $N - ii + b2 * jj <= j <= 78 + N - ii + b2 * jj$ or $k > b2 * jj$ is not satisfied, statement instances defined with vector $(i, j, k)^T$, where j, k do not satisfy the above constraints, are all within rectangular tiles. Analyzing the constraints above, we may conclude that increasing the value of $b2$ increases the percentage of instances of statement $s1$ included in non-corrected target tiles. On the other hand, increasing this value leads to increasing data per a target tile and reducing parallelism degree. So, there exist "the golden mean" of $b2$, which maximizes target ISS based tiled code performance.

The value of $b3$ influences only one loop index, k, in the following constraints of the set $(R^{+}(TILE\_GT) \cap TILE)$: $k >= b3 * kk$ and $k <= b3 - 1 + b3 * kk$. Increasing the value of $b3$ increases the percentage of instances of statement $s1$ included in non-corrected target tiles. On the other hand, increasing this value leads to increasing the stride between cache lines which are referenced at each loop nest iteration (see Listing 1), this can dramatically reduce data reuse. So, a value of $b3$ cannot be large.

Summing up, we may expect that good original tiles are formed with the following matrix $B = [1, b_2, b_3]$ and $b_2 > b_3$, i.e., when we tile only the two inner loops. This assumption is confirmed by means of the results of our experimental study presented in the next section.

Listing 2 Parallel ISS based parametric 3-D tiled code of Nussinov's algorithm.

```
for( c1 = 0; c1 <= floord((b1+b2) * N - (2*b1+b2), b1*b2); c1 += 1)
#pragma omp parallel for
  for( c3 = max(0, c1 - (N + (b1-1)) / b1 + 1); c3 <= min((N - 2) / b2, (b1 * c1 + b1-2) /
      ↳ / (b1+b2)); c3 += 1){
    for( c5 = 0; c5 <= b2 * c3 / b3; c5 += 1)
      for( c7 = max(-N + b1 * c1 - b1 * c3 + 1, -N + b2 * c3 + 2); c7 <= min(0, -N + b1 *
          ↳ c1 - b1 * c3 + b1); c7 += 1) {
        if (N + b1 * c3 + c7 >= b1 * c1 + 2) {
          for( c11 = b3 * c5; c11 <= min(b2 * c3, b3 * c5 + b3-1); c11 += 1)
            S[(-c7)][(b2*c3-c7+1)] = max(S[(-c7)][c11+(-c7)] +
                ↳ S[c11+(-c7)+1][(b2*c3-c7+1)] , S[(-c7)][(b2*c3-c7+1)]);
        } else
          for( c9 = N - b1 * c1 + (b1+b2) * c3; c9 <= min(N - 1, N - b1 * c1 + (b1+b2) *
              ↳ c3 + b2-1); c9 += 1)
            for( c11 = b3 * c5; c11 <= min(b2 * c3, b3 * c5 + b3-1); c11 += 1)
              S[(N-b1*c1+b1*c3-1)][c9] = max(S[(N-b1*c1+b1*c3-1)][c11+(N-b1*c1+b1*c3-1)]
                  ↳ + S[c11+(N-b1*c1+b1*c3-1)+1][c9] , S[(N-b1*c1+b1*c3-1)][c9]);

      }
  for( c7 = max(-N + b1 * c1 - b1 * c3 + 1, -N + b2 * c3 + 2); c7 <= min(0, -N + b1 * c1
      ↳ - b1 * c3 + b1); c7 += 1)
   for( c9 = b2 * c3 - c7 + 1; c9 <= min(N - 1, b2 * c3 - c7 + b2); c9 += 1)
     for( c10 = max(0, b2 * c3 - c7 - c9 + 2); c10 <= 1; c10 += 1) {
       if (c10 == 1) {
         S[(-c7)][c9] = max(S[(-c7)][c9] , S[(-c7)+1][c9-1] + sigma(-c7, c9);
       } else {
         if (N + b1 * c3 + c7 >= b1 * c1 + 2)
           for( c11 = 0; c11 <= b2 * c3; c11 += 1)
             S[(-c7)][c9] = max(S[(-c7)][c11+(-c7)] + S[c11+(-c7)+1][c9] , S[(-c7)][c9]);
         for( c11 = b2 * c3 + 1; c11 < c7 + c9; c11 += 1)
           S[(-c7)][c9] = max(S[(-c7)][c11+(-c7)] + S[c11+(-c7)+1][c9] , S[(-c7)][c9]);
       }
    }
  }
```

ISS based parametric tiled code construction

To improve locality of tiled code, we use a model known as tile size selection (TSS) which can be classified into model-driven empirical search based. It is used to characterize and prune the space of good tile sizes. For each tile size in the pruned search space, a version of the program is generated and run on the target architecture, and the tile size with the least execution time is selected [17].

To apply TSS, we first form parametric 3-D tiled code to avoid generation and compilation of a separate code each time when tile size is changed.

For this purpose, applying our source-to-source optimizing compiler TRACO [18], we generate two non-parametric tiled codes for different values of elements of matrix $\boldsymbol{B} = [b_1, b_2, b_3]$ according to the technique presented in our paper [4]. We choose those values to be prime numbers to avoid generation of simplified non-parametric tiled code. We strive to generate tiled code whose structure is the same regardless of values of elements of matrix $\boldsymbol{B} = [b_1, b_2, b_3]$. Next using those codes, we construct parametric tiled code.

An additional file presents generated tiled codes where *for* loops shown in violet correspond to $\boldsymbol{B}_1 = [23, 47, 113]$, while *for* loops shown in red state for $\boldsymbol{B}_2 = [37, 79, 167]$, [see Additional file 2]. Applying the way, presented in our paper [4], we prove that those codes are valid.

Analyzing those generated codes, we may conclude that i) their structures are the same, only integer factors, present in the same code position, are different; ii) there exist the following linear expressions defining the init-statement, condition, and iteration expression of *for* loops: $b1 + b2$, $b_2, b_2 - 1, b_2 + 1, b_3, b_3 - 1$; iii) there exists the non-linear expression of the form $b1 * b2$.

Taking into account the above conclusions, we form the following general linear model which is valid for each integer factor, say y, present in the expressions of tiled loop nest:

$y = a_0 * b_0 + a_1 * b_2 + a_1 * b_1 + a_2 * b_2 + a_3 * b_3 + a_4$, where a_i, $i = 0, .., 4$, are unknown integer coefficients, $b_0 = b_1 * b_2$.

Let us note that we replaced the non-liner expression $b_1 * b_2$ with the linear one b_0.

We use the iscc calculator [19] to find unknown coefficients $a_i, i = 0, 1, 2, 3$, in the above model as follows.

For each pair of values y_1, y_2, which appear in the same code position of the two generated nonparametric codes, we form a system of equations as follows

$$\begin{cases} y_1 = a_0 * b_{01} + a_1 * b_{21} + a_1 * b_{11} + a_2 * b_{21} + a_3 * b_{31} + a_{41}, \\ y_2 = a_0 * b_{02} + a_1 * b_{22} + a_1 * b_{12} + a_2 * b_{22} + a_3 * b_{32} + a_{42}, \end{cases}$$

Table 1 Finding integer coefficients $a_i, i = 0 \ldots 4$, of the model $y = a_0 * b_0 + a_1 * b_2 + a_1 * b_1 + a_2 * b_2 + a_3 * b_3 + a_4$, where b_0=$b1$*$b2$

y_1 $b_{1,2,3}$=[23,47,113]	y_2 $b_{1,2,3}$=[37,79,167]	a_0	a_1	a_2	a_3	a_4	Formula	Lines, [see Additional file 2]
70	116	0	1	1	0	0	b1+b2	6, 10, 56
93	153	0	2	1	0	0	2*b1+b2	6
1081	2923	1	0	0	0	0	b1*b2	6
22	36	0	1	0	0	-1	b1-1	10
23	37	0	1	0	0	0	b1	10, 15, 43, 46, 56, 62
21	35	0	0	1	0	-2	b1-2	10
113	167	0	0	0	1	0	b3	40, 49, 59
112	166	0	0	0	1	-1	b3-1	49, 59
46	78	0	0	1	0	-1	b2-1	56
47	79	0	0	1	0	0	b2	10, 15, 21, 30, 34, 40, 43, 49, 52, 59

where $b_{ij}, j = 1, 2$, are particular values of $b_i, i = 1, 2, 3$, for the first ($j = 1$) and second ($j = 2$) nonparametric codes, and apply the iscc calculator to resolve that system. It is worth noting that for each pair of y_1, y_2, the general model is simplified so that the resulting system includes only at most two unknowns, the reminding ones are absent.

For example, in the codes presented in [see Additional file 2] in line 6, we have in the same code position integers 93 and 153. We build the following set according to the iscc calculator syntax [19] $\{[a1, a2] : 23 * a1 + 47 * a2 = 93 \wedge 37 * a1 + 79 * a2 = 153\}$.

The constraints of this set are the two linear equations with the two unknowns (a_1, a_2), they obtained from the general model. The iscc calculator returns the following solution $\{[2, 1]\}$, i.e., $a1 = 2, a2 = 1$, and $a0 = a3 = a4 = 0$. Hence, in the parametric code in line 6, we insert the expression $2 * b1 + b2$.

Table 1 presents all solutions for integers available in the examined nonparametric codes. Using those solutions, we form the parametric code presented in Listing 2. In [see Additional file 2], that code is presented with dark lines.

For so obtained parametric code, inter-tile dependences are described with non-affine expressions, so we cannot prove its validity applying the way presented in paper [4]. However, we seek for the best tile size using the previously mentioned TSS technique, which envisages running tiled code for particular fixed values of $b_i, i = 1, 2, 3$. So before running each fixed tiled code, we are able to check its validity applying the way presented in paper [4] because all inter-tile dependences for such a code are affine.

Results and discussion

To carry out experiments, we have used a machine with an Intel Xeon processor E5-2699 v3 (2.3 Ghz in base and 3.6 Ghz in turbo, 18 cores/36 threads, 576 KB L1 Cache for code and data separately, 4.5 MB L2 Cache and 45MB L3 Cache) and 128 GB RAM. All programs were compiled by means of the Intel C++ Compiler (*icc* 15.0.2) with the -O3 flag of optimization. To implement multi-threaded parallel processing, the OpenMP programming interface [20] was chosen.

We experimented with randomly generated RNA strands [1] of length 2200 and 5000, the size of the average and longest human mRNA, respectively. We examined

Table 2 Execution time (in seconds) of serial ISS based tiled code for some tile sizes, N =2200

No.	b1	b2	b3	Time
1	1	128	16	3.2760
2	1	128	8	3.2910
3	1	150	8	3.3264
4	1	128	12	3.3506
5	1	96	16	3.3602
6	1	128	24	3.3913
7	1	150	12	3.4042
8	1	128	6	3.4247
9	1	96	8	3.4357
10	1	200	16	3.4645
...	...	...	...	...
467	2	150	16	6.6576
...	...	...	...	...
Execution time of the original code:				12.2802
7985	400	1	1	12.2872
7986	400	48	1	12.3162
...	...	...	...	...
8000	1	1	1	24.8607

also longer strands (up to 10000) to illustrate benefits of tiling the innermost loop nest.

We considered 20 possible tile sizes along each dimension from the set {1, 2, 4, 6, 8, 12, 16, 24, 32, 40, 48, 64, 96, 128, 150, 200, 256, 300, 400, 512}. This leads to the search space including 20^3 = 8000 possible tile sizes.

To carry out experiments, we wrote a script which automatically fulfills the following tasks: i) chooses one tile size from the search space (values of $b_i, i = 1, 2, 3$), ii) checks the validity of the tiled code with the chosen tile size according to the way presented in paper [4], iii) spawns the tiled code with the chosen tile size, iv) measures execution time, v) repeats steps i) - iv) for each tile size within the search space and collects all execution times. It worth noting that parametric code compilation runs only one time that greatly reduces search time.

Table 2 presents execution time of serial ISS based tiled code for some tile sizes. The execution time of the original (untiled) loop nest is 12.28 seconds. The results show that tiling of the two innermost loops allows for reaching minimal execution time of 3.276 seconds, this results in the maximal speed-up of 3.7. Under speed-up we mean the ratio of original program execution time to that of tiled one. Tiling the outermost loop allows us to reduce time execution to only 6.65 seconds. It is worth noting that only 15 tile sizes in the examined search space lead to greater execution time than that of the original program (see the last lines in Table 2).

Figure 1 depicts execution times of serial ISS based tiled code for the four tile sizes of the outermost loop. As we can see, choosing b_1=1 leads to the maximal tiled code performance. The explanation of this fact is presented in the previous sub-section. For this code, the best tile size within the examined search space is [1,128,16].

We carried out search of the best tile size in the same search space for multi-core tiled code with the bigger problem size, N=5000, and presented execution times in Table 3. For 32 threads, we observed super-linear tiled code speed-up of 112.9 for the tile size of $[1 \times 96 \times 8]$. The reason of super-liner speed-up is the cache affect resulting from the different memory hierarchies of the modern parallel computer used for carrying out experiments.

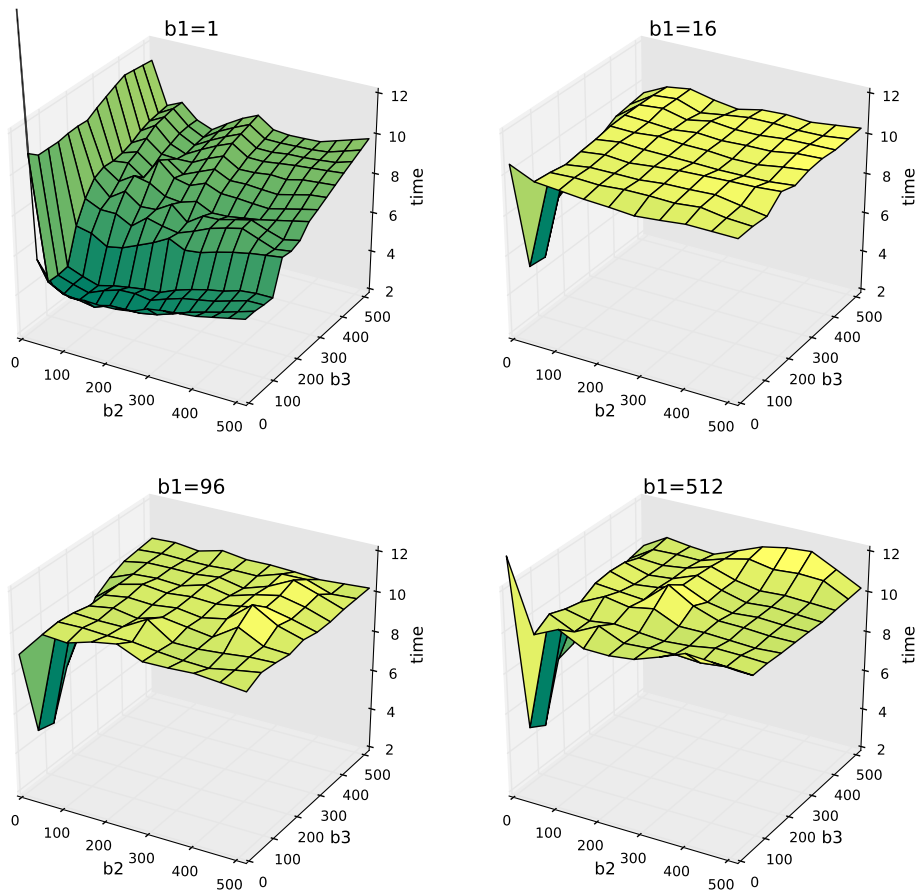

Fig. 1 Execution time (in seconds) of serial ISS based tiled code, N = 2200, run on Intel Xeon E5-2699 v3. Results show that the maximal performance of serial ISS based tiled code is achieved when the outermost loop remains untiled (b_1=1)

Table 3 Execution time (in seconds) of parallel ISS based tiled code for some tile sizes, $N = 5000$, 32 threads used

No.	b1	b2	b3	Time
1	1	96	8	7.8751
2	1	150	12	8.0246
3	1	96	12	8.1903
4	1	128	12	8.1952
5	1	128	16	8.2199
6	1	128	6	8.2816
7	1	150	16	8.2831
8	1	50	8	8.3449
9	1	128	8	8.3841
10	1	96	6	8.4597
...	...	...	...	...
143	2	6	300	10.4351
...	...	...	...	...
Execution time of the original code:				334.3200
7993	2	1	2	335.3765
7994	1	2	2	411.5945
...	...	...	...	...
8000	1	2	1	889.6510

Increasing the number of processors leads to increasing the size of accumulated caches from different processors. With the larger accumulated cache size, more or even all of the working data can fit into caches and memory access time reduces dramatically, which this considerably improve code locality.

Obtained results show how much important is tiling of the innermost loop. To our best knowledge, such a tiling is not possible by means of optimizing compilers based on affine transformations. For example, the-state-of-the-art PluTo compiler (version 0.11.4) fails to tile the innermost loop of the examined program. The interesting fact is that the best code performance is achieved when the outermost loop nest remains untiled, tiling only the two innermost loops allows us to achieve better tiled code locality for the platform chosen for carrying out experiments. It is worth also noting that for the best tile size, the value of b_2 has to be roughly tenfold bigger than that of b_3. The explanations of those facts are given in the previous section.

The results in Table 4, graphically presented in Fig. 2, demonstrate that our generated tiled code is scalable, i.e., increasing the number of threads increases code speed-up.

We compared the performance of ISS based tiled code with that of the manual parallel and cache efficient implementations [21, 22] of Nussinov's RNA folding presented in Listing 3.

Chang et al. [21] modified Nussinov's recurrences equations to simplify parallelization for multi-core architectures. RNA folding starts with initializing elements $S(i,i)$ of the main diagonal of Nussinov's matrix S and elements $S(i,i+1)$ of the diagonal just above the main one, then elements of the remaining diagonals in the order $S(i,i+2)\ldots S(i,i+N-1)$ are calculated. All parallel threads synchronize before moving to the next diagonal.

Listing 3 Nussinov's RNA folding implementations with Chang and Li modifications [22].

```
1  #pragma omp parallel for
2  for(i=0; i<=N-1; i++)
3   S[i][i] = 0;
4
5  #pragma omp parallel for
6  for(i=0; i<=N-2; i++)
7   S[i][i+1] = 0;
8
9  for(diag=2; diag<=N-1; diag++){
10 #pragma omp parallel for private(_max, t)
11  for(row=0; row<=N-diag-1; row++){
12    col = diag + row;
13    _max = S[row+1][col-1]+sigma(row,col);
14    for(k=row; k <=col-1; k++){
15 #ifdef CHANG
16      t = S[row][k] + S[col][k+1];
17 #endif
18 #ifdef LI
19      t = S[row][k] + S[k+1][col];
20 #endif
21      _max = max(_max, t);
22    }
23    S[row][col] = _max;
24 #ifdef LI
25    S[col][row] = _max;
26 #endif
27  }
28 }
```

Table 4 Execution time (in seconds) of the Nussinov RNA folding codes for $N = 5000$ and different numbers of threads used

Threads	Original	Chang	Li	PluTo [8 × 8 × 1]	ISS [2 × 6 × 300]	ISS [1 × 96 × 8]
1	334.32	382.46	81.54	238.66	221.11	54.66
2		198.12	37.96	164.66	120.90	30.83
4		100.17	20.19	90.16	67.74	18.29
8		53.07	13.62	46.01	35.29	10.67
16		28.74	10.50	25.84	19.06	8.22
32		16.55	9.75	13.94	10.65	7.82

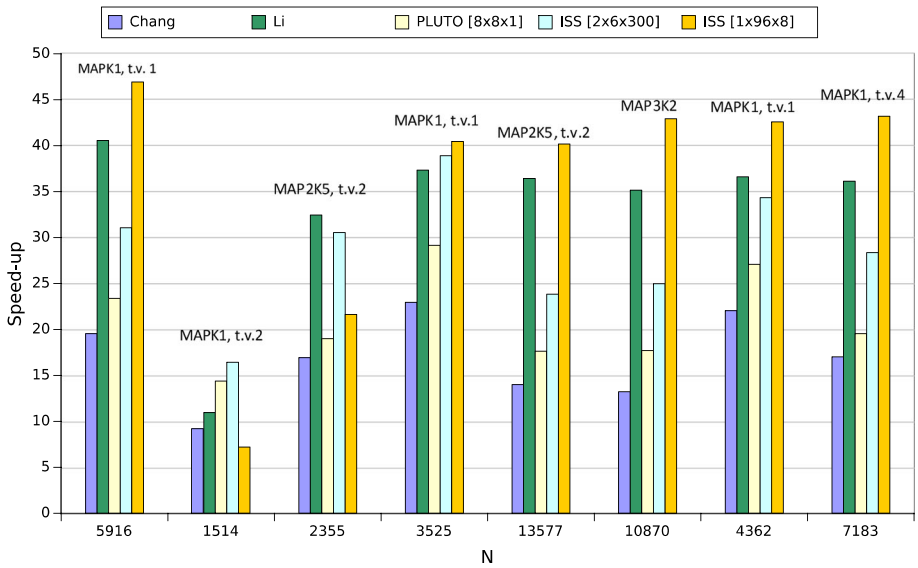

Fig. 2 Speed-up of parallel codes for Nussinov's matrix size of 5000 run on Intel Xeon E5-2699 v3. The horizontal coordinate represents the number of threads, the vertical one shows the speed-up of the examined codes

Li et al. [22] suggested a cache efficient version of Chang's code by using the lower triangle of matrix S to store the transpose of the computed values in the upper triangle of S [22]. They store $S[row][k] + S[k+1][col]$ to variable t (line 19 instead of Chang's line 16) and additionally store the value of $\_max$ to $S[row][col]$ at the end of the loop body (line 25). Values of $S[k+1][col]$ locate in the same column but values of $S[col][k+1]$ locate in the same row, for $row \leq k < col$. Li's modifications accelerate rapidly code execution because reading values in a row is more cache efficient than reading values in a column [22].

Results in Table 4 show that our tiled code implementing Nussinov's algorithm with the tile size $[1 \times 96 \times 8]$ outperforms the implementations of Chang and Li (see Listing 3) for each examined number of threads (from 1 to 32) when N=5000.

This table includes also execution times of tiled code generated with the PluTo compiler, which tiles the two outermost loops[2]. The tile size $[8 \times 8 \times 1]$ was chosen from many different tile sizes, examined by us, as one exposing the highest code performance. Those times are smaller than those achieved with Chang's code. The cache efficient code proposed by Li et al. outperforms PluTo code and our 3-D tiled code. Only tiling of the two innermost loops allows us to achieve higher speed-up than that of Li's implementation. Speed-up of the examined programs is depicted in Fig. 2.

Furthermore, we studied code performance for different problem sizes defined as an RNA strand length, which is an important characteristic of Nussinov's folding. We examined eight mRNAs of homo sapiens mitogen-activated protein kinase (MAPk) from the NCBI database[3]. Code execution times are presented in Table 5 while corresponding speed-up is depicted in Fig. 3. We observe that our code demonstrates higher speed-up than that of the reminding examined codes when the length

Table 5 Execution time (in seconds) of the Nussinov RNA folding codes for 32 threads and different lengths of RNA strands. mRNAs acquired from the NCBI database

mRNA definition	Lenght	Serial time	Chang	Li	PluTo [8 × 8 × 1]	ISS [2 × 6 × 300]	ISS [1 × 96 × 8]
MAPK1, trans. var. 1	5916	606,32	31,05	14,95	25,94	19,52	12,92
MAPK1, trans. var 2	1514	2,30	0,25	0,21	0,16	0,14	0,32
MAP2K5, trans. var 2	2355	15,57	0,92	0,48	0,82	0,51	0,72
MAP2K7, trans. var. 1	3525	102,32	4,46	2,74	3,51	2,63	2,53
MAP2K6, trans. var. 2	13577	6127,09	437,60	168,25	347,78	257,00	152,54
MAP3K2	10870	3033,73	229,53	86,30	171,54	121,44	70,68
MAP4K3, trans. var 1	4362	221,82	10,07	6,06	8,19	6,46	5,21
MAP4K4, trans. var. 4	7183	926,43	54,43	25,65	47,40	32,67	21,45

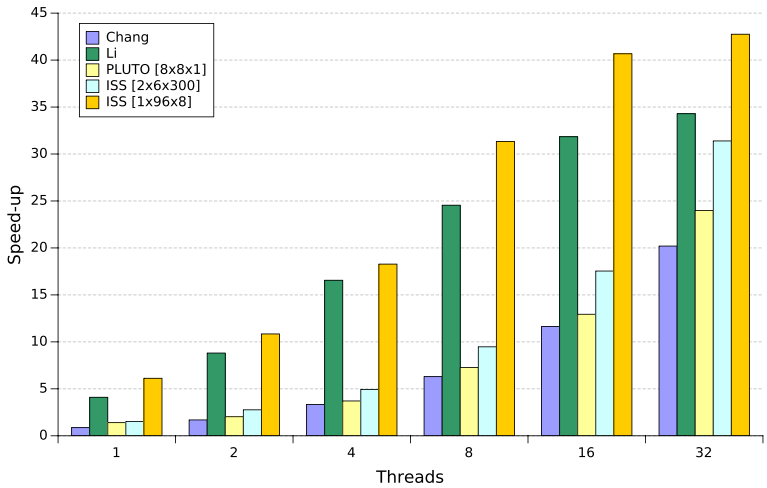

Fig. 3 Speed-up of parallel codes run on Intel Xeon E5-2699 v3, 32 threads used. The horizontal coordinate represents Nussinov's matrix size, the vertical one shows the speed-up of the studied codes. mRNAs acquired from the NCBI database

of RNA strands is bigger than 2500. For short sequences (less than 2500) and 32 threads, related codes are faster (from 0,1 to 0,3 second per one strand) than ours. However, for short sequences, computation time is less than one second per one strand. The power of the presented approach is noticeable for longer strands, for example, our code for MAP2K6 variant 2 demonstrates 16 seconds time benefit per one strand against cache efficient Li's code.

The performance improvement of the code generated with the presented technique against that of Li's code for longer sequences is reached due to i) application of a tiling technique, which allows for increasing parallel code coarseness and locality, ii) choice of the optimal original tile size in the defined search space. All those factors together lead to significant improvement in code performance.

Summing up, we may conclude that the efficiency of cache reuse provided with ISS based tiled code becomes a dominant factor in achieving high code performance despite code complexity. Although our tiled code is more complex than the examined ones, choosing the best original tile size allows for achieving higher performance in comparison with the related examined codes on the multi-core machine used for experiments.

Conclusion

In this paper, we presented an approach which allows us to choose in a given search space the best original tile size and tile dimension for generation of serial and parallel ISS based tiled codes implementing Nissinov's RNA folding. Those codes are generated using the transitive closure of dependence graphs – the transitive dependences are applied to the statement instances of interest to produce valid tiles. Such a technique is within the well-known iteration space slicing framework.

Analyzing the constraints of a set representing valid target tiles, we make an assumption about good original tile size and tile dimension and confirm this assumption with carrying out experiments. The key step of this approach is constructing parallel parametric code, where variables defining tile size are parameters. The usage of parametric code allows us to compile target code only one time that significantly reduces search time.

The experimental study allows us to conclude that i) tiling the two innermost loops is the best tiling scheme for ISS based tiled code, i.e., the outermost loop has to be untiled; ii) the size of the second dimension of an original tile must be roughly tenfold bigger than the size of the third one.

Our implementation of Nussinov's algorithm improves code locality and outperforms the serial original code by a factor of 3.7. We demonstrated super-linear speed-up of 112.9 for parallel code run with 32 threads. The tuned tiled code is more cache efficient than the closely related implementations of Li and Chang when the length of RNA strands is bigger than 2500 for the studied multi-core machine.

Under Nussinov's algorithm conditions, the problem of folding a nucleotide sequence into a structure with minimal free energy becomes a simpler problem of finding a structure with the maximum number of base pairs [1]. Zuker et al. [23] refined Nussinov's algorithm by using a thermodynamic energy minimization model, which produces more accurate results at the expense of greater computational complexity, but code implementing Zuker's algorithm is affine. This allows us to apply the approach presented in this paper to that

code. In future, we intend to apply our tiling strategies to generate parallel code implementing Zuker's algorithm.

In future, we plan to engage heuristics and artificial intelligence methods in the tile size selection technique to reduce the search time of the best tile size. Furthermore, we plan to adopt the presented approach for multi- and many-core graphic cards using popular parallel processing interfaces.

Endnotes

[1] In paper [4], we demonstrate that tiled code performance does not change based on strings themselves, but it depends on the size of a string.

[2] Wonnacot et al. widely demonstrated the weakness of tiling the two outermost loops for Nussinov's algorithm in paper [12].

[3] https://www.ncbi.nlm.nih.gov/

Additional files

Additional file 1: Set $R^{+}(TILE\_GT) \cap TILE$. Presented in the ISL format. (PDF 4 kb)

Additional file 2: Construction of parametric code. Construction of parallel parametric 3-D-tiled code implementing Nussinov's algorithm using two nonparameteric codes. (PDF 22 kb)

Abbreviations

AST: Abstract syntax tree; ATF: Affine transformation framework; ISCC: Integer set counting calculator; ISS: Iteration space slicing; TSS: Tile size selection

Acknowledgements

Not applicable.

Funding

No specific funding was received for this study.

Availability of data and materials

Our compiler is available at the website http://traco.sourceforge.net. Experimental results and source codes are available at the TRACO repository https://sourceforge.net/p/traco/.

Authors' contributions

MP proposed the main concept of the presented technique, implemented it in the TRACO optimizing compiler, and carried out the experimental study. WB checked the correctness of the presented technique, participated in its implementation and the analysis of the results of the experimental study. Both 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: 1 August 2017 Accepted: 2 January 2018
Published online: 15 January 2018

References

1. Nussinov R, Pieczenik G, Griggs JR, Kleitman DJ. Algorithms for Loop Matchings. SIAM J Appl Math. 1978;35(1):68–82.
2. Mullapudi RT, Bondhugula U. Tiling for dynamic scheduling. In: Rajopadhye S, Verdoolaege S, editors. Proceedings of the 4th International Workshop on Polyhedral Compilation Techniques (IMPACT'14). Vienna: 2014. http://impact.gforge.inria.fr/impact2014/papers/impact2014-mullapudi.pdf.
3. Iooss G, Rajopadhye S, Alias C, Zou Y. Mono-parametric Tiling is a Polyhedral Transformation. [Research Report] RR-8802, INRIA Grenoble - Rhône-Alpes; CNRS; 2015. p. 40. https://hal.inria.fr/hal-01219452/document.
4. 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.
5. Pugh W, Rosser E. Iteration space slicing for locality. In: Gao GR, Pollock LL, Cavazos J, Xiaoming L, editors. LCPC. Lecture Notes in Computer Science, vol. 1863. La Jolla: Springer; 1999. p. 164–84.
6. Bondhugula U, Hartono A, Ramanujam J, Sadayappan P. A practical automatic polyhedral parallelizer and locality optimizer. SIGPLAN Not. 2008;43(6):101–13. https://doi.org/10.1145/1379022.1375595.
7. Griebl M. Automatic Parallelization of Loop Programs for Distributed Memory Architectures: University of Passau; 2004. Habilitation thesis.
8. Xue J. Loop Tiling for Parallelism. In: The Springer International Series in Engineering and Computer Science, vol. 575. US: Springer; 2000. https://books.google.pl/books?id=DPJNwR2SBF0C.
9. Hartono A, et al. PrimeTile: A Parametric Multi-Level Tiler for Imperfect Loop Nests. In: ACM International Conference on Supercomputing (ICS). New York: 2009.
10. Hartono A, Baskaran MM, Ramanujam J, Sadayappan P. Dyntile: Parametric tiled loop generation for parallel execution on multicore processors. In: 2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS); 2010. p. 1–12. https://doi.org/10.1109/IPDPS.2010.5470459.
11. Baskaran MM, Hartono A, Tavarageri S, Henretty T, Ramanujam J, Sadayappan P. Parameterized tiling revisited. In: Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO '10. New York: ACM; 2010. p. 200–9.
12. Wonnacott D, Jin T, Lake A. Automatic tiling of "mostly-tileable" loop nests. In: IMPACT 2015: 5th International Workshop on Polyhedral Compilation Techniques. Amsterdam: 2015. http://impact.gforge.inria.fr/impact2015/papers/impact2015-wonnacott.pdf.
13. Pugh W, Wonnacott D. In: Banerjee U, Gelernter D, Nicolau A, Padua D, editors. An exact method for analysis of value-based array data dependences. Berlin, Heidelberg: Springer; 1994. p. 546–66.
14. Bielecki W, Palkowski M. Tiling arbitrarily nested loops by means of the transitive closure of dependence graphs. Int J Appl Math Comput Sci (AMCS). 2016;26(4):919–39.
15. Bielecki W, Kraska K, Klimek T. Using basis dependence distance vectors in the modified floyd–warshall algorithm. J Comb Optim. 2015;30(2):253–75.
16. Grosser T, Verdoolaege S, Cohen A. Polyhedral ast generation is more than scanning polyhedra. ACM Trans Program Lang Syst. 2015;37(4):12–11250.
17. Yuki T, Renganarayanan L, Rajopadhye S, Anderson C, Eichenberger AE, O'Brien K. Automatic creation of tile size selection models. In: Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO '10. New York: ACM; 2010. p. 190–9. https://doi.org/10.1145/1772954.1772982.
18. Palkowski M, Bielecki W. TRACO: source-to-source parallelizing compiler. Comput Inform. 2016;35(6):1277–306.
19. Verdoolaege S. Counting affine calculator and applications. In: First International Workshop on Polyhedral Compilation Techniques (IMPACT'11). Charmonix; 2011. http://perso.ens-lyon.fr/christophe.alias/impact2011/impact-05.pdf.
20. OpenMP Architecture Review Board. OpenMP Application Program Interface Version 4.5. 2015. http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf. Accessed 10 Jan 2018.

21. Chang DJ, Kimmer C, Ouyang M. Accelerating the Nussinov RNA folding algorithm with CUDA/GPU. In: The 10th IEEE International Symposium on Signal Processing and Information Technology; 2010. p. 120–5. https://doi.org/10.1109/ISSPIT.2010.5711746.
22. 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.
23. Zuker M, Stiegler P. Optimal computer folding of large rna sequences using thermodynamics and auxiliary information. Nucleic Acids Res. 1981;9(1):133–48.