Włodzimierz BIELECKI, Piotr SKOTNICKI

West Pomeranian University of Technology, Szczecin

doi:10.15199/48.2015.11.41

Concurrent Start Tiling of Stencil Computations based on the Transitive Closure of a Data Dependence Graph

Abstract. *Stencil computations stand at the core of a wide range of scientific and engineering solutions. Load-balanced execution of stencil kernels, allowing for full utilization of processing units from the very beginning, is therefore the subject of a considerable amount of research. This paper presents a novel approach to generating parallel tiled code of stencil loops, based on the application of the transitive closure of a data dependence graph and a combination of the polyhedral model and the iteration space slicing framework.*

Streszczenie. *Iteracyjne obliczenia, będące funkcją wartości punktów pewnej przestrzeni w czasie, stanowią podstawę szerokiego zakresu rozwiązań naukowych i inżynieryjnych. Efektywne wykonanie realizujących je pętli programowych, poprzez pełne i zrównoważone wykorzystanie dostępnych jednostek obliczeniowych od samego początku, jest przedmiotem znacznej liczby badań. Artykuł prezentuje nowe podejście do blokowania omawianych pętli, bazujące na zastosowaniu domknięcia przechodniego grafu zależności danych oraz technice podziału przestrzeni iteracji.* ***(Transformacja pętli programowych przez blokowanie z równoczesnym rozpoczęciem obliczeń za pomocą domknięcia przechodniego grafu zależności danych)***

Keywords: optimizing compilers, tiling, stencil, transitive closure, parallel computing, dependence graph, code locality
Słowa kluczowe: kompilatory optymalizujące, blokowanie pętli, domknięcie przechodnie, obliczenia równoległe, graf zależności, lokalność kodu

Introduction

Stencil computations are a commonly used class of iterative kernels which involve updating a *D*-dimensional spatial grid over *T* time steps. A new value of each grid point at time *t* is defined as a function of the values of neighboring points at the time steps preceding *t*. Algorithms for solving partial differential equations as well as image processing methods are the examples of scientific problems where computations following the aforementioned pattern can be often encountered.

From the implementation perspective, a stencil kernel comprises a set of nested loops iterating over a time domain and spatial dimensions respectively, with actual calculations taking place in the innermost nest. Fig. 1 shows a sample 2-point stencil kernel, updating the values of array `A`.

```
for (t=1; t<T; ++t) {
  for (i=1; i<N; ++i) {
    A[t][i] = A[t-1][i-1] + A[t-1][i+1];
  }
}
```

Fig. 1. Code of a sample 2-point stencil kernel

Obviously, a low locality of data accesses spread along all dimensions leads to reduced performance and makes stencil codes a subject to applying optimization techniques. *Tiling* is a key code transformation for improving both spatial and temporal localities by means of grouping the points of an iteration space into smaller blocks (*tiles*), therefore allowing for an efficient use of hardware registers and a cache memory.

In this paper, a novel approach to tiling stencil computations based on the application of the transitive closure of a data dependence graph is presented. Additionally, we demonstrate how to deal with inter-tile dependences, which in turn allows for parallel execution of generated tiles with a concurrent start of computations.

Background

The presented algorithm exploits a combination of the *Polyhedral Model* [3] and the *Iteration Space Slicing* framework [10, 14].

In this paper, we deal with *affine loop nests* where lower and upper bounds, array subscripts, and conditionals are affine functions of surrounding loop indices and symbolic constants, and the loop steps are known constants. A loop nest is *perfectly nested* if all its statements are contained in the innermost nest.

A *statement instance* $s(I)$ is a particular execution of a loop statement s for a given iteration I. Two statement instances $s_1(I)$ and $s_2(J)$ are *dependent* if both access the same memory location and if at least one access is a write. $s_1(I)$ and $s_2(J)$ are called the source and the target of a dependence, respectively, provided that $s_1(I)$ is executed before $s_2(J)$. The sequential ordering of statement instances, denoted $s_1(I) \prec s_2(J)$, is induced by the lexicographic ordering of iteration vectors, or by the textual ordering of statements if $I = J$. An iteration vector can be represented by a k-integer tuple of loop indices in $\mathbb{Z}^k$ iteration space. Consequently, a dependence relation is a mapping from tuples to tuples of the form $\{\, [source] \to [target] : constraints \,\}$, where $source$ defines dependence sources, $target$ defines dependence targets, and $constraints$ is a Presburger formula that imposes constraints on the variables and/or expressions within $source$ and $target$ tuples. Throughout this paper, we use the syntax of *Barvinok* [12] to present the results of calculations on sets and relations.

A code transformation is performed by means of extracting *program slices* – sets of statement instances that can affect the values of memory locations at some points of interest – and separating an iteration space into subspaces. The overall process starts with the program analysis phase which translates the code to its polyhedral representation and computes data dependences. Then, algorithms aimed at optimizing the code manipulate the integer sets of the polyhedral model to find a more efficient execution order of statement instances, respecting the data dependences. Eventually, the code generation phase restores a high level, but optimized, representation of the original program.

Let us remind that *"The key step in calculating an iteration space slice is to calculate the transitive closure of the data dependence graph of the program"* [10]. Given a tuple relation R, its positive transitive closure is defined as follows:

$$R^+ = \{\, e \to e' \mid e \to e' \in R \vee \exists e''\, s.t.\, e \to e'' \in R \wedge e'' \to e' \in R^+ \,\}. \tag{1}$$

The transitive closure of a data dependence graph and, in particular, its application to given iterations of interest, provide a solution to the problem of finding all transitively dependent points of an iteration space that must be included in a slice. Techniques for computing the transitive closure of a data dependence graph are presented in papers [4, 13].

Tiling algorithm

To demonstrate the proposed algorithm, let us consider the working example from Fig. 1 with the 6 x 9 iteration domain. The dependences of the loop nest can be described by the following relations:

R1 := { [*t*, *i*] -> [*t*+1, *i*+1] : 1 <= *t* <= 5 and 1 <= *i* <= 8 };

R2 := { [*t*, *i*] -> [*t*+1, *i*-1] : 1 <= *t* <= 5 and 2 <= *i* <= 9 }.

We start with forming initial tiles of the size 3 x 3. Fig. 2 illustrates the 6 x 9 iteration space and the dependence graph, where vertices represent the iterations of the loop nest, and directed edges connect pairs of dependent iterations. The dashed lines form initial tiles, grouping the iterations into blocks *T00*, *T01*, *T02*, *T10*, *T11*, *T12*. A tile identifier corresponds to its position relative to the iteration space axes.

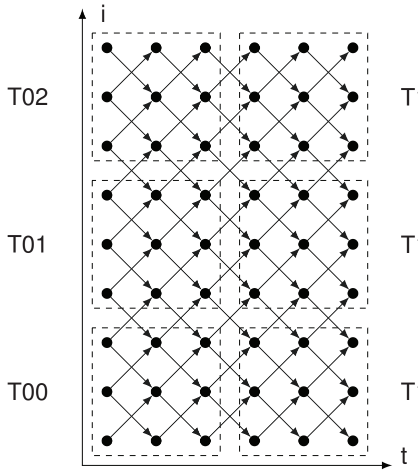

Fig. 2. Iteration space with initial 3 x 3 rectangular tiles

Tiling is said to be *valid* if tiles are executed atomically without violating data dependences. Clearly, scanning the initial rectangular tiles in the lexicographic order is invalid due to not respecting inter-tile dependences. That is, iteration (2,4) is executed after iteration (1,3) is finished. However, iteration (2,3) is dependent on the result of iteration (1,4). To respect dependences among tiles we may proceed as follows. First, from each tile we remove the iterations that are the targets of either direct or transitive dependences whose sources are contained in the other tiles. This leads to the removal of iterations { (2,3), (3,2), (3,3) } from *T00*, iterations { (2,4), (2,6), (3,4), (3,5), (3,6) } from *T01*, and iterations { (2,7), (3,7), (3,8) } from *T02*. As a result, we have formed the first set of independent *subtiles*, namely $T00^1$, $T01^1$, and $T02^1$, presented in Fig. 3. Next, we subtract the iterations belonging to all extracted subtiles from the original iteration space and repeat the procedure above, however, this time without considering the dependences originated from any of the already formed subtiles. This implies the removal of iteration (3,3) from *T00*, iterations { (3,4), (3,6) } from *T01*, and iteration (3,7) from *T02*, which eventually forms subtiles $T00^2$, $T01^2$, and $T02^2$. The procedure is repeated until each iteration within each original tile is assigned to any of subtiles. As far as the working example is considered, the final result is presented in Fig. 4.

Let us note that all dependences are carried along the time dimension. This enforces a traversal of tiles in the increasing order along the time axis. For that reason, at each stage of calculations we consider only the dependences among tiles within the same *time period*, therefore splitting tiles *T10*, *T11*, *T12* into subtiles is carried out in the same way as that for tiles *T00*, *T01*, and *T02*.

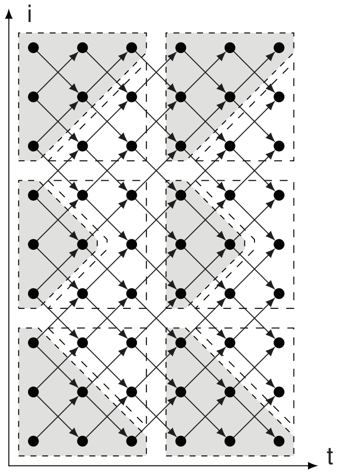

Fig. 3. Computed subtiles (shaded) and the remaining iterations

Finally, we define the order in which iterations are executed. The execution of subtiles is divided into phases (*waves*) whose number is equal to the number of subtiles computed for a parametric tile. A wavefront operates on the subtiles of the currently processed wave in parallel in all tiles. The procedure is repeated for each time period. Iterations inside each subtile are executed according to the original, lexicographic order.

We can observe that the described tiling scheme has the *concurrent start* property, which means a wavefront processes all the tiles from the very beginning, and retains a constant load-balance throughout the computations.

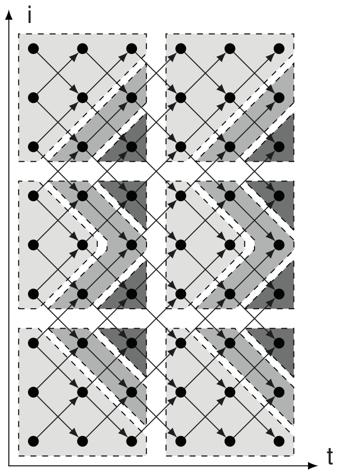

Fig. 4. Complete tiling scheme

In a formal way, initial rectangular tiles can be represented by a parametric set *TILE* that defines the iterations included in a tile identified by symbolic constants (*tt, ii*). For the working example, set *TILE* is defined as follows:

TILE := [*tt*, *ii*] -> { [*t*, *i*] : *t* >= 1 + 3*tt* and *tt* <= 1 and *t* <= 3 + 3*tt* and *tt* >= 0 and *i* >= 1 + 3*ii* and *ii* <= 2 and *i* <= 3 + 3*ii* and *ii* >= 0 }.

For the purpose of determining which iterations should be removed in order to form a subtile, we exploit the following definition of an *invalid dependence target for a time period*:

Definition 1. If for the same time period there exists a direct or transitive dependence whose target belongs to set *TILE* and its source belongs to a tile with a less or greater identifier than that of *TILE*, then the target of this dependence is invalid within set *TILE*.

Forming a set of invalid targets requires the introduc-

tion of two additional sets, namely *TILE_LT* and *TILE_GT*. They are parametric sets that for given values of symbolic constants representing any tile of interest contain the iteration points belonging to tiles of the same time period, and of less or greater identifiers, respectively. As far as the working example is considered, these sets are defined as follows:
TILE_LT := [*tt*, *ii*] -> { [*t*, *i*] : *tt* <= 1 and *tt* >= 0 and *i* >= 1 and *i* <= 3*ii* and *ii* <= 2 and *t* >= 1 + 3*tt* and *t* <= 3 + 3*tt* };
TILE_GT := [*tt*, *ii*] -> { [*t*, *i*] : *tt* <= 1 and *tt* >= 0 and *i* <= 9 and *ii* >= 0 and *i* >= 4 + 3*ii* and *t* >= 1 + 3*tt* and *t* <= 3 + 3*tt* }.

To obtain a set of invalid targets within a tile we use the operation of the application of relation R to set S, which results in the range of relation R with domain S:

(2) $$R(S) = \{\, e' \mid \exists e \in S \, s.t. \, e \to e' \in R \,\}.$$

Therefore, by applying the relation representing the positive transitive closure of a data dependence graph, *R+*, to sets *TILE_LT* and *TILE_GT*, and subtracting the results from the original space *TILE*, we are able to extract the first parametric subtile:
SUBTILE1 := *TILE* - *R+*(*TILE_LT*) - *R+*(*TILE_GT*).

The remaining iterations of the original set *TILE* can be obtained with the following formula:
TILE2 := *TILE* - *SUBTILE1*.

To form subsequent subtiles, we iteratively carry out the following steps: (1) form sets *TILE_LT2* and *TILE_GT2* corresponding to *TILE2*, (2) apply the relation of the positive transitive closure to those sets, (3) subtract the results from *TILE2*.

To generate parallel tiled code, we apply the following scheme. The outermost loop enumerates tiles in the increasing order of identifier values corresponding to a time dimension. The iterator of this loop represents a time period. Each its iteration executes consecutive subtiles in a serial manner. The execution of a single subtile starts with iterating over the ranges of identifier values corresponding to spatial dimensions, which forms a separate and fully parallel loop nest for each subtile. In the body of a loop nest responsible for a given set *SUBTILE* we insert the code produced by means of applying to this set any code generator allowing for scanning its elements in the lexicographic order, e.g. *CLooG* [2].

Algorithm 1 provides a formal description of the approach presented in this paper.

Algorithm 1 Tiling for perfectly nested stencil loops

Input: A perfect loop nest of depth d; constants b_1,b_2,...,b_d defining the size of a single tile.
Output: Parallel tiled code.

(1) Form the following vectors and matrix:
vector ***I*** whose elements are original loop indices i_1,i_2,...,i_d;
vector ***II*** whose elements ii_1,ii_2,...,ii_d define the identifier of a tile;
vectors ***LB*** and ***UB*** whose elements are lower lb_1,...,lb_d and upper ub_1,...,ub_d bounds of indices i_1,i_2,...,i_d of the original loop nest, respectively;
vector ***1*** whose all d elements are equal to the value 1;
vector ***0*** whose all d elements are equal to the value 0;
diagonal matrix ***B*** whose diagonal elements are constants b_1,b_2,...,b_d defining a tile size.

(2) Perform a data dependence analysis to produce a set of relations describing all the dependences in the original loop nest. Remove redundant dependences applying any well-known technique.

(3) Compute the positive transitive closure *R+* of the union of all the relations returned by step (2) applying any known algorithm.

(4) Form a set *TILE* including the iterations belonging to a parametric tile defined with parameters ii_1,ii_2,...,ii_d as follows:
TILE(***II***,***B***) = [***II***]$\to$\{ [***I***] | ***B********II***+***LB*** $\leq$ ***I*** $\leq$ min(***B*****(***II***+***1***)+***LB***-***1***,***UB***) AND ***II*** $\geq$ ***0*** \}

(5) Form a set *II_SET* including the identifiers of all tiles:
II_SET = { [***II***] | ***II*** $\geq$ ***0*** AND ***B********II***+***LB*** $\leq$ ***UB*** }

(6) $k \leftarrow 0$

(7) `while` *TILE* $\neq \varnothing$ `do:`

(7.1) Form a set *TILE_LT* as the union of all the tiles whose identifiers are lexicographically less than that of *TILE*, and which are in the same time period as that of *TILE*, as follows:
TILE_LT(***II***,***B***) = [***II***]$\to$\{ [***I***] | $\exists$ ***II'*** s.t. ***II'*** $\prec$ ***II*** AND ii_1=ii'_1 AND ***II***,***II'*** $\in$ *II_SET* AND ***I*** $\in$ *TILE*(***II'***,***B***) \}

(7.2) Form a set *TILE_GT* as the union of all the tiles whose identifiers are lexicographically greater than that of *TILE*, and which are in the same time period as that of *TILE*, as follows:
TILE_GT(***II***,***B***) = [***II***]$\to$\{ [***I***] | $\exists$ ***II'*** s.t. ***II'*** $\succ$ ***II*** AND ii_1=ii'_1 AND ***II***,***II'*** $\in$ *II_SET* AND ***I*** $\in$ *TILE*(***II'***,***B***) \}

(7.3) $k \leftarrow k + 1$

(7.4) Compute a subtile as follows:
SUBTILE$_k$:= *TILE* - *R+*(*TILE_LT*) - *R+*(*TILE_GT*)

(7.5) Subtract the computed subtile from the iteration space:
TILE := *TILE* - *SUBTILE*$_k$

(8) Generate the outermost loop with iterator ii_1, ranging from 0 to $(ub_1 - lb_1) \,/\, b_1$ inclusive.

(9) `for each` set *SUBTILE*$_{1..k}$ `do:`

(9.1) Generate a fully parallel loop nest of the rest ii_2,...,ii_d iterators, ranging from 0 to $(ub_n - lb_n) \,/\, b_n$ inclusive, $2 \leq n \leq d$.

(9.2) In the body of the innermost loop insert the tiled code generated by means of applying any code generator scanning the elements of set *SUBTILE* in the lexicographic order.

Related work

A considerable amount of research has been carried out to devise a tiling scheme to maximize a locality and parallelism. In contrast to our approach, well-known techniques are based on the *Affine Transformations* framework.

Attempts to circumvent the problem of inter-tile dependences, inhibiting a concurrent start, by means of partitioning are not new. Paper [8] discusses the *split tiling* technique, based on dividing each tile into two subregions, thereby enabling a concurrent start. The paper also investigates the benefits of *overlapped tiling* which eliminates inter-tile dependences through the introduction of redundant calculations in adjacent tiles.

More recently, paper [1] presents the implementation details of the *diamond tiling* scheme. This technique employs affine transformations to divide an iteration space into parallelotopes.

Stencil kernels expose a significant degree of parallelism, which makes them amenable to execution on massively-parallel architectures. Some works have looked at offloading stencil computations to GPUs. Paper [7] uses trapezoidal shapes obtained by means of the split tiling technique. *Hybrid hexagonal tiling* [6] improves the diamond tiling scheme to make the shape of tiles more suitable for GPGPU.

The algorithm presented in this paper derives from the previous work described in [5] which formalized the basis of forming and reshaping tiles by means of the application of the transitive closure of a data dependence graph. However, that paper focuses on improving the locality of a general class of perfectly nested loops, without exploring opportunities for parallel execution.

Results of experiments

The algorithm presented in this paper was implemented in a tool which exploits *Polyhedral Extraction Tool* [11] for extracting a polyhedral model, and uses *Integer Set Library* [13] for performing dependence analysis and manipulating integer sets and relations. After finding subtiles, code is generated with CLooG [2]. The implementation has been incorporated into the *TC* optimizing compiler[1].

The correctness of the approach was verified by means of comparing the results obtained through execution of original and transformed loop nests. For experiments, we have chosen a 3-point kernel of a 1-D heat equation solver with non-periodic boundaries as a representative example of stencil computations. The hardware and software configuration of the environment used for the experiments is presented in Table 1. The parallelism of tiled code was represented by means of the *OpenMP API* [9]. The problem size and serial code execution time are shown in Table 2. The results of the experiments are summarized in Table 3.

Table 1. Environment used for experiments

Processor	Intel Core i7-4790
Clock	3.6 GHz
Number of cores	4
Number of threads	8
L1 data cache / core	32 KB
L2 cache / core	256 KB
L3 cache / socket	8192 KB
RAM memory	8 GB @ 1333 MHz
Linux kernel	3.16.0 x86_64
Compiler	gcc 4.8.2
Compiler flags	-O3 -fopenmp

Table 2. Problem size and serial code execution time

Problem size	Sequential execution time [ms]
1000 x 3600000	5353.379

Table 3. Parallel execution times and speed-up of tiled code

Block size	TC parallel execution time [ms]	TC parallel speed-up
50 x 5000	582.597	9.1888
50 x 10000	587.066	9.1189
50 x 25000	604.018	8.8629
100 x 5000	579.795	9.2332
100 x 10000	574.736	9.3145
100 x 25000	576.125	9.2920
125 x 5000	587.435	9.1131
125 x 10000	572.673	9.3481
125 x 25000	569.852	9.3943
150 x 5000	586.451	9.1284
150 x 10000	571.711	9.3638
150 x 25000	**568.867**	**9.4106**

Conclusion

In this paper, we presented a novel approach to tiling perfectly nested loops of stencil computations. The algorithm is based on the application of the transitive closure of a data dependence graph and a combination of the polyhedral model and the iteration space slicing framework. We demonstrated that this approach generates parallel tiled code whose speed-up is satisfactory for practical needs. The main merit of the presented algorithm in comparison with techniques based on affine transformations is that it allows for arbitrary automatically formed shapes of tiles with a concurrent start.

In our future research, we plan to improve this approach in order to provide a balanced distribution of work among tiles and to increase the granularity of parallel computations. This will allow us to increase the speed-up of parallel tiled code.

[1] http://tc-optimizer.sourceforge.net

REFERENCES

[1] Bandishti V., Pananilath I., Bondhugula U.: Tiling Stencil Computations to Maximize Parallelism, In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, IEEE Computer Society Press, (2012).

[2] Bastoul C.: Code Generation in the Polyhedral Model Is Easier Than You Think, In Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques, pp. 7–16, IEEE Computer Society, (2004).

[3] Benabderrahmane M. W., Pouchet L. N., Cohen A., Bastoul C.: The Polyhedral Model Is More Widely Applicable Than You Think, Compiler Construction, Springer Berlin Heidelberg, pp. 283–303, (2010).

[4] Bielecki W., Klimek T., Pałkowski M., Beletska A.: An Iterative Algorithm of Computing the Transitive Closure of a Union of Parameterized Affine Integer Tuple Relations, In COCOA 2010: Fourth International Conference on Combinatorial Optimization and Applications, Lecture Notes in Computer Science, vol. 6508/2010, pp. 104–113, (2010).

[5] Bielecki W., Pałkowski M.: Perfectly Nested Loop Tiling Transformations Based on the Transitive Closure of the Program Dependence Graph, Soft Computing in Computer and Information Science, vol. 342, pp. 309–320, Springer International Publishing, (2015).

[6] Grosser T., Cohen A., Holewinski J., Sadayappan P., Verdoolaege S.: Hybrid Hexagonal/Classical Tiling for GPUs, In Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization, p. 66, ACM, (2014).

[7] Grosser T., Cohen A., Kelly P. H., Ramanujam J., Sadayappan P., Verdoolaege S.: Split Tiling for GPUs: Automatic Parallelization Using Trapezoidal Tiles, In Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units, pp. 24–31, ACM, (2013).

[8] Krishnamoorthy S., Baskaran M., Bondhugula U., Ramanujam J., Rountev A., Sadayappan P.: Effective Automatic Parallelization of Stencil Computations, ACM SIGPLAN Notices, vol. 42, no. 6, pp. 235–244, ACM, (2007).

[9] OpenMP Architecture Review Board: OpenMP Application Program Interface Version 3.0, www.openmp.org/mp-documents/spec30.pdf, (2008).

[10] Pugh W., Rosser E.: Iteration Space Slicing and Its Application to Communication Optimization, In Proceedings of the 11th International Conference on Supercomputing, pp. 221–228, ACM, (1997).

[11] Verdoolaege S., Grosser T.: Polyhedral Extraction Tool, Second International Workshop on Polyhedral Compilation Techniques (IMPACT'12), Paris, France, (2012).

[12] Verdoolaege S.: Barvinok – User Guide, Version: barvinok-0.37, compsys-tools.ens-lyon.fr/iscc/barvinok.pdf, (2014).

[13] Verdoolaege S.: Integer Set Library – Manual, Version: isl-0.14, isl.gforge.inria.fr/manual.pdf, (2014).

[14] Weiser M.: Program Slicing, In IEEE Transactions on Software Engineering, pp. 352–357, (1984).

Authors: *prof. dr hab. inż. Włodzimierz Bielecki, email: wbielecki@wi.zut.edu.pl; mgr inż. Piotr Skotnicki, email: pskotnicki@wi.zut.edu.pl; West Pomeranian University of Technology, Faculty of Computer Science and Information Technology, ul. Żołnierska 49, 70-210 Szczecin, Poland*