Michał GOZDALIK
WEST POMERANIAN UNIVERSITY OF TECHNOLOGY IN SZCZECIN,
Zolnierska Street 52, 71-210 Szczecin, Poland

Use of the tiling method inside synchronization of free slices of code in OpenMP standard in order to achieve speedup enhancement

M.Sc. Michał GOZDALIK

A PhD student at the West Pomeranian University of Technology in Szczecin. Currently occupied with creating a tool allowing generation of parallel code in C, with the consent of OpenMP standard, which could take the most possible advantage of multi-processor machines.

e-mail: mgozdalik@wi.ps.pl

Abstract

In last few years, there were discovered many methods aiming at enhancing the speedup of parallel programs. In this paper three methods are tested according to a speedup parameter enhancement. These methods are: the tiling, the slicing, and the tiling inside slicing. In Sections 3, 4, and 5 the theoretical basis for chosen transformation are described. Algorithms of transformation processes as operations on a polyhedral model are presented. The problems of transformation costs are also discussed. For experimental studies a UTDSP benchmark was used. From each section, one representative sample was chosen. The results were also examined against a data locality. This aspect of chosen transformation methods was examined as well.

Keywords: OpenMP, tiling, shared memory programming.

Zwiększanie przyspieszenia aplikacji równoległych przy użyciu metody podziału na bloki, wewnątrz części kodu wolnych od synchronizacji

Streszczenie

W artykule przedstawiono problem doboru metody transformacji pętli celem uzyskania możliwie maksymalnego przyspieszenia. Do badań wybrano benchmark UTDSP z uniwersytetu w Toronto. Z każdej sekcji benchmarku wybrano reprezentanta, który poddany został transformacjom tiling, slicing oraz transformacji tiling wewnątrz slicingu. W pierwszym rozdziale przedstawiony został wstęp do transformacji pętli. Rozdział drugi zawiera informacje teoretyczne na temat modelu polihedronu jako formy reprezentacji pętli, na której przeprowadzane są transformacje, a wynikowy model jest bazą do generowania kodu źródłowego. Kolejne rozdziały przedstawiają opis teoretyczny transformacji tiling oraz slicing. Przedstawiono w nich algorytm tworzenia tych transformacji wraz z przekształceniami matematycznymi, opisującymi transformacje na modelu polihedronu. W końcowej części pracy badano wpływ wybranych transformacji na przyspieszenie programów. Wyniki badań przedstawione zostały w formie zagregowanych wykresów przyspieszeń poszczególnych aplikacji.

Słowa kluczowe: OpenMP, programowanie równoległe.

1. Introduction

Parallel program transformations aim at transforming a loop in such a manner that the result program will enhance the speedup. The difficulty of programming multi core architectures to effectively tap the potential speedup is a well-known challenge. The long running programs spend most of their time inside finite loops. Effective transformations of such loops can significantly increase the speedup of parallel programs.

In this paper a new approach is developed to increase the speedup of parallel programs. It joins two transformations to enlarge the locality effect and remove synchronization barriers wherever it is possible.

In the sections of this paper all theoretical and practical examples are examined to reveal the strongest and weakest aspects of this approach.

2. Polyhedral model

The polyhedral model is an abstract representation of loops, whose iteration space is well-known or can be defined during the runtime. Each loop statement can be defined as an integer point in the n-dimensional space named a polyhedron. In such a representation, it is easy to define any affine transformation. It is only necessary to obtain representation of a data dependency in each statement of a loop. This problem is easily solved by linear programming and linear algebra. Moreover, there are no difficulties to automatically generate the parallel code of a loop after transformation from such a model. As a result, the generated code contains reordered statements inside the loop or even extra nested loops which enhances data locality. All benefits of the polyhedral model are applicable to loop nests in which data dependencies and loops bounds can be reflected as affine combinations of the outer loop variables and parameters.

In Fig. 1 the polyhedral model representation of a loop iteration space is shown. The loop code is

```
for (i = 1; i <= 7; i++) {
      for (i = j - 1; j <= 6; j++) {
              S(i, j);
      }
}
```

where *S* is a regular loop statement, other than the loop control statement, such as break or continue. Each dot represents an iteration index and can be combined with a vertex. Dependencies can be described as a vector which contains values for the indices of the loop surrounding the statement S with all boundaries.

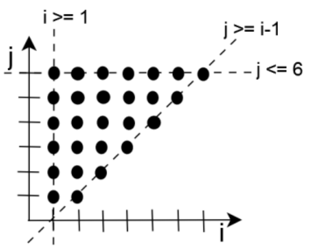

Fig. 1. Example of polyhedral model
Rys. 1. Przykład reprezentacji pętli w postaci modelu polihedronu

According to the definition, a polyhedron is the set of all vectors $\vec{x} \in Q^n$ such that $A\vec{x} + \vec{b} \geq 0$. A bounded polyhedron is named a polytope. Each instance of statement S is defined by an iteration vector $\vec{\imath}$ during runtime. Such a vector contains indices of the loop surrounding *S* from outermost to innermost. The statement *S* is also combined with a polytope of dimension *n*, so each point in the polytope is *n* dimensional vector and the polytope can be represented as a set of bounding hyperplanes. It is crucial to mention that this is only true when the loop bounds are linear combinations of outer loop indices and symbolic constants representing the problem size.

3. Synchronization of free slices method

This method generally bases on finding sets of iterations, which can be executed in parallel, without using synchronization techniques. It can be achieved by the following algorithm presented below.

Let q be a number of vertexes in a data dependencies graph. S will be an indicator of a set with all data dependencies relations, where relation R_{ij} is a union of relations dependencies between instruction s_i and s_j. Now it is possible to notate

$$S := \{R_{ij} \mid i,j \in <1, q>\}$$

For each of relation R_{ij} in set S, it is necessary to expand a relation by adding one dimension, which will represent the *i-th* and *j-th* instruction number. For example

$$R_{ij} := \{[e] \to [e']\} \to R_{ij} := \{[e,i] \to [e',j]\}$$

Now it is crucial to establish R as a union of all relations in set S.

$$R := \bigcup_{i \geq 1, j \leq q, R_{ij} \in S} R_{ij}$$

Having the set R, for all instructions s_i we need to find a set *UDS(i)* which is a difference of union of innermost and outermost domains corresponding to instruction $s_{i..}$

$$UDS(i) := \bigcup_{1 \leq k \leq g} R_{ik} - \bigcup_{1 \leq k \leq g} R_{ki}$$

Finally, a set *UDS* needs to be calculated, as a sum of all sets *UDS(i)*, to construct relation *R_UCS(i)* as it is stated below.

$$R_{UCS(i)} := \{[e] \to [e'] \mid e \in UDS(i), e' \in UDS, e' > e\}$$

R_UCS(i) is a relation which represents all joined sources of synchronization of free slices in the data dependency graph for relation *R*. It is crucial that redundant dependencies are not taken into consideration, even if they occur in the data dependencies graph. This situation takes place when the instruction depends on itself among iterations. Subtraction of *UDS(i)* and a range of *R_UCS(i)* is a source of synchronization of the free slice in a parallel code.

4. The Tiling method

In general, a goal of tiling transformation is to partition the iterations space into uniform parts of a given size and shape. Generally, there are two types of tiling regarding the non distributed memory machines. Rectangular and parallel piped tiling methods are described as models of granularity transformations. Their purpose is to prepare a portion of data which can be stored in the processor cache memory. Latency of cache memory, especially latency of first level cache memory, is even seventy times lower than that of the random access memory. This feature shortens an idle time of processors, which results in enhancement of the parallel programs speedup parameter.

In this paper only the rectangular tiling model is taken into consideration, as one of the simplest and with the lowest extra costs. Broadly speaking, rectangular tiling is modeled as a mapping from Z^n to Z^{2n}. It uses squares or rectangles of the same shape and size to partition an iteration space. Fig. 2 shows an example of rectangular tiling.

To specify the beginning of a tile, it is crucial to establish one of integer points in the tile as the tile origin. Tile origin is placed in the left lower corner of the tile. All tile origins define a lattice. In Fig. 3 the tile origins are marked as opened circles.

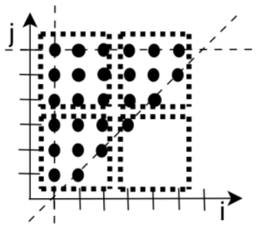

Fig. 2. Example of 3x3 tiling
Rys. 2. Przykład tilingu o rozmiarach 3x3

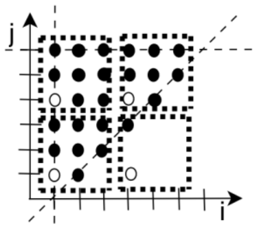

Fig. 3. Tile origins
Rys. 3. Punkty reprezentatywne dla metody tiling

To perform tiling transformation we need to know two vectors: the tile offset vector $\vec{o}$ and the tile size vector $\vec{s}$. The tile transformation R is a bijective. Its representation is shown below.

$$R := \left\{ Z^n \to Z^{2n} \mid R(\vec{\imath}) = \begin{pmatrix} \vec{t} \\ \vec{e} \end{pmatrix} = \begin{pmatrix} \left\lfloor \dfrac{\vec{\imath} - \vec{o}}{\vec{s}} \right\rfloor \\ \vec{\imath} \end{pmatrix} \right\}$$

In this representation we need to show all iterations as an integer convex polytope, to perform all operations on matrices. It is possible to construct loop nests in other way than an integer polytopes, but this solution is very rare and uncommon in use.

5. The tiling inside synchronization of free slice method

Ian Foster was one of pioneers who stated the methodology of designing parallel algorithms. He named his methodology PCAM (Partitiioning-Communication-Agglomeration-Mapping). The name originates from four steps of design process which are:

1. *Partitioning* – tasks and instructions are portioned into smaller parts. In this stage implementation issues are not taken into consideration.
2. *Communication* – in this stage multi processes communication is established. Also, structures and communications algorithms are stated.
3. *Agglomeration* – tasks and communication issues established in the previous steps are evaluated on account of implementation costs. If it is necessary, tasks are combined in larger groups.
4. *Mapping* – tasks are mapped into processor threads to enhance the efficiency manner.

According to the second and third step of the PCAM model, the tiling inside the synchronization of free slice method will enlarge positive effects on the program speedup and efficiency. This statement is based on the fact that the synchronization of free slice algorithm will decrease communication level between processors and tiling algorithm will increase reusability of the first and second level cache memory. Joining this algorithms in that manner should increase the program speedup parameter. However, a question may be raised whether transformation costs will be covered by the speedup effect. The next section with tests on UTDSP (University of Toronto Digital Signal Processing) benchmark will answer this question.

6. Experimental studies

The experimental studies were carried out on the University of Toronto Digital Signal Processing (UTDSP) benchmark. The examples were taken from representative groups of algorithms in the benchmark. The benchmark was split into two main groups. The loops whose name started from FIR are implementations of a finite impulse response filter. The loops whose names started from LATNRM are implementations of a normalized lattice filter. The loops whose names started from LMSFIR are implementations of an adaptive least mean squares filter. For purposes of filter transformations, implementation of the matrix multiplication algorithm was also added as a loop whose name started from MULT. It is important that algorithms and their implementations which are part of the UTDSP benchmark are used in real signal processing software.

The experimental tests were carried out on 32 processors machine. The machine consisted of four processors with eight cores each. The total amount of cache was 12 MB, but each processor could share only maximum 4 MB. Experimental examples were executed five times in a row. To calculate the speedup parameter, the mean of five execution times was taken as input parameter. It is worth noting that each execution time was not measured in a seconds unit. The base of measurements was a tick. The tick is a time in which a processor can execute an atomic instruction. To count the processor ticks, *gettimeofday* function was used. This C function can establish the number of processor ticks only for instructions which exist in a program which is being executed. It is a crucial feature, because during program execution the processor also executes instructions from the operating system and other programs. To eliminate the impact of these programs on measurements, the processor tick count method was chosen. Moreover, all examples were compiled using a gcc compiler without applying any optimization methods built in a gcc compiler.

All results irrespective of the method chosen were tested against scalability. The problem scale, which is placed on *x* axis, is a number of iterations the main loop has made. In the figures in which the slicing method occurs, the data are stored in the first level cache when the problem scale is between 1200 and 2048. When the problem scale is between 2048 and 5120, all data occupy L1 and L2 cache. When the problem scale is 5120 and above, the data also occupy the random access memory. When the tiling method was under tests, a tiling block was chosen to fulfill all data in separate memory levels.

The results of experiments in which only the slicing method was used are shown in Fig. 4.

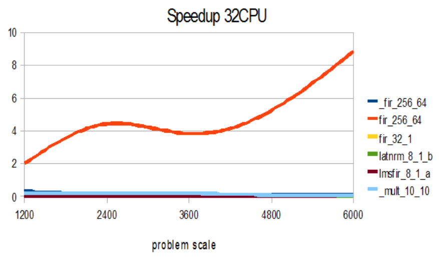

Fig. 4. The results for synchronization free slice method
Rys. 4. Wyniki badań dla metody slicing

In the slicing method only fir_256_64 loop speedup significantly grows. The cost of creating and manipulating the threads consumed all parallel execution benefits in the rest of the examples.

The results of experiments using the tiling method, where all data were placed in L1 memory are shown in Fig. 5.

In the example presented above the tiling method with a block size 64 KB provides better results than the slicing method. The examples, where more than one statement was inside a parallel loop, achieve speedup significantly higher than 1.

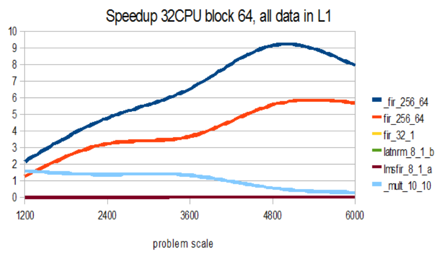

Fig. 5. The results for tiling method. All data stored in L1cache
Rys. 5. Wyniki badań dla metody tiling i danych mieszczących się w pamięci L1

The results of experiments using the tiling method, where all data were placed in a L2 memory, are presented in Fig. 6.

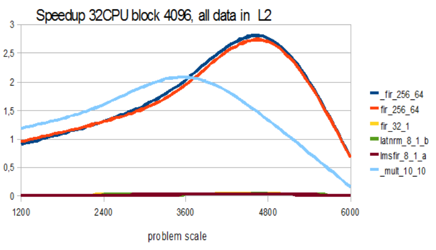

Fig. 6. The results for tiling method. All data stored in L2 cache
Rys. 6. Wyniki badań dla metody tiling i danych mieszczących się w pamięci L2

In this example a tiling block size was 4096 KB. This size was too large to store data in a L1 cache memory, but fits a L2 cache memory. According to Figs. 5 and 6, the cache level where data were stored is a major aspect of the speedup parameter. When data were stored in a L2 cache memory, the speedup parameters were approximately three times lower than those in the previous example.

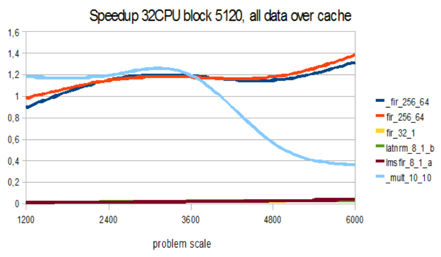

Fig. 7. The results for tiling method. The part of data stored in RAM
Rys. 7. Wyniki badań dla metody tiling i danych nie mieszczących się w pamięci cache

In Fig. 7 in which most part of data were stored in a random access memory, the speedup parameter was the lowest. The memory latency was a crucial aspect of effective tiling transformation.

The results of experiments using the tiling inside the slicing method, where all data were placed in a L1 memory are presented in Fig. 8.

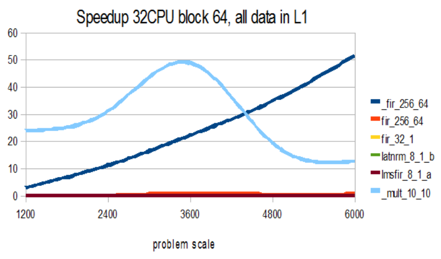

Fig. 8. The results for tiling inside slicing method. All data stored in RAM
Rys. 8. Wyniki badań dla metody tiling wewnątrz metody sliping i danych mieszczących się w pamięci L1

Using the tiling inside the slicing method has the best effect on the speedup parameter. The most scalable example was _fir_256_64, where speedup grows with a problem size. The speedup values were higher than the number of processors. This situation is called hiper-speedup. Unfortunately, a fir_256_64 loop speedup parameter was much lower than in cases where only the tiling method was applied, which means that the cost of using two transformations is higher than the speedup effects, when there are few statements inside a parallel loop.

The results of experiments with the use of the tiling inside the slicing method, where all data were placed in a L2 memory, are shown in Fig. 9.

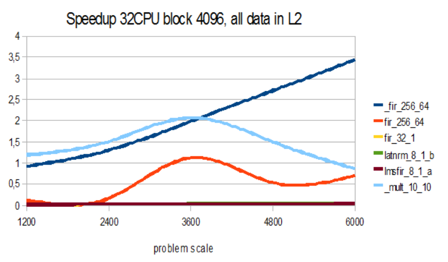

Fig. 9. The results for tiling inside slicing method. All data stored in L2 cache
Rys. 9. Wyniki badań dla metody tiling wewnątrz slicing i danych mieszczących się w pamięci L2

This example shows that a block size in the tiling method has a huge effect on the program speedup. When data were placed in a second level cache memory, the speedup parameter for loop _fir_256_64 was approximately fifteen times lower.

The results of experiments using the tiling inside the slicing method, where part of the data were placed in a random access memory, are shown in Fig. 10.

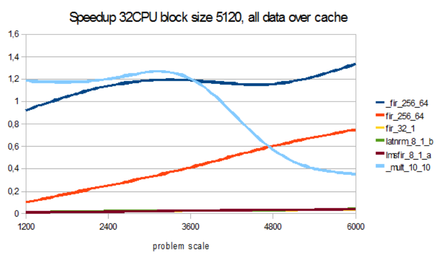

Fig. 10. The results for tiling inside slicing method. Partially, data stored in random access memory
Rys. 10. Wyniki badań dla metody tiling wewnątrz slicing i danych nie mieszczących się w pamięci cache

Storing data in a random access memory caused speedup parameter downgrade to the lowest value from among all experiments.

The all experimental studies reveal that all methods have strong and weak points. The final effect depends on loop structures. All loop transformation methods provide additional piece of a code, sometimes by adding extra loops. In cases in which inside the loop there are few statements, a transformation generates so many additional statements that the expected speedup is very low. In the experimental studies presented in this paper such a situation took place for latnrm_8_1_b, fir_32_1 and lmsfir_8_1_a. The transformed code contained so many additional statements that its execution consumed all the benefits of the transformation and the speedup parameter was barely over the zero value.

However, when there were a lot of statements inside parallel loops, the tiling inside the slicing method enlarged the speedup parameter to the value nearly twice as big as a number of processors. The most crucial aspect of transformation was to establish the tile size in such a manner that all data were stored in a first level cache memory. When the data exceeded the cache memory, the results were not satisfactory.

7. Summary

The experimental studies reveal that each transformation under the test consists of the strongest and weakest points. One of the most important aspects of enhancing the speedup parameter is utilization of the first level cache memory. Generally, when data were stored in the first level cache memory, irrespective of the transformation method, the speedup parameter was significantly higher than in examples where data were too large to be stored in a L1 cache memory. The higher level memory was used, the lower the speedup parameter values were achieved.

The second aspect of enhancing the speedup parameter is the number of statements in a loop under transformation. In general, the more statements in the loop which is under transformation, the better speedup of a parallel program can be achieved. Evidently, more statements can be reflected in more dependencies in a code, but almost every type of a dependency can be resolved or honored. In the examples under the test, there were three programs whose loops under transformation contained only one statement. These programs were fir_32_1, latnrm_8_1_t and lmsfir_8_1_a. Irrespective of the transformation chosen, the speedup parameter was barely over the zero value for these three programs.

According to the experiments, it is possible to establish some clues on choosing the most efficient transformation matching the specific problem. The slicing transformation performed the best results on the program fir_256_64. Its structure is a loop inside a loop with a statement. Such a structure is called imperfectly nested loop. It is worth noting that statements inside the innermost loop contain all data dependencies types.

The tiling method gave the best results also for a program whose structure is a not perfectly nested loop. The _fir_256_64 program is nearly the same as the fir_256_64 program. The only difference is in statements inside the innermost loop. These statements operate on array data type, which can be easily separated into rectangular code tiles.

The tiling inside the slicing method gave the best results for the program which consisted of several nested loops. One of the loops was perfectly nested, which means that there was no statement inside the parent loop. The _mult_10_10 program consists of significantly the highest number of statements inside the loop under transformation. According to the experimental studies, the tiling inside the slicing method should be chosen when the number of statements in the loop under transformation is significantly high or statements are inside a multiple times nested loop.

The future research will provide the automatic method of choosing transformation parameters, depending on a given program. Moreover, the OpenMP schedule methods will be taken into consideration.

8. References

[1] Ruud Van Der Pas Barbara Chapman, Gabriele Jost. Using OpenMP . The MIT Press, 2007.
[2] Uday Bondhugula, Muthu Baskaran. Affine transformations for communication minimal parallelization and locality optimization of arbitrary nested loop sequences. OSU-CISRC-5/07-TR43.
[3] Intel white pages. http://www.intel.com. Intel website.
[4] Jingling Xue, Loop tiling for parallelism. Kluwer Academic Publishers.
[5] Mostafa Abd-El-Barr Hesham El-Rewini. Advanced computer architecture and parallel processing. Wiley Interscience, 2005.
[6] Addison Wesley. An introduction to parallel computing. Addison Wesley, 2003.

otrzymano / received: 06.12.2011
przyjęto do druku / accepted: 03.01.2012 *artykuł recenzowany / revised paper*