An approach to form affine time partitioning for statement instances of arbitrarily nested loops

Abstract

A novel approach to form affine time partitioning for statement instances of arbitrary nested loops is presented. It is based on extracting free-scheduling which next is used to form a system of equations to produce legal time partitioning. The approach requires an exact dependence analysis. To carry out experiments, the dependence analysis by Pugh and Wonnacott was chosen. Examples illustrating the approach and the results of experiments are presented.

Keywords: loop parallelization, free schedule, affine time partitioning.

1. Introduction

The Affine Transformation Framework (ATF) unifies a large class of transformations, including loop interchange, reversal, skewing, fusion, fission, reindexing, scaling and statement reordering [7, 8]. In this framework, instances of each loop statement are identified by the loop index values of their surrounding loops, and affine expressions are used to map these loop index values into:

- **Space partitions**, where loop statement instances belonging to the same space partition are mapped to the same processor;
- **Time partitions**, where loop statement instances belonging to the same iteration are mapped to different processors or blocks of memory, respectively.

These two approaches are combined in the framework of the current work.

In this paper, we deal with affine loop nests where lower and upper bounds as well as array subscripts and conditions are affine functions of surrounding loop indices and possibly of structure parameters, and the loop steps are known constants.

A loop is perfectly nested if all its statements are comprised within the innermost nest. Otherwise, the loop is called arbitrarily nested.

Following work [2], we refer to a particular execution of a statement for a certain iteration of the loops, which surround this statement, as an operation.

Two operations \( I \) and \( J \) are dependent if both access the same memory location and if at least one access is a write. We refer to \( I \) and \( J \) as the source and destination of a dependence, respectively, provided that \( I \) accesses the same memory location earlier than \( J \).

To describe the algorithm and carry out experiments, we chose the dependence analysis proposed by Pugh and Wonnacott [3] where dependences are represented with dependence relations of the form

\[
\{(\text{input list})\rightarrow(\text{output list}) : \text{constraints}\},
\]

where \( \text{input list} \) and \( \text{output list} \) are the lists of variables and/or expressions used to describe input and output tuples, and \( \text{constraints} \) is a Presburger formula describing the constraints imposed upon \( \text{input list} \) and \( \text{output list} \).

A dependence relation is a mapping from one iteration space to another, and is represented by a set of linear constraints on variables that stand for the values of the loop indices at the source and destination of a dependence, and the values of the symbolic constants.

A one-dimensional affine partition mapping for a statement \( s \) in a loop is an affine expression:

\[
F_s = C_s I_s + c_s,
\]

1. Background

In this paper, we deal with affine loop nests where lower and upper bounds as well as array subscripts and conditions are affine functions of surrounding loop indices and possibly of structure parameters, and the loop steps are known constants.

A loop is perfectly nested if all its statements are comprised within the innermost nest. Otherwise, the loop is called arbitrarily nested.

Following work [2], we refer to a particular execution of a statement for a certain iteration of the loops, which surround this statement, as an operation.

Two operations \( I \) and \( J \) are dependent if both access the same memory location and if at least one access is a write. We refer to \( I \) and \( J \) as the source and destination of a dependence, respectively, provided that \( I \) accesses the same memory location earlier than \( J \).

To describe the algorithm and carry out experiments, we chose the dependence analysis proposed by Pugh and Wonnacott [3] where dependences are represented with dependence relations of the form

\[
\{(\text{input list})\rightarrow(\text{output list}) : \text{constraints}\},
\]

where \( \text{input list} \) and \( \text{output list} \) are the lists of variables and/or expressions used to describe input and output tuples, and \( \text{constraints} \) is a Presburger formula describing the constraints imposed upon \( \text{input list} \) and \( \text{output list} \).

A dependence relation is a mapping from one iteration space to another, and is represented by a set of linear constraints on variables that stand for the values of the loop indices at the source and destination of a dependence, and the values of the symbolic constants.

A one-dimensional affine partition mapping for a statement \( s \) in a loop is an affine expression:

\[
F_s = C_s I_s + c_s,
\]
where $C_i$ is an integer $1 \times n_i$ matrix, and $c_i$ is an integer, $n_i$ is the number of loops surrounding statement $s$. Consider a $p$-statement loop originating dependencies represented with a set of $n$ dependence relations

$$R_i = \{ s1(I_k) \rightarrow s2(I_k) : constraints_k \},$$

where $k = 1, 2, \ldots, n$ and $s1_k, s2_k$ are the identifiers of statements whose instances originate sources and destinations of dependences described with $R_i$, respectively ($s1_k, s2_k \in [1,2,\ldots,n]$). In case of time partitioning for each statement $s$, we have to find an affine mapping (2) such that

$$F_{s1k}(I_k) \leq F_{s2k}(I_k),$$

guaranteeing that in the transformed loop, the destination of a dependence will be executed no earlier than the corresponding dependence source. We can rewrite condition (4) as follows

$$C_{s1k} \times J_k + c_{s1k} - C_{s2k} \times I_k - c_{s2k} \geq 0,$$

where $k = 1, 2, \ldots, n, C_{s1k}$ and $C_{s2k}$ are $1 \times n_i, 1 \times n_2$ matrices, respectively, while $c_{s1k}$ and $c_{s2k}$ are integers representing constant terms of affine mappings.

Time partitioning is known also as scheduling [7].

**Definition 1** [4]

A free schedule assigns operations as soon as their operands are available, that is, mapping $\sigma: I \rightarrow Z$ such that

$$\sigma(p) = \begin{cases} 0 & \text{if there is no } p' \in I \text{ s.t. } p' < p \\ 1 + \max(\sigma(p'), p' \in I, p' < p) & \end{cases}.$$  

(6)

The free schedule is the "fastest" schedule possible. Its total execution time is:

$$T_{free} = 1 + \max(q_{free}(p), p \in I)$$

(7)

Below, we present the main idea of the algorithm, extracting the free schedule for a given loop, details of this algorithm are presented in [1]. Following that algorithm, for each statement all operations are divided into two sets containing independent and dependent operations (sources and destinations of dependences), respectively. Using the second set, those operations are found for which all operands are available. They form the operations of the first set can be combined with the operations of arbitrary layers. Operations in each time partition can be executed in parallel.

In arbitrarily nested loops, statements may have different domains. In general, to find free schedule, we should deal with each statement independently as presented in paper [1].

Using sets $Lay_1$, $Lay_2$, $Lay_3$, we can produce parallel code using any well-known technique, for example, the function codegen in the Omega library [5].

Let us illustrate the algorithm by means of the following example.

**Example 1:**

For $i=1$ to 6 by 1 do
For $j=1$ to 10 by 1 do
$a(1+i, 3*i+1+j+3) = a(1+i+j+1, 1+2*j+4)$

Using Petit [5], we extract the following dependence relations:

$$R_1 = \{ [i,2i]->[i,2i+1] : 1 <= i <= 4 \},$$

$$R_2 = \{ [i,j]->[i',i+i'+1] : j=2i' \land 1<=i<=i'<=5 \},$$

$$R_3 = \{ [i,j]->[j-i,2i] : 2i+2<=j<=i+7,10 \land 1<=i \}.$$

Following the algorithm presented in [1] to extract the free schedule for this example, we get three layers $Lay_1$, $Lay_2$, $Lay_3$, where operations in each layer can be executed in parallel:

$$Lay_1 = \{ \{1,2\} \cup \{1,9\} \cup \{1,1\} \cup \{i,j\} : 2<=j<=i,1<=i \leq i' \leq 5 \},$$

$$Lay_2 = \{ \{i,j\} : 2i+2<=j<=i+7,10 \land 1<=i \}.$$

There must be explicit synchronization between the execution of layers (time partitions). Figure 1 presents the iteration space with dependences and the layers for the examined example.

There are also independent operations contained in set IND:

$$IND = \{ \{1,3\} \cup \{1,9\} \cup \{i,j\} : 2i+4 <= i <= 6 \land 1 <= j <= 2i-1 \} \cup \{(5,10)\} \cup \{1,9\},$$

that can be combined with operations of arbitrary layers.

**3. Algorithm to form time partitions**

The idea of the algorithm presented in this section is as follows. We extract firstly sets $Lay_1, i=1,2,\ldots,p$, following the algorithm presented in [1]. Then, we generate equations to find unknowns representing affine time partitioning. From all solutions to those equations, we choose minimal non-zero solution such that it preserves all loop dependences, i.e., this solution forms legal affine time mapping. If there is no such a solution, we form sets $Lay_2, i=1,2,\ldots,p$, and repeat the above process until yielding an appropriative affine time mapping.

**Algorithm 1. Finding time partitioning for loop statement instances**

**Input:** 1) Set $S$ containing relations $R_i, i=1,2,\ldots,n$ where $n$ is the number of relations, describing all the dependences in a loop; 2) the number of loop statements, $p$; 3) the maximal number of the algorithm iterations, $N$.

**Output:** Matrices $C_i$ of size $1 \times n_i$ and integers $c_i$ presenting affine time partitioning, $i=1,2,\ldots,p$; $j=1,2,\ldots,n_i$, where $n_i$ is the number of loops surrounding statement $i$.
Method:

1. \( k = 1 \)
2. Extract sets \( \text{Lay}_k \), \( i = 1, 2, \ldots, p \), using set \( S \) and the algorithm presented in [1].
3. \( i = 1 \)
4. While \( \text{Lay}_k \) is empty AND \( i \neq p \) do \( i = i + 1 \);
   
   \( e' = i \); form the following system of equations

   \[
   C_{s1k} X_e + C_{s2k} Y_e \quad \text{for} \quad k = 1, 2, \ldots, n
   \]

   \[
   \sum_{s=1}^{n} C_{s1k} X_e + \sum_{s=1}^{n} C_{s2k} Y_e \quad \text{s.t.} \quad X_e, Y_e \in \text{Lay}_{ki} \quad \& \quad X_e < Y_e
   \]  

   \( \forall k \neq p \) note that the system above is to find some elements \( X_e, Y_e \) belonging to \( \text{Lay}_k \), but not all the elements belonging to the same time partition under the free schedule contained into \( \text{Lay}_k \).
5. \( i = 1 \)
6. If \( k \neq N \), then \( k = k + 1 \) and go to step 2; otherwise, the end, the algorithm fails to extract affine time partitions.

Let us illustrate the presented algorithm by means of the following example.

Example 2:

\[
\begin{align*}
\text{for} & \quad i = 1 \quad \text{to} \quad 10 \quad \text{do} \\
& \quad \text{for} \quad j = 1 \quad \text{to} \quad 10 \quad \text{do} \\
& \quad a(i, j+3) = a(i+1, 2j+1);
\end{align*}
\]

Using Petit [5, 6], we extract the following dependence relation

\[
R_1 = \{ [1] \rightarrow [i + 1, j + 2] : 1 \leq i \leq 9 \quad \& \quad 2 \leq j \leq 6 \}.
\]

Following the algorithm [1], we extract the following set

\[
\text{Lay}_1 = \{ [1, j] : 2 \leq j \leq 6 \} \cup \{ [1, j] : \text{Exists}(\alpha : 2 \alpha = 1 + j \quad \& \quad 2 \leq \alpha = 1 \quad \& \quad 3 \leq j \leq 5) \}
\]

To find affine time partitioning, we form the following equation

\[
C_{11} x_1 + C_{12} x_2 = C_{11} y_1 + C_{12} y_2
\]

that corresponds to the system (8). The constraint exists \( X_e, Y_e \) s.t. \( X_e, Y_e \in \text{Lay}_{ki} \) for this example is as follows

\[
\{ (x_1 = 1 \quad \& \quad x_2 = 2) \quad \| \quad (x_1 = 1 \quad \& \quad x_3 = 3) \quad \| \quad \ldots \} \quad \& \quad \{ (y_1 = 1 \quad \& \quad y_2 = 2) \quad \| \quad (y_1 = 1 \quad \& \quad y_3 = 3) \quad \| \quad \ldots \}.
\]

The constraint \( X_e < Y_e \) is of the form

\[
x_1 < y_1 \quad \& \quad (x_1 = y_2 \quad \& \quad x_2 < y_3).
\]

Putting all together, we form the following solution to the system above:

\[
C_{11} = 0 \quad \& \quad C_{12} = \text{Integers}
\]

We choose the solution \( C_{11} = 1, C_{12} = 0 \) and following step 5 check whether this solution is legal

\[
C_{11} (i + 1) + C_{12} (2j - 2) - C_{11} i - C_{12} j = C_{11} + C_{12} (j - 2) = C_{11} = 1
\]

for all \( 1 \leq i \leq 9 \) and \( 2 \leq j \leq 6 \).

Hence, the chosen solution is a valid affine time partitioning. Let us consider another loop.

Example 3:

\[
\text{for} \quad i = 1 \quad \text{to} \quad 10 \quad \text{do}
\]

\[
\text{for} \quad j = 1 \quad \text{to} \quad 10 \quad \text{do}
\]

\[
a(i, j) = a(i, j + 1);
\]

Petit exposes the following dependence relation for this loop

\[
R_2 = \{ [i, j] -> [i, i - 1] : 2 \leq i \leq 5 \}.
\]

The algorithm presented in [1] produces the following sets.

\[
\text{Lay}_2 = \{ [1, j] : 2 \leq i \leq 5 \}.
\]

Following step 3.1 of the presented algorithm, we get

\[
C_{11} x_1 + C_{12} x_2 \quad \& \quad \{ (x_1 = 2 \quad \| \quad x_1 = 3 \quad \| \quad x_1 = 4 \quad \| \quad x_1 = 5) \quad \& \quad (y_2 = 2 \quad \| \quad y_2 = 3 \quad \| \quad y_2 = 4 \quad \& \quad x_1 < y_1) \}
\]

While step 3.2 yields the following system

\[
C_{11} x_1 + C_{12} x_2 = C_{11} y_1 + C_{12} y_2 \quad \& \quad \{ (x_1 = 2 \quad \| \quad x_1 = 3 \quad \| \quad x_1 = 4 \quad \| \quad x_2 = 2) \quad \| \quad (x_2 = 4 \quad \& \quad x_2 = 3) \quad \| \quad (x_2 = 5 \quad \& \quad x_2 = 4)) \}
\]

Applying Mathematica [9], we get a set of solutions to the system above and next examine each solution to choose a legal solution. Such a solution is of the form

\[
(\{x_1 = 1 \quad \& \quad x_2 = 2\} \quad \| \quad \{x_1 = 1 \quad \& \quad x_3 = 3\} \quad \| \quad \ldots \} \quad \& \quad \{y_1 = 1 \quad \& \quad y_2 = 2\} \quad \| \quad \{y_1 = 1 \quad \& \quad y_3 = 3\} \quad \| \quad \ldots \}.
\]

In Table 1, we summarize the results for the examined examples. The second column contains the number of layers needed to find a legal solution. The last column presents affine time partitioning.
The results of experiments demonstrate that applying affine time partitioning allows for using multiple GPUs to reduce the execution time of parallel programs.

5. Related work

Well-known techniques to form affine time partitioning are based on forming system of equations (4) – see background – and then such a system should be resolved applying standard linear algebra techniques (Gaussian Elimination, Fourier-Motzkin Elimination) as well as the Affine Form of Farkas Lemma [2, 4, 8]. Affine mappings are represented by all linearly independent solutions \([C_1, c_1, \ldots, C_n, c_n]\) and \([C_2, c_2, \ldots, C_n, c_n]\) to system (5) and valid for any \(I, J\) satisfying constraints \(k, k = 1, 2, \ldots, n\). These linearly independent solutions can be found by means of techniques presented in [10, 11].

The main drawback of well-known techniques is the considerable computing complexity that prevents their usage in optimizing compilers.

The technique, presented in this paper, is based on forming a limited number of linear equations that can be resolved by means of standard algorithms whose complexity is much less than that of classic techniques permitting for extracting affine time partitioning.

6. Conclusion and future work

In this paper, we presented a technique permitting for building affine time mapping. In comparison with well known-techniques, it has reduced computing complexity and can be applied to parallelize arbitrarily nested loops. In our future work, we plan to implement this approach and verify it on real-life codes to be parallelized and run in GPUs.

The work was supported by finances of West Pomeranian Provincial Administration.

7. References