Communication Papers of the Federated Conference on Computer Science and Information Systems pp. 111–114
DOI: 10.15439/2018F8
ISSN 2300-5963 ACSIS, Vol. 17

Accelerating Minimum Cost Polygon Triangulation Code with the TRACO Compiler

Marek Palkowski, Wlodzimierz Bielecki
West Pomeranian University of Technology in Szczecin
ul. Zolnierska 49, 71-210 Szczecin, Poland
Email: mpalkowski@wi.zut.edu.pl, wbielecki@wi.zut.edu.pl

***Abstract*—In this paper, we present automatic loop tiling and parallelization for the minimum cost polygon triangulation (MCPT) task. For this purpose, we use the authorial source-to-source TRACO compiler. MCPT is a recursive algorithm encountering each subproblem many times in different branches of its recursion tree. The most intensive computing part is a triple nested polyhedral program loop nest filling a cost table using the MCPT recursive. First, the code is tiled by means of the transitive closure of a dependence graph. TRACO allows for tiling of the innermost loop nest that is not possible by means of other closely related compilers. We tile only the two innermost loops and apply skewing to serialize the outermost one and parallelize the innermost ones. An experimental study carried out on multi-core computers demonstrates considerable speed-up of tiled code, which overcomes that obtained for code generated with the closely related PLuTo compiler based on the affine transformations framework.**

I. Introduction

The cost of moving data from main memory can be higher than the cost of computation on modern multi-core platforms. This disparity between communication and computation prompts to design algorithms for better locality and parallelism. Loop nest tiling allows for both coarsening code parallelism and improving its locality that leads to increasing parallel code performance. Widely-known tiling techniques based on the polyhedral model[1] use linear or affine transformations of program loop nests [2], [3], [4], [5], [6].

Dynamic programming (DP) is typically applied to optimization problems. In such problems, there can be many possible solutions and we wish to find a solution with the optimal (minimum or maximum) value. Computing intensive DP tasks like minimal cost polygon triangularization can be presented as loop nests within the polyhedral model, however, they involve non-uniform dependences. This limits many optimization techniques such as permutation, diamond tiling [7], or index set splitting [8] to improve cache efficiency.

State-of-the-art automatic optimizing compilers, such as PLuTo [2], have provided empirical confirmation of the success of polyhedral-based optimization. PLuTo and similar optimizing compilers apply the affine transformation framework (ATF), which has demonstrated considerable success in generating high-performance parallel codes in particular for stencils. However, in general, this framework is not able to tile all loops in dynamic programming code [9], [10].

In this paper, we use an alternative approach based on the transitive closure of dependence graphs, which allows us to tile bands of non-permutable loops [11] and extract parallelism when affine transformations miss it [11]. This approach is implemented in the TRACO [12] compiler.

TRACO does not find and use any affine function to transform the loop nest. It is based on the Iteration Space Slicing Framework introduced by Pugh and Rosser [13] and applies the transitive closure of a dependence graph to carry out corrections of original rectangular tiles so that all dependences available in the original loop nest are preserved under the lexicographic order of target tiles. The transitive closure of a graph G is a graph where there is an edge between vertices if they are connected directly or indirectly in the graph G.

In this paper, we show that such a technique enables tiling for all MCPT loops in opposite to affine transformation algorithms implemented in PLuTo. We discuss the performance of parallel tiled MCPT code generated by TRACO and executed on modern multi-core processors and compare it with that of PLuTo tiled code.

II. Minimal cost polygon triangulation

A polygon is a piecewise linear closed curve in the plane. A convex polygon has interior angles that are each strictly less than 180°. A triangulation of a polygon is a set of chords of the polygon that divide the polygon into disjoint triangles (polygons with 3 sides).

In the optimal (polygon) triangulation problem, we are given a convex polygon and a weight function defined on triangles formed by sides and chords. The problem is to find a triangulation that minimizes the sum of the weights of the triangles in the triangulation.

Let cost $w(i,j,k)$ denote the length of the perimeter of $\triangle v_i v_j v_k = |v_i v_j| + |v_j v_k| + |v_k v_i|$. Then minimal cost polygon triangulation is as follows,

$$c[i][j] = \begin{cases} 0 & j < i+2, \\ \max\limits_{i<k<j}(c[i][k] + c(k][j] + w(i,j,k)) & \text{otherwise.} \end{cases} \quad (1)$$

Possible values of $c[i][j]$ fall into two cases. If $j < i+2$ then the polygon with vertices $v_i \ldots v_j$ has fewer than 3 vertices,

[1]The polyhedral model is a mathematical formalism for analyzing, parallelizing, and transforming program loop nests whose all bounds and all conditions are affine expressions in the loop iterators and symbolic constants called parameters [1].

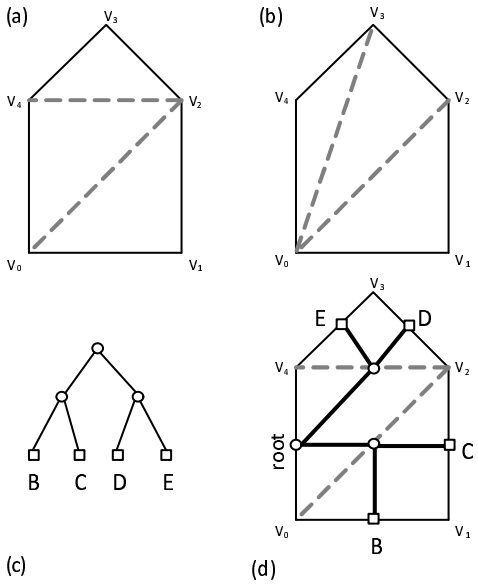

Fig. 1. The example of minimal cost polygon triangulation

and no triangulation is possible, so an appropriate minimum triangulation cost is 0, otherwise there is one or more choices of k where $i < k < j$. To list all triangles, we have to traceback using a history recorded in a separate array of the best vertex to do recursive triangulations at each step.

The memory complexity of the defined cost matrix is $\mathcal{O}(n^2)$. The time complexity of a direct implementation of this algorithm is $\mathcal{O}(n^3)$.

Figures 1a and 1b present two choices of given polygon triangulation (except mirror ones). There is a surprising correspondence between the triangulation of a polygon and the parenthesization of an expression. This correspondence is best explained using trees, see Figure 1c. A full parenthesization of an expression corresponds to a full binary tree, sometimes called the parse tree of an expression. Each leaf of a parse tree is labeled by one of the polygon sides. The root of a tree is a side between the first and last vertices. The parse tree for the parenthesized product is defined with the expression $((BC)(DE))$. The triangulation of the polygon with the parse tree overlaid is depicted in Figure 1d (assuming that triangulation presented in Figure 1a has the minimal cost).

Summing up, minimal cost polygon triangulation corresponds to dynamic programming tasks like chain matrix multiplication or an optimal binary search tree.

III. Loop tiling and parallelization of MCPT code

To find the minimal cost for cell c[i][j], previous cells are scanned in the corresponding row and column. This is typical for dynamic programming tasks such as Nussinov's algorithm [14]. The MCPT algorithm is also within nonserial polyadic dynamic programming (NPDP). The term nonserial polyadic stands for another family of dynamic programming (DP) with nonuniform data dependences (some elements of dependence distance vector are not constant), which is more difficult to be optimized.

Our idea to form valid target tiles is different from that based on affine transformations. First, we apply the transitive closure of the dependence graph representing all the dependences available in the loop nest, to check whether the original tiles are valid. A valid tile with identifier II does not include any dependence destination whose corresponding dependence source belongs to a tile whose identifier is greater than II. If there exist invalid tiles, we correct them with transferring invalid destinations to the tiles including the corresponding sources [11]. It is worth noting that there is no cycle in the corresponding inter-tile dependence graph and any parallelization technique can be applied.

Listing 1 presents polyhedral affine loop nest calculating the cost table of polygon triangulation defined using formula (Eq. 1). It is worth noting that the table is filled in a diagonal fashion, i.e., from diagonal elements to element [0][n-1].

The loop nest can be tiled by both PLuTo and TRACO, however, only TRACO allows us to tile all the three loops of the nest. We discovered empirically that the best tile size is [1x128x16], i.e., the first loop has not be tiled. The second loop is parallel, it does not carry any dependence because cells are scanned in a diagonal fashion [14]. Listing 2 presents the tiled parallel OpenMP code generated by TRACO. The compiler automatically detects that the second loop enumerating tiles does nor carry any dependence.

Such a code cannot be generated by means of PLuTo because it is able to tile only the two outermost loops, the innermost loop remains untiled. For tiled code generated by PLuTo, we empirically discover that the best tile size is [8x8x1], PLuTo code can be found at https://sourceforge.net/p/traco/code/HEAD/tree/trunk/examples/trian.c.

IV. Experimental study

This section presents the results of the comparison of TRACO and PLuTo tiled code performance. To carry out experiments, we have used a computer with the following features: Intel Xeon CPU E5-2699 v2, 3.6GHz, 18 cores, 36 Threads, 45 MB Cache, 16 GB RAM. We examined parallel code performance also using a coprocessor Intel Xeon Phi 7120P (16GB, 1.238 GHz, 61 cores, 30.5 MB Cache). Programs were compiled with the Intel C Compiler (icc 15.0.2) and optimized at the `-O3` level.

Table 1 presents execution times (in seconds) for various numbers of random points of polygon vertexes. Figure 2 depicts the speed-up and efficiency of the tiled programs. Analyzing the results obtained, we may conclude that the TRACO code performance overcomes that of the PLuTo one. For one and two threads, super-linear speed-up is observed. Tiling of the innermost loop allows us to achieve the minimal execution time even without using all threads available on the computer used for experiments. Although PLuTo code seems to be more scalable regarding the number of threads, poor locality limits its speed-up on the modern multi-core machine used for experiments.

Listing 1. Serial loop nest implementing minimum cost polygon triangulation.

```
for (gap = 0; gap < N; gap++){
   for (j = gap; j < N; j++){     // i = j - gap
      if (gap < 2)  // polygon vi...vj has fewer than 3 vertices,
         table[j-gap][j] = 0;
      else{
         table[j-gap][j] = INT_MAX;
         for (k = j-gap+1; k < j; k++){
            table[j-gap][j]  = MIN(table[j-gap][j], table[j-gap][k] +
               ↳ table[k][j] + cost(j-gap,j,k));
} } } }
```

Listing 2. Parallel tiled loop nest implementing minimum cost polygon triangulation.

```
for( c1 = 0; c1 < N; c1 += 1) // tiles of gap (serial)
  #pragma omp parallel for
  for( c3 = 0; c3 <= (N - c1 - 1) / 128; c3 += 1) {  // tiles of j (parallel)
    if (c1 >= 2) {
      for( c4 = 1; c4 <= 2; c4 += 1) {
        if (c4 == 2) {
          for( c5 = 0; c5 <= floord(c1 - 2, 16); c5 += 1)  // tiles of k (serial)
            for( c9 = c1 + 128 * c3; c9 <= min(N-1, c1 + 128 * c3 + 127); c9 += 1)
              for( c11 = -c1 + 16 * c5 + c9 + 1; c11 <= min(c9 - 1, -c1 + 16 *
                 ↳ c5 + c9 + 16); c11 += 1)
                table[c9-c1][c9] = MIN(table[c9-c1][c9], table[c9-c1][c11] +
                   ↳ table[c11][c9] + cost[c9-c1][c9][c11]);
        } else
          for( c9 = c1 + 128 * c3; c9 <= min(N - 1, c1 + 128 * c3 + 127); c9 += 1)
            table[c9-c1][c9] = INT_MAX;
      }
    } else
      for( c9 = c1 + 128 * c3; c9 <= min(N - 1, c1 + 128 * c3 + 127); c9 += 1)
        table[c9-c1][c9] = 0;
  }
```

V. Conclusion

In this paper, we presented the usage of the TRACO compiler, implementing optimizing loop nest algorithms based on the transitive closure of dependence graphs, to the dynamic programming of minimal cost polygon triangulation. Results of experiments demonstrate that the speed-up of parallel tiled code generated by TRACO is higher than that of code generated by means of the state-of-the-art PLuTo compiler. Tiling the innermost loop of the examined loop nest allows us to considerably accelerate NDPD programs.

In future, we plan to study arbitrarily shaped tiling based on transitive closure aimed at generating more flexible code for affine loop nests typical for a code of dynamic programming code.

References

[1] W. Kelly and W. Pugh, "A framework for unifying reordering transformations," Univ. of Maryland Institute for Advanced Computer Studies Report No. UMIACS-TR-92-126.1, College Park, MD, USA, Tech. Rep., 1993.

[2] U. Bondhugula *et al.*, "A practical automatic polyhedral parallelizer and locality optimizer," *SIGPLAN Not.*, vol. 43, no. 6, pp. 101–113, Jun. 2008. doi: 10.1145/1379022.1375595 Http://pluto-compiler.sourceforge.net/.

[3] M. Griebl, "Automatic parallelization of loop programs for distributed memory architectures," 2004.

[4] F. Irigoin and R. Triolet, "Supernode partitioning," in *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL '88. New York, NY, USA: ACM, 1988. doi: 10.1145/73560.73588. ISBN 0-89791-252-7 pp. 319–329.

[5] A. Lim, G. I. Cheong, and M. S. Lam, "An affine partitioning algorithm to maximize parallelism and minimize communication," in *In Proceedings of the 13th ACM SIGARCH International Conference on Supercomputing*. ACM Press, 1999. doi: 10.1145/305138.305197 pp. 228–237.

[6] J. Xue, "On tiling as a loop transformation," 1997.

[7] U. Bondhugula, V. Bandishti, and I. Pananilath, "Diamond tiling: Tiling techniques to maximize parallelism for stencil computations," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 5, pp. 1285–1298, May 2017. doi: 10.1109/tpds.2016.2615094

[8] U. Bondhugula, A. Acharya, and A. Cohen, "The pluto+ algorithm: A practical approach for parallelization and locality optimization of affine loop nests," *ACM Trans. Program. Lang. Syst.*, vol. 38, no. 3, pp. 12:1–12:32, Apr. 2016. doi: 10.1145/2896389

TABLE I
EXECUTION TIME (IN SECONDS) OF THE ORIGINAL, TRACO AND PLUTO TILED CODES IMPLEMENTING MCPT.

N	1 Thread			2 Threads		4 Threads		8 Threads		16 Threads		32 Threads	
	Orig.	TRACO	PLuTo	TRACO	PLuTo	TRACO	PLuTo	TRACO	PLuTo	TRACO	PLuTo	TRACO	PLuTo
1000	2.22	1.17	1.75	0.85	1.30	0.68	1.15	0.67	01.08	0.61	0.98	**0.58**	**0.78**
1500	7.14	3.21	6.42	2.37	4.71	2.25	3.48	**1.97**	2.75	2.16	**2.66**	2.31	2.75
2000	16.78	7.19	15.42	5.58	9.67	**4.61**	7.89	4.75	6.98	4.76	**5.91**	4.78	6.33
2500	34.93	14.14	33.40	**9.97**	20.98	10.73	17.96	10.42	15.14	10.25	13.24	11.03	**11.55**
3000	62.06	24.61	63.54	16.94	42.09	**16.32**	32.31	17.92	28.88	17.73	21.16	17.23	**20.99**
5000	524.04	197.48	465.38	94.37	266.62	83.28	186.24	87.54	137.11	**75.67**	118.73	87.02	**96.29**

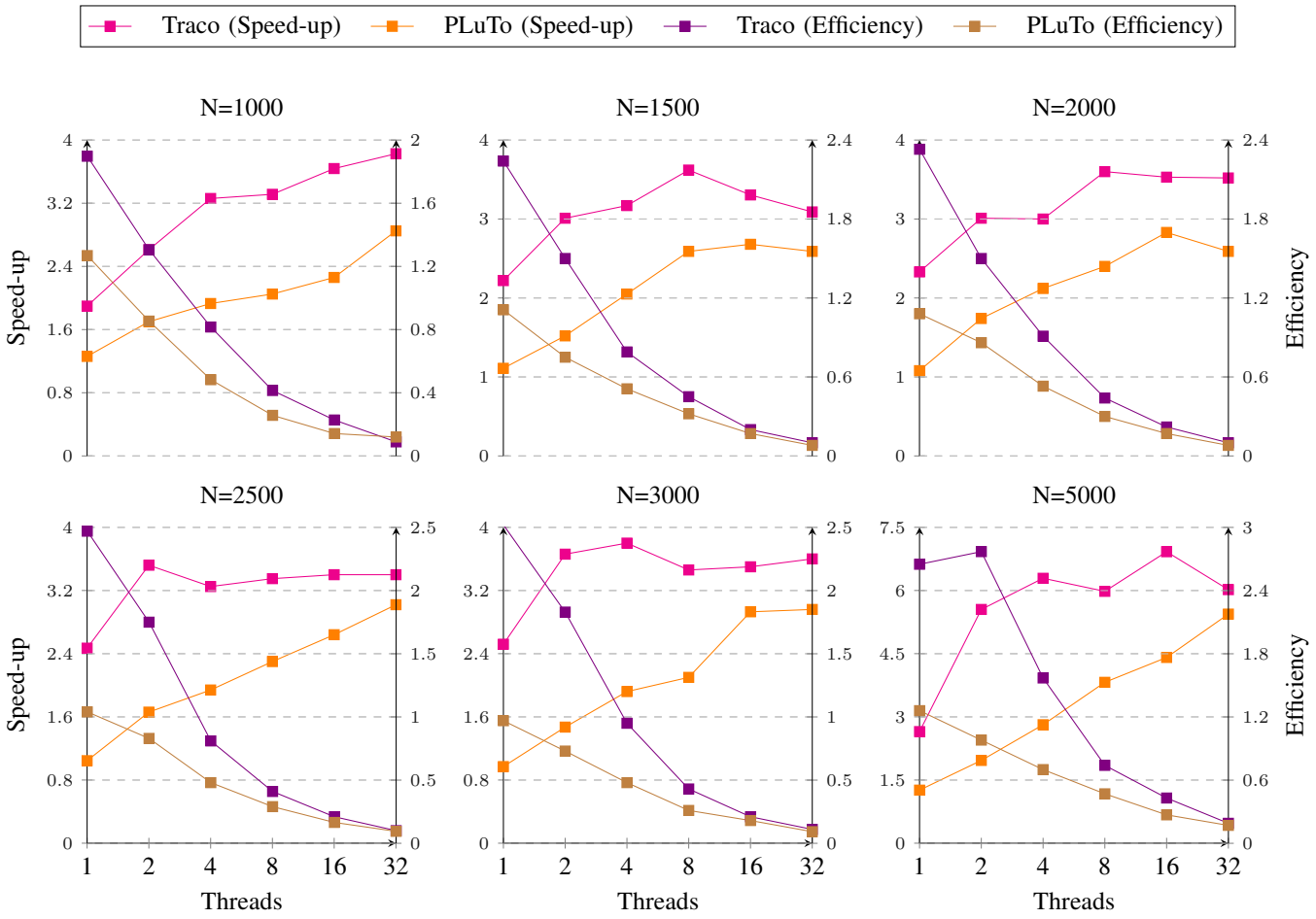

Fig. 2. Speed-up and efficiency of the TRACO and PLuTo codes.

[9] D. G. Wonnacott and M. M. Strout, "On the scalability of loop tiling techniques," in *Proceedings of the 3rd International Workshop on Polyhedral Compilation Techniques (IMPACT)*, January 2013.

[10] R. T. Mullapudi and U. Bondhugula, "Tiling for dynamic scheduling," in *Proceedings of the 4th International Workshop on Polyhedral Compilation Techniques*, Vienna, Austria, Jan. 2014.

[11] W. Bielecki and M. Palkowski, "Tiling of arbitrarily nested loops by means of the transitive closure of dependence graphs," *International Journal of Applied Mathematics and Computer Science (AMCS)*, vol. Vol. 26, no. 4, pp. 919–939, December 2016. doi: 10.1515/amcs-2016-0065

[12] ——, "A parallelizing and optimizing compiler - traco," 2013. [Online]. Available: http://traco.sourceforge.net

[13] W. Pugh and E. Rosser, "Iteration space slicing and its application to communication optimization," in *International Conference on Supercomputing*, 1997. doi: 10.1145/263580.263637 pp. 221–228.

[14] M. Palkowski and W. Bielecki, "Parallel tiled Nussinov RNA folding loop nest generated using both dependence graph transitive closure and loop skewing," *BMC Bioinformatics*, vol. 18, no. 1, p. 290, 2017. doi: 10.1186/s12859-017-1707-8