Proceedings of the Federated Conference on Computer Science and Information Systems pp. 523–526
DOI: 10.15439/2017F19
ISSN 2300-5963 ACSIS, Vol. 11

Optimizing Numerical Code by means of the Transitive Closure of Dependence Graphs

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*—A challenging task in numerical programming modern computer systems is to effectively exploit the parallelism available in the architecture and manage the CPU caches to increase performance. Loop nest tiling allows for both coarsening parallel code and improving code locality. In this paper, we explore a new way to generate tiled code and derive the free schedule of tiles by means of the transitive closure of loop nest dependence graphs. Multi-threaded code executes tiles as soon as their operands are available. To design the approach, loop dependences are presented in the form of tuple relations. Discussed techniques are implemented in the source-to-source TRACO compiler. Experimental study, carried out on multi-core architectures, demonstrates the considerable speed-up of tiled numerical codes generated by the presented approach.**

I. Introduction

ON MODERN architectures, the cost of moving data from main memory can be higher than the cost of computation. This disparity between communication and computation prompts that designing algorithms for better locality and parallelism exploiting even with simple memory models is a challenging task. Loop nest tiling allows for both coarsening parallel code and improving its locality that leads to increasing parallel code performance.

Widely known tiling techniques use the polyhedral model and affine transformations of program loop nests [1], [2], [3], [4], [5]. State-of-the-art automatic parallelizers, such as PLuTo [1], have provided empirical confirmation of the success of polyhedral-based optimization.

Techniques based on the polyhedral model and affine transformations include the following three steps: i) program analysis aimed at translating high level codes with data dependence analysis to their polyhedral representation, ii) program transformation with the aim of improving program locality and/or parallelization, iii) code generation [1].

To implement the second step of the approach mentioned above, PLuTo and similar optimizing compilers apply the affine transformation framework (ATF), which has demonstrated considerable achievement in obtaining high performance parallel codes. However, this framework is not able to parallelize some classes of serial code.

Wonnacott and Strout outlined limitations of tiling transformations that have been released in tools like PLuTo [6]. Techniques involve pipelined execution of tiles, which prevents full concurrency from the start and do not allow full scaling.

Neverthless, there are known some attempts to enhance the power of ATF. In paper [7], tiling for dynamic scheduling is discussed. Wonnacott et al. [8] introduce the definition of mostly-tileable loop nests for which classic tiling is prevented by an asymptotically insignificant number of iterations.

Our research is concerned with alternative approaches that allow us to tile bands of non-permutable loops [9] and find parallelism when affine solutions miss it. These algorithms are implemented in the source-to-source compiler, TRACO[1].

TRACO realizes all the three steps of the approach mentioned above. However, the tool does not find and use any affine function in the second step to transform the loop nest. TRACO is based on the idea of the Iteration Space Slicing Framework introduced by Pugh and Rosser [10] and applies the transitive closure of a program dependence graph to extract independent subspaces in the original loop nest iteration space.

In paper [11], we proposed a technique to find the tile free schedule[2] adopting parallelization based on the power k of relation R, R^k. Unfortunately, when relation R^k cannot be calculated exactly, the value of k in the R^k constraints is usually unbounded and valid code generation is impossible. It is worth to mention that computing exact R^k guarantees computing exact R^+, but not vice versa [12].

In this paper, we show how this limitation can be overcome by means of applying positive transitive closure, R^+, and transitive closure, R^*, (instead of the power k of relation R, R^k) to form the free schedule of valid tiles. The proposed approach generates parallel tiled code even when producing a band of fully permutable loops with ATF is not possible. We present the performance of eight real-life parallel tiled numerical programs generated by TRACO and executed on modern multi-core processors and co-processors.

II. Background

The polyhedral model is a mathematical formalism for analyzing and transforming program loop nests whose all bounds and all conditions are affine expressions in the loop iterators and symbolic constants called parameters [13]. Loop transformations based on transitive closure [9], [10], [14] are mainly focused on representation and manipulation of sets and relations. A set contains integer tuples that satisfy some

[1] traco.sourceforge.net
[2] tiles are executed as soon as it is possible (their operands are available)

Presburger formula built from affine constraints, conjunctions (and, $\wedge$), disjunctions (or, $\vee$), projections (exists, $\exists$) and negations (not, $\neg$). Relations are defined in a similar way, except that the single space is replaced by a pair of spaces separated by the arrow sign "$\rightarrow$", see paper [12].

The considered approach uses an exact dependence analysis [15] which returns dependences in the form of relations. The pairs of input and output spaces represent loop statement instances corresponding to data dependence sources and destinations, respectively.

Basic operations on sets and relations include intersection ($\cap$), union ($\cup$), difference ($-$), composition ($\circ$), domain (dom), range (ran), relation application ($R(S)$). Manual [12] describes the operations in detail.

In the sequential loop nest, the iteration i executes before j if i is *lexicographically less* than j, denoted as

$i \prec j, i.e., i_1 < j_1 \vee \exists k \geq 1 : i_k < j_k \wedge i_t = j_t, for\ t < k.$

The positive transitive closure of a lexicographically forward relation R, R^+, is defined as follows [16]:

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

It describes which vertices e' in a dependence graph (represented by relation R) are connected directly or transitively with vertex e. Transitive closure, R^*, additionally includes the identity relation, $I = \{e \rightarrow e\}$.

An *ultimate dependence source* is a source that is not the destination of another dependence. Set, *UDS*, comprising all ultimate dependence sources, can be found as domain(R) – range(R), where R represents all loop nest dependences.

Let *IS* be a polytope representing the loop nest iteration space while the tuple (IS, E) represents a dependence graph, where E is the set of edges defining dependences. The function $t : IS \rightarrow Z$, which assigns time execution to each loop nest statement instance, is called a valid schedule if it preserves all data dependences: $(\forall x, x' : x, x' \in IS \wedge (x, x') \in T : t(x) < t(x'))$ [17]. The schedule that maps every $x \in IS$ onto the first possible time allowed by the dependences is called the free schedule.

III. Free scheduling algorithm

We use the technique, presented in paper [14], to extract fine-grained parallelism based on the free schedule which represents unique time partitions; statement instances within a time partition are independent. Let us remind the idea of that approach. First, we calculate relation, R', by inserting variables k and k+1 into the first position of the input and output tuples of relation R which is the union of all dependence relations. Variable k defines execution time for each partition including a set of independent statement instances. Next, we find the transitive closure of relation R', R'^*, and form the following relation

$FS = \{[X] \rightarrow [k, Y] : X \in UDS(R) \wedge (k, Y) \in \text{Range}((R')^* \backslash \{[0, X]\}) \wedge \neg(\exists\ k' > k \text{ s.t. } (k', Y) \in \text{Range}(R')^+ \backslash \{[0, X]\})\}$,

where $(R')^* \backslash \{[0, X]\}$ defines the domain of relation R'^* restricted to the set including only ultimate dependences sources (the first time partition); the constraint $\neg(\exists\ k' > k$ s.t. $(k', Y) \in \text{Range}(R')^+ \backslash \{[0, X]\})$ guarantees that partition k includes only those statement instances whose operands are available, and each statement instance belongs to only one time partition [14].

The first element of the tuple of the set Range(*FS*) points out the time of partition execution. Parallel code that visits each element of the set Range(*FS*) in lexicographical order can be obtained by applying any well-known code generator, for example, [18], [19]. The outermost sequential loop of such code scans the values of variable k (representing the time of partition execution) while inner parallel loops scan independent instances of partition k.

IV. The Loop Nest Tiling Algorithm

To improve code locality, we apply loop tiling. In paper [9], we demonstrated how to generate valid tiled code using the transitive closure of dependence graphs. That approach envisages forming the following sets:

- *TILE(**II**, **B**)* includes iterations belonging to a parametric tile: *TILE*(***II***, ***B***) = $\{[\boldsymbol{I}] \mid \boldsymbol{B}^*\boldsymbol{II} + \boldsymbol{LB} \leq \boldsymbol{I} \leq \min(\boldsymbol{B}^*(\boldsymbol{II} + \boldsymbol{1}) + \boldsymbol{LB} - \boldsymbol{1}, \boldsymbol{UB}) \wedge \boldsymbol{II} \geq 0\}$, where vectors ***LB*** and ***UB*** include the lower and upper loop index bounds of the loop nest, respectively; matrix ***B*** defines the size of original tiles; elements of vector ***I*** represent the statement instances contained in the tile whose identifier is ***II***; **1** is the vector whose all elements have value 1,[3]
- *TILE_LT*(GT) are the unions of all the tiles whose identifiers are lexicographically less (greater) than that of *TILE*(***II***, ***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*($\boldsymbol{II}'$, ***B***)$\}$,[4]
- *II_SET* =$\{[\boldsymbol{II}] \mid \boldsymbol{II} \geq 0 \wedge \boldsymbol{B}^*\boldsymbol{II} + \boldsymbol{LB} \leq \boldsymbol{UB}\}$ represents all tile identifiers,
- *TILE_ITR* = *TILE* $-$ R^+(*TILE_GT*) does not include any invalid dependence target, i.e., it does not include any dependence target whose source is within set *TILE_GT*,
- *TVLD_LT* = (R^+(*TILE_ITR*) $\cap$ *TILE_LT*) $-$ R^+(*TILE_GT*) includes all the statement instances 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*,
- *TILE_VLD* = *TILE_ITR* $\cup$ *TVLD_LT* defines target tiles,
- *TILE_VLD_EXT* is built by means of inserting i) into the first positions of the tuple of set *TILE_VLD* elements of vector ***II***: $ii_1, ii_2, ..., ii_d$; ii) into the constraints of set *TILE_VLD* the constraints defining tile identifiers $\boldsymbol{II} \geq 0$ and $\boldsymbol{B}^*\boldsymbol{II} + \boldsymbol{LB} \leq \boldsymbol{UB}$. This set represents valid target tiles. To scan their elements in lexicographic order, we

[3]The notation $x \geq (\leq)\ y$ where x, y are two vectors in $\mathbb{Z}^n$ corresponds to the component-wise inequality, that is, $x \geq (\leq)\ y \iff x_i \geq (\leq)\ y_i$, i=1,2,...,n.

[4]"$\prec$" and "$\succ$" denote the lexicographical relation operators for two vectors,

can appy any code generator, for example, CLooG [18] or the isl AST generator [19].

V. The free schedule of target tiles

The approach, presented in this paper, combines the approaches presented in the two previous sections. We generate valid tiles and next apply the free schedule for those tiles. For this purpose, relation, R_TILE, is computed which describes dependences among generated tiles but ignores dependences within each tile as follows

R_TILE:={[***II***]−>[***JJ***]: ∃ ***I***, ***J*** s.t. ***J*** ∈ *R*(***I***) ∧ (***II, I***) ∈ *TILE_VLD_EXT*(***II***) ∧ (***JJ, J***) ∈ *TILE_VLD_EXT*$_i$(***JJ***)},

where ***II, JJ*** are the vectors representing tile identifiers; vectors ***I***, ***J*** comprise the statement instances belonging to the tiles whose identifiers are ***II, JJ***, respectively.

Next, we calculate relation, *R_TILE*′, by inserting variables *k* and and *k*+1 into the first position of the input and output tuples of relation *R_TILE*, respectively. In the following steps, we calculate the transitive closure of this relation and form set, *UDS_TILE,* including the tile identifiers which are not dependence destinations.

We use sets *R_TILE*′ and *UDS_TILE* to calculate relation, *FS*. Then, we form the free schedule for generated target tiles. Finally, we generate code scanning statement instances within the set Range(*FS*) in lexicographical order.

Algorithm 1 presents the discussed above approach in details. The proof of its correctness is presented in papers [9], [14].

VI. Experimental study

To evaluate the performance of tiled code generated by means of Algorithm 1, we have considered the following eight numerical polyhedral programs[5]:

- *floyd* - Floyd-Warshalls all-pairs shortest-paths from PolyBench/C[6],
- *trmm* - Triangular matrix-multiply from PolyBench/C,
- *k23* - 2-D implicit hydrodynamics fragment from Livermore Loops[7],
- *wz* - WZ factorization: dense, square, non-structured matrix factorization algorithm [20],
- *edge_detect* - 2D-convolution routine to expose edge information from the UTDSP Benchmark suite[8],
- *trisolv* - Triangular solver from PolyBench/C,
- *corcol*, *covcol* - Correlation and Covariance Computations, data-mining programs from PolyBench/C.

The programs *floyd*, *wz*, and *k23* cannot be parallelized by the algorithm based on the power k of relations R, R^k [11] because ISL returns only an approximation of R^k, where k is unbounded that prevents code generation – the number of time partitions is unbounded. Whereas, the transitive closure

[5]Source and target codes of the examined programs are available in the repository https://sourceforge.net/p/traco/code/HEAD/tree/

[6]http://web.cse.ohio-state.edu/ pouchet/software/polybench/

[7]http://www.netlib.org/benchmark/livermorec

[8]http://www.eecg.toronto.edu/ corinna/DSP/infrastructure/UTDSP.html

Algorithm 1: Parallel tiled code generation

Input: A loop nest and its all dependences represented with relation *R*; diagonal matrix ***B***, defining the size of rectangular original tiles.

Output: Code generated according to the free schedule of target tiles: tiles for each time partition are enumerated in parallel whereas statement instances in each tile are scanned serially.

Method:

1) Calculate sets *II_SET*, *TILE_VLD*, and *TILE_VLD_EXT* according to the loop nest tiling algorithm [9].
2) Form relation R_TILE and transform it into relation R_TILE' as follows
   R_TILE′:={[k, ***II***]−>[k+1, ***JJ***]: ∃ ***I***, ***J*** s.t. (***II, I***) ∈ *TILE_VLD_EXT*(***II***) ∧ (***JJ, J***) ∈ *TILE_VLD_EXT*(***JJ***) ∧ ***J*** ∈ *R*(***I***) *AND* k ≥ 0 },
   where ***II, JJ*** are the vectors representing tile identifiers.
3) Calculate set, *UDS_TILE*, as follows
   UDS_TILE:=*II_SET* − range (R_TILE).
4) Form the following relation
   $FS = \{[X] \rightarrow [k, Y] : X \in UDS\_TILE \wedge (k, Y) \in \text{Range}((R\_TILE')^* \setminus \{[0, X]\}) \wedge \neg(\exists\ k' > k \text{ s.t. } (k', Y) \in \text{Range}(RTILE')^+ \setminus \{[0, X]\})\}$,
   where the first element of the second tuple is a parameter k defining time under the free schedule while the next elements (represented with Y) identify tiles.
5) Calculate the set Range(FS) and extend this set by inserting in its last tuple positions the elements of the tuple of set $TILE\_VLD$, returned by step 1, and insert the constraints of set $TILE\_VLD$ into the constraints of set Range(FS).
6) Apply to the set, returned by step 5, CLooG [18] or the isl code generator [19], and postprocess the code to a compilable form of the following structure:

```
seqfor // enumerating time partitions
 parfor // enumerating tile identifiers
        // for a given time partition
  seqfor // enumerating statement instances within
         // the tiles whose identifiers are
         // defined by the previous parfor loop
```

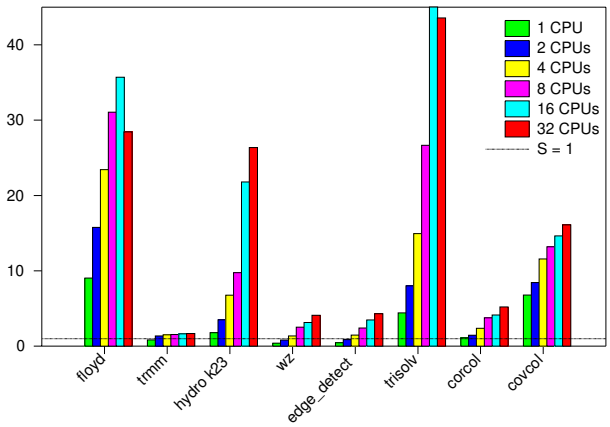

Fig. 1. Speed-up of tiled programs executed on Intel Xeon E5-2695

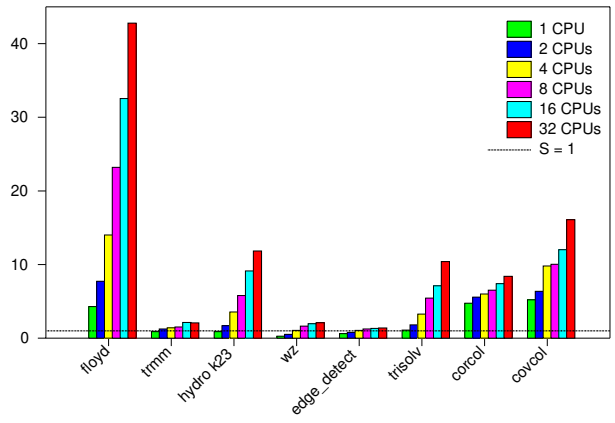

Fig. 2. Speed-up of tiled programs executed on Intel Xeon Phi 7120P

of R_TILE' can be calculated exactly for those programs as well as for the rest of the examined loop nests.

To carry out experiments, we have used a computer with the following features: Intel Xeon CPU E5-2695 v2, 2.40GHz, 12 cores, 24 Threads, 30 MB Cache, 16 GB RAM. We examined parallel code performance using also 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.

Figures 1 and 2 depict the speed-up of the programs executed on Xeon E5-2695 v2 and Xeon Phi 7120P cores, respectively. The speedup, *S=T(1)/T(P)*, is defined as the ratio of the time of an original program execution to that of the corresponding parallel tiled one on *P* processors. The baseline *S=1* presents the speed-up equal to 1.

Analyzing the results, we may conclude that for the studied programs, performance improvement is achieved by means of the presented algorithm. For some programs due to considerable increasing program locality super-linear speed-up is observed.

VII. Conclusion

In this paper, we presented a novel approach based on the transitive closure of dependence graphs to form tiles and their free schedule. The algorithm was implemented in the open source TRACO compiler. Experiments demonstrated that the speed up of examined parallel numerical codes generated by the approach can be achieved on shared memory machines with multi-core processors. The usage of the free schedule of tiles instead of that of loop nest statement instances improves memory utilization and allows us to adjust the parallelism grain-size to match the inter-processor communication capabilities of the target architecture.

In future, we plan to study parametric tiling based on transitive closure aimed at generating more flexible code for affine loop nests in numerical programs.

References

[1] 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

[2] M. Griebl, "Automatic parallelization of loop programs for distributed memory architectures," Habilitation thesis, Department of Informatics and Mathematics, University of Passau., 2004. [Online]. Available: http://www.fim.unipassau.de/cl/publications/docs/Gri04.pdf

[3] 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 pp. 319–329.

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

[5] J. Xue, "On tiling as a loop transformation," *Parallel Processing Letters*, vol. 7, no. 04, pp. 409–424, 1997. doi: 10.1142/s0129626497000401

[6] 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.

[7] 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.

[8] D. Wonnacott, T. Jin, and A. Lake, "Automatic tiling of "mostly-tileable" loop nests," in *5th International Workshop on Polyhedral Compilation Techniques*, Amsterdam, 2015.

[9] W.Bielecki and M. Palkowski, "Tiling 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

[10] 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.

[11] W. Bielecki, M. Palkowski, and T. Klimek, "Free scheduling of tiles based on the transitive closure of dependence graphs," vol. 11TH International Conference on Parallel Processing and Applied Mathematics, Part II, LNCS 9574 proceedings, 2015. doi: 10.1007/978-3-319-32152-3.13 pp. 133–142.

[12] S. Verdoolaege, "Integer set library - manual," Tech. Rep., 2011. [Online]. Available: www.kotnet.org/~skimo//isl/manual.pdf,

[13] 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.

[14] W. Bielecki and M. Palkowski, *Facing the Multicore - Challenge II: Aspects of New Paradigms and Technologies in Parallel Computing*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, ch. Using Free Scheduling for Programming Graphic Cards, pp. 72–83.

[15] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott, *New User Interface for Petit and Other Extensions*, 1996.

[16] W. Bielecki, T. Klimek, M. Palkowski, and A. Beletska, "An iterative algorithm of computing the transitive closure of a union of parameterized affine integer tuple relations," in *COCOA 2010: LNCS*, vol. 6508/2010, 2010. doi: 10.1007/978-3-642-17458-2.10 pp. 104–113.

[17] C. Lengauer, *Loop parallelization in the polytope model*. CONCUR'93, Springer Berlin Heidelberg, 1993, pp. 398–416.

[18] C. Bastoul, "Code generation in the polyhedral model is easier than you think," in *PACT'13 IEEE Int. Conference on Parallel Architecture and Compilation Techniques*, Juan-les-Pins, 2004. doi: 10.1109/pact.2004.1342537 pp. 7–16.

[19] T. Grosser, S. Verdoolaege, and A. Cohen, "Polyhedral ast generation is more than scanning polyhedra," *ACM Trans. Program. Lang. Syst.*, vol. 37, no. 4, pp. 12:1–12:50, Jul. 2015. doi: 10.1145/2743016

[20] J. Bylina and B. Bylina, "Parallelizing nested loops on the Intel Xeon Phi on the example of the dense WZ factorization," in *2016 Federated Conference on Computer Science and Information Systems (FedCSIS)*, Sept 2016. doi: 10.15439/2016f436 pp. 655–664.