More Definite Results from the PLuTo Scheduling Algorithm

Athanasios Konstantinidis
Imperial College London
ak807@doc.ic.ac.uk

Paul H. J. Kelly
Imperial College London
p.kelly@doc.ic.ac.uk

ABSTRACT

The *PLuTo* scheduling algorithm is a well-known algorithm for automatic scheduling in the polyhedral compilation model. It seeks linearly independent affine partition mappings for each statement of a Static Control Program (SCoP), such that total communication is minimized. In this paper we show that this algorithm can be sensitive to the layout of the global constraint matrix thus resulting to varying performances of our target code simply by changing the order of the constraint coefficients. We propose a simple technique that automatically determines the right layout for the constraints and as a result immunizes the *PLuTo* scheduling algorithm from constraint ordering variations.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Metrics—*complexity measures, performance measures*

General Terms

Algorithms, Design, Performance

Keywords

Polyhedral Model, Scheduling, Automatic parallelization

1. INTRODUCTION

In the early nineties the polyhedral computation model began to increasingly gain reputation as an efficient way for automatic parallelization and loop-nest transformation of Static Control Programs (SCoP). New techniques for exact array data-flow analysis [8], polyhedra scanning [2] and automatic scheduling of Z-Polyhedra [9][10] paved the way for practical polyhedral compilation frameworks [14][12][16].

Early approaches relied on a space-time view of a schedule where two different algorithms were used for calculating a space mapping (machine independent parallelization for time minimization) and time mapping (machine dependent placement for communication minimization) respectively [11][5][6]. Griebl [13] refined that model by introducing an index-set splitting technique and a placement algorithm for distributed memory machines.

Lim et al. [15][1] later presented a holistic perspective of the problem with a single scheduling algorithm that could find linearly independent partition mappings that maximize the degree of parallelism while minimizing synchronization at the same time. This idea was extended by Uday et al. [4] [3] with a scheduling algorithm that minimizes communication for locality optimization. This approach will be detailed in paragraph 2.1 since it is the main focus of this paper and will be refered to as the Pluto scheduling algorithm or simply Pluto. It was shown [4] [3] that the Pluto algorithm is able to find qualitatively better solutions than both Griebl's [13] and Lim et al. [15] algorithms.

Pluto, as well as other widely used polyhedral compilation frameworks [14][16], rely on a Parametric Integer Programming library (PIP solver) [7] for solving a global constraint matrix. The solution is a 1-dimensional scheduling function (affine partition mapping) obtained by finding the lexicographic minimum values for the constraint coefficients. In this paper we show that the resulting partition mappings from the Pluto scheduler can sometimes be vulnerable to variations in the ordering of the variables of the global constraint matrix. In other words, by slightly altering the order of the constraint coefficients we can get a different answer from the PIP solver.

This weakness is also pointed out by Feautrier [9] who proposed a minimum latency method (already used in Pluto) and a method based on the affine form of the duality theorem [17] for more definite solutions. We propose a simpler solution to this problem by taking into account the direction of each dependence vector. Using this information we can find the right order for the constraint coefficients in the global constraint matrix and guarantee qualitatively better solutions. The key insight behind the proposed solution is that some dependences extend along multiple dimensions while the scheduling algorithm is only aware of their cost in each one separately.

2. BACKGROUND

The polyhedral model is an alternative to abstract syntax trees (AST) that makes it possible for a compiler to find optimal compositions of loop transformations in one step without being subject to phase-ordering limitations [12]. It is consisted of three main data structures namely the domain, the schedule and the dependences for each statement.

The domain of each statement S is a 2-dimensional integer matrix D_S that stores a finite number of affine inequalities (half-spaces/loop bound constraints) representing a convex Z-Polyhedron. For example, if S is surrounded by m loops then it is represented by an m-dimensional polytope as shown bellow, with $\vec{x_S}$ being its iteration vector and $\vec{n}$ a vector of structure parameters:

$$D_S \cdot \begin{pmatrix} \vec{x_S} \\ \vec{n} \\ 1 \end{pmatrix} \geq 0 \tag{1}$$

Using such a domain we can effectively represent all runtime instances of a statement, something not possible with an AST. This representation enables us to find schedules (i.e. affine transformations) representing compositions of loop transformations by effectively imposing a new lexicographic ordering to the domain that can optimize various properties at the same time.

Consequently, the objective of a fully automatic polyhedral compiler framework [4] [14] (as opposed to a semi-automatic one [12]) is to derive optimal schedules for each statement – representing compositions of loop transformations – by solving an affine constraint system built from the dependences E of the source program. These dependences are exact in the sense that they represent directed edges from one run-time instance to another effectively forming a directed dependence graph. Each polyhedral dependence $e \in E$ consists of a dependence polyhedron P_e (7) with two sections : the source and destination domains (could be a subset of their actual execution domains) and an affine transformation (h_transformation) from the destination instances to their source instances. The reader can refer to [8] [18] for a more detailed definition of dependence analysis in the polyhedral model.

2.1 PLuTo Scheduling

A 1-d affine transform of a statement S denoted by $\Phi(\vec{x_S}) = h \cdot \vec{x_S}$ – where $\vec{x_S}$ is the iteration vector of S and h a row vector – maps each run-time instance of S to a new hyperplane instance on the transformed iteration space. Two run-time instances $\vec{x_{S_1}}$ and $\vec{x_{S_2}}$ of S where $\Phi(\vec{x_{S_1}}) = \Phi(\vec{x_{S_2}})$ belong to the same hyperplane instance or in other words to the same loop iteration of the transformed iteration space. Therefore, $\Phi(\vec{x_S})$ effectively represents a new loop in the transformed space that can be either parallel, pipeline parallel (carries dependences with a non-negative distance vector) or sequential.

Obviously, in order to obtain such transforms we need to make sure that they do not violate any of the dependences E of the original program. In other words we need to make sure that for each dependence $e \in E$ the destination runtime instance $\vec{x_{dest}} \in P_e$ is mapped on the same or a greater hyperplane instance than the source $\vec{x_{src}} \in P_e$. In the Pluto context these are called permutability constraints or legality-of-tiling-constraints and are formulated as follows:

$$\Phi(\vec{x_{dest}}) - \Phi(\vec{x_{src}}) \geq 0, \qquad \forall \vec{x_{dest}}, \vec{s_{src}} \in P_e, \quad \forall e \in E \tag{2}$$

The Pluto algorithm then constructs a new set of constraints by introducing a cost function $\delta_e(\vec{n})$ to (2). This is defined as an affine form on the structure parameters and is used to bound the communication distance for each dependence edge :

$$\delta_e(\vec{n}) \geq \Phi(\vec{x_{dest}}) - \Phi(\vec{x_{src}}), \qquad \forall \vec{x_{dest}}, \vec{s_{src}} \in P_e, \quad \forall e \in E \tag{3}$$

Pluto also adds an extra set of constraints to avoid the trivial solution of all transform coefficients being zero. These are called non-trivial solution constraints and simply enforce $\Phi(\vec{x_S}) \geq 1$ for each statement S.

The Pluto algorithm iteratively finds linearly independent and fully permutable affine transforms and stops when $max(m_{S_i})$ – where m_{S_i} the dimensionality of statement S_i – solutions have been found and all the dependences are killed. A dependence is killed by a 1-d affine transform if the following condition holds:

$$\Phi(\vec{x_{dest}}) - \Phi(\vec{x_{src}}) > 0 \tag{4}$$

If the algorithm fails to find more solutions, it removes the dependences killed so far and tries again. If it fails to find any solution, it cuts the dependence graph into strongly-connected components by adding a scalar dimension to the schedule of each statement and then removes any killed dependences and tries again. After obtaining at least one solution the algorithm appends orthogonality constraints to ensure linear independence of the new solutions with the previously found ones.

Like all fully automatic polyhedral frameworks, Pluto relies on the affine form of Farkas lemma [17] and Fourier-Motzkin elimination in order to eliminate unnecessary columns and end up with a system of constraints solely on the unknown schedule coefficients. We can then invoke a PIP solver to obtain a solution (if any) by finding the lexicographic minimum for each schedule coefficient. In order to minimize communication the cost coefficients (3) are being put in the leading minimization positions ensuring that the cost is minimum for each solution.

In the next chapter we show that sometimes the results we get from the PIP solver are actually sensitive to the minimization order or in other words to the order in which we layout our constraint columns (schedule coefficients).

3. ORDERING SENSITIVITY AND PROPOSED SOLUTION

3.1 Motivating Example

The problem manifests itself when there is a situation in which we have the same communication cost δ_e (see (3)) for more than one solution so the minimization algorithm will pick a solution according to the ordering of the schedule coefficients. The example of Figure 1 shows two schedules for statement S_0 that even though both have the same degree of parallelism the second one has a fully parallel loop as opposed to two pipeline parallel ones (wavefront parallelism) of the first schedule.

First of all, by laying out the constraints from both dependences we realize that at the beginning there is no possible solution that has zero communication i.e. there is no fully parallel loop. Therefore, at the first iteration of the algorithm the minimum communication cost is 1 and can be obtained by two solutions, namely $\Phi(\vec{x_{S_0}}) = i$ and $\Phi(\vec{x_{S_0}}) = j$ i.e. minimum coefficients for i and j for communication cost 1. By putting the coefficient of i (a_i) before that of j (a_j) in the global constraint matrix the PIP solver will minimize a_i first, giving as the answer $a_i = 0, a_j = 1$ or $\Phi(\vec{x_{S_0}}) = j$ as the first solution. By adding the orthogonality constraints we then get the $\Phi(\vec{x_{S_0}}) = i$ as our second linearly independent solution. If we now reverse the order of a_i and a_j we will get $\Phi(\vec{x_{S_0}}) = i$ and $\Phi(\vec{x_{S_0}}) = j$. From Figure 1 we see that the order in which we get these two solutions matters since Figure 1-(b) presents a sequential loop followed by a parallel one while Figure 1-(a) presents a pipeline/wavefront parallel loop nest.

Of course, one cannot be certain about which one of the two possible solutions will turn out to be better in practice. It is very likely that fully parallel loops will perform better in most cases, since pipeline/wavefront parallelism comes with a startup and drain cost. However, depending on the problem sizes, a pipeline might end up being equally good or even better if it has better temporal or spatial locality along its wavefront. In the next paragraph we show a simple method to get the right order for the constraint coefficients that takes pipeline cost and temporal locality into account.

3.2 Proposed Solution

The reason why the scheduling algorithm is unable to distinguish between these two solutions is because both dependences in Figure 1 have the same communication cost along each dimension i and j. The difference between them lies on their direction i.e. one of the dependences extends into both i and j dimensions as opposed to the other one that extends along i only (carried by only one of the two loops). By extracting and using this information we could be able to determine the right order for the constraint coefficients and distinguish between pipeline and fully parallel degrees of parallelism.

Let S_{dest} be the destination statement of a dependence edge $e \in E$. We define a bit vector $\vec{V_e}$ with size $min(m_{dest}, m_{src})$ (dimensionalities of S_{dest} and S_{src}) that would store the direction information for e. We also store a boolean attribute H_e which is false if a dependence vector extends along more than one domension. In particular :

$$\vec{V_e}[i] = \begin{cases} 1 & \text{if } e \text{ extends along } i, \\ 0 & \text{if } e \text{ doesn't extend along } i \end{cases}, \qquad (5)$$
$$0 \leq i < min(m_{dest}, m_{src})$$

$$H_e = \begin{cases} true & \text{if } e \text{ is horizontal}, \\ false & \text{if } e \text{ is diagonal} \end{cases} \qquad (6)$$

Each dependence edge $e \in E$ is represented by a dependence polyhedron P_e defined as follows :

$$P_e = \begin{array}{c c} & \begin{array}{ccc} m_{dest} & m_{src} & (n+1) \end{array} \\ \begin{array}{c} L \\ \\ m_{src} \end{array} & \left[\begin{array}{cc|c} D_{dest} & \varnothing & \\ \hdashline \varnothing & D_{src} & \\ \hline \multicolumn{3}{c}{h\_transformation} \end{array}\right] \end{array} \cdot \begin{bmatrix} \vec{x_{S_{dest}}} \\ \vec{x_{S_{src}}} \\ \vec{n} \\ 1 \end{bmatrix} \begin{array}{c} \geq 0 \\ \hline = 0 \end{array} \qquad (7)$$

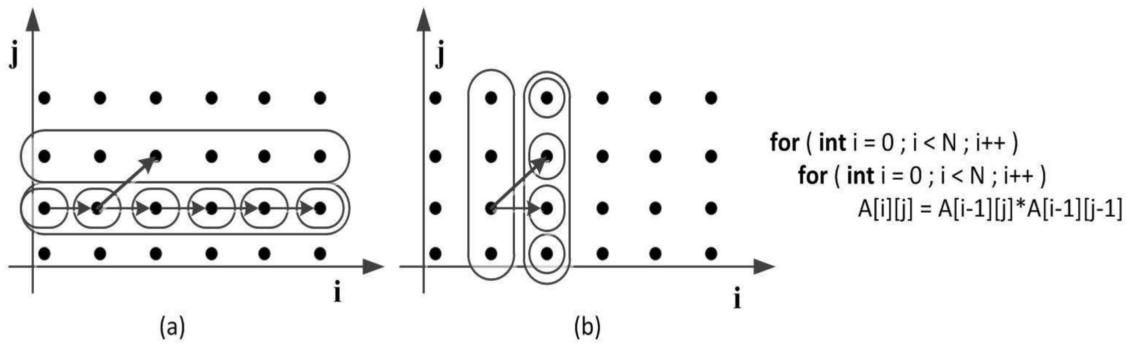
By taking (7) into account we can use Algorithm 1 to populate the direction vectors for each $e \in E$.

Algorithm 1 Direction extraction

```
1: for all dependences $e \in E$ do
2:   $V_e$ initialized to 0
3:   bool $H_e = true$
4:   int $count = 0$;
5:   for $i = 0$ to $min(m_{dest}, m_{src})$ do
6:     if $P_e[L+i][m_{dest}+i] + P_e[L+i][i] = 0$ then
7:       if $\exists j \neq i, (m_{dest}+i)$ s.t. $P_e[L+i][j] \neq 0$ then
8:         $V_e[i] = 1$;
9:         count++;
10:      end if
11:    else
12:      count++;
13:    end if
14:  end for
15:  if $count > 1$ then
16:    $H_e = false$
17:  end if
18: end for
```

Upon construction of the global constraint matrix we can determine the order of the transform coefficients for each statement using Algorithm 2.

In our example we have two dependence edges e_1 and e_2 where $V_{e_1} = [1, 1]$ and $V_{e_2} = [1, 0]$. Furthermore, the first edge e_1 is diagonal so $H_{e_1} = false$ and $H_{e_2} = true$. There-



```
for ( int i = 0 ; i < N ; i++ )
   for ( int i = 0 ; i < N ; i++ )
         A[i][j] = A[i-1][j]*A[i-1][j-1]
```

Figure 1: Schedule (a) is $\Phi(x_{S_0}) = (j, i)$ and results in a pipeline parallel loop nest while schedule (b) is $\Phi(x_{S_0}) = (i, j)$ and results in one fully parallel loop.

Algorithm 2 Coefficient ordering algorithm

```
 1: Let $N$ be the total number of statements in the source
    program
 2: for all Statements $S_i$, $0 \leq i < N$ do
 3:   Let $V_{S_i}$ a bit vector with size $m_{S_i}$ initialized to $\vec{0}$
 4:   for all $e \in E$ s.t. $S_{dest} = S_i$ do
 5:     if $H_e = true$ then
 6:       $V_{S_i} = V_{S_i}$ OR $V_e$
 7:     end if
 8:   end for
 9:   for each element $j$ of $V_{S_i}$ do
10:     if $V_{S_i}[j] = 0$ then
11:       Put coefficient $a_{S_j}$ in leading minimization posi-
          tion
12:     end if
13:   end for
14: end for
```

fore, Algorithm 2 will give us $V_{S_0} = [1, 0]$ and as a result we will put a_{S_j} in the leading minimization position.

By applying this technique we can choose fully parallel degrees of parallelism instead of pipeline ones. However, as we already mentioned this might not be the best strategy depending on problem sizes and locality along a wavefront. A wavefront for statement S on an m-dimensional loop nest can be represented by the following hyperplane :

$$\Phi_{wave_S}(\vec{x_S}) = \overbrace{\begin{bmatrix}1 & 1 & \dots & 1\end{bmatrix}}^{m} \cdot \vec{x_S} \qquad (8)$$

We can measure the volume of temporal locality within a wavefront by counting the Read-after-Read (input) dependences that satisfy the following condition :

$$\Phi_{wave_{S_{dest}}}(\vec{xs_{dest}}) = \Phi_{wave_{S_{src}}}(\vec{xs_{src}}) \qquad (9)$$

We can then define empirical thresholds for the structure parameters and the temporal reuse along a wavefront to decide whether pipeline parallelism would be better for a particular hardware architecture or not. Deriving these empirical thresholds for different architectures requires experimental investigation that could be subject for future research.

4. CONCLUSIONS

In this paper we showed that a widely used polyhedral scheduling algorithm for automatic parallelization [4] [3] can sometimes be sensitive to the layout of the global constraint matrix that we use to obtain our solutions. To overcome this ambiguity we propose an empirical methodology based on the direction of each dependence vector that tries to find the right order for the unknown transformation coefficients. The right order assumes that a fully parallel degree of parallelism is usually better than a pipeline/wavefront one. However, we showed that the volume of temporal reuse along a wavefront can be calculated enabling us to derive empirical machine-dependent thresholds to make a more precise decision.

5. ACKNOWLEDGEMENTS

The authors would like to thank EPSRC and Codeplay for funding this research. We would also like to thank Alastair Donaldson, Uday Bondhugula and Armin Größlinger for their valuable comments and advice.

6. REFERENCES

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[2] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 7–16, 2004.

[3] U. Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *Compiler Construction*, volume 4959 of *Lecture Notes in*

Computer Science, pages 132–146. Springer Berlin / Heidelberg, 2008.

[4] U. Bondhugula and J. Ramanujam. Pluto: A practical and fully automatic polyhedral parallelizer and locality optimizer. Technical report, 2007.

[5] A. Darte and F. Vivien. Automatic parallelization based on multi-dimensional scheduling, 1994.

[6] M. Dion and Y. Robert. Mapping affine loop nests. *Parallel Computing*, 22(10):1373 – 1397, 1996.

[7] P. Feautrier. Parametric integer programming. *RAIRO Recherche Op'erationnelle*, 22, 1988.

[8] P. Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20:23–53, 1991.

[9] P. Feautrier. Some efficient solutions to the affine scheduling problem. i. one-dimensional time. *International Journal of Parallel Programming*, 21:313–347, 1992.

[10] P. Feautrier. Some efficient solutions to the affine scheduling problem. part ii. multidimensional time. *International Journal of Parallel Programming*, 21:389–420, 1992.

[11] P. Feautrier. Toward automatic distribution. *Parallel Processing Letters*, 4, 1994.

[12] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *International Journal of Parallel Programming*, 34:261–317, 2006.

[13] M. Griebl. *Automatic Parallelization of Loop Programs for Distributed Memory Architectures*. University of Passau, 2004. habilitation thesis.

[14] C. Lengauer. Loop parallelization in the polytope model. In *CONCUR'93*, volume 715 of *Lecture Notes in Computer Science*, pages 398–416. Springer Berlin / Heidelberg, 1993.

[15] A. W. Lim and M. S. Lam. Maximizing parallelism and minimizing synchronization with affine transforms. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '97, pages 201–214, New York, NY, USA, 1997. ACM.

[16] L.-N. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, and P. Sadayappan. Combined iterative and model-driven optimization in an automatic parallelization framework. In *Conference on Supercomputing (SC'10)*, New Orleans, LA, Nov. 2010.

[17] A. Schrijver. *Theory of linear and integer programming*. John Wiley & Sons, Inc., New York, NY, USA, 1986.

[18] N. Vasilache, C. Bastoul, A. Cohen, and S. Girbal. Violated dependence analysis. In *Proceedings of the 20th annual international conference on Supercomputing*, ICS '06, pages 335–344. ACM, 2006.