Note for *More Legal Transformations for Locality*

Ying Cao

March 3, 2020

Contents

1 What problem is this paper going to solve?

Exploiting data locality is one of the keys to achieve high performance level. The basic framework for increasing the cache hit rates aims at *moving references to a given memory cell (or cache line) to neighboring iterations of some innermost loop.* Such a transformation modifies the operation execution order, the existence of a good solution highly depends on data dependences.

2 Background

1. When loop bounds and conditionals only depend on surrounding loop counters, formal parameters and constants, the iteration domain *can always* be specified by a set of linear inequalities defining a polyhedron.
   (a) The term *polyhedron* will be used in a broad sense to denote a *convext set of points in a lattice* (also called $\mathbb{Z}$-polyhedron or lattice-polyhedron).
2. Each statement may include one or several *references* to arrays.
3. The subscript function $f(\boldsymbol{x})$ maps iteration vectors to array elements. When the subscript function $f(\boldsymbol{x})$ of a reference is affine, we can write it as $f(\boldsymbol{x}) = F\boldsymbol{x} + \boldsymbol{a}$ where F is called the *subscript matrix*.

   For the example below, the reference to the array B is $B\left[f(\boldsymbol{x})\right]$ with $f\begin{pmatrix} i \\ j \end{pmatrix} = \begin{bmatrix} 1 & 1 \\ 2 & 0 \end{bmatrix}\begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \end{pmatrix}$

```
1  for(int i = 1; i < n; ++i){
2    for(int j = 1; j < n; ++j){
3      if(i <= n + 2 - j)
4  S1:   B[i + j][2 * i + 1] = ...
5    }
6  }
```

3 Affine transformations for Locality

3.1 Formulation

The goal of a transformation is to modify the original execution order of the operations. To do this, the optimizing compilers build schedules at **the statement level** by finding a function which specify an execution time for each instance of the corresponding statement.
The schedule functions are usually choosen affine, thus have the following shape:

$$\theta_S(\boldsymbol{x}_S) = T_S\boldsymbol{x}_S + \boldsymbol{t}_s \tag{1}$$

1. $\boldsymbol{x}_S$ is the iteration vector.
2. T_S is a constant transformation matrix.
3. $\boldsymbol{t}_S$ is a constant vector possibly including affine parameteric expressoins using the structure parameters of the program (i.e. the symbolic constants, mostly array sizes or iteration bounds).

Linear transformations can express most of the useful transformations.

3.2 An example

Consider an original polyhedron defined by the system $A\boldsymbol{x} + \boldsymbol{c} \geq \mathbf{0}$, and the transformation function θ leads to the target index $\boldsymbol{y} = T\boldsymbol{x}$, then we can deduce that the transformed polyhedron canbe defined by:

$$\left(AT^{-1}\right)\boldsymbol{y} + \boldsymbol{c} \geq \mathbf{0}$$

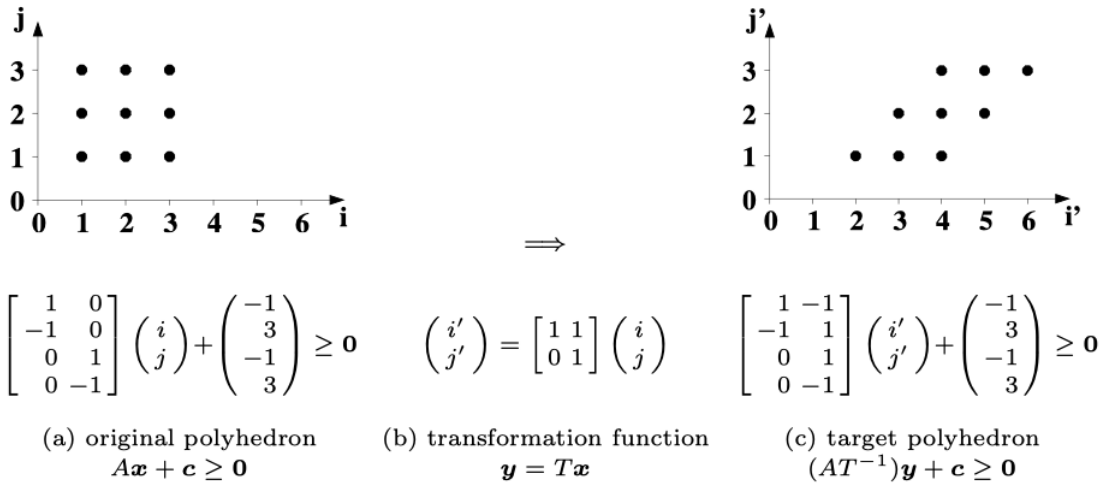

Figure 1: a skewing transformation

3.3 How dependences can be expressed exactly using linear (in)equalities ?

A convenient wany to represent the scheduling constraints between the program operations is the ***Dependence Graph***. In this directed graph:

1. Each program statement is represented using a unique vertex;
2. The existing dependence relations between statement instances are represented using edges.
   - Each vectex is labelled with the iteration domain of the corresponding statement
   - Each edges is labelled with the dependence polyhedra which describes the dependence relation between the source and destination statements.

A statement R ***depends*** on a statement S (written SδR) if there exits an operation $S(\boldsymbol{x}_1)$, an operation $R(\boldsymbol{x}_2)$ and a memory location m such that:

1. $S(\boldsymbol{x}_1)$ and $R(\boldsymbol{x}_2)$ refer the same memory location m, and at least one of them writes to that location.
2. $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ respectively belong to the iteration domain S and R;
3. in the original sequential order, $S(\boldsymbol{x}_1)$ is executed before $R(\boldsymbol{x}_2)$.

From this definition, it is easy to describe the ***dependence polyhedra*** of each dependence relation between two statements with affine (in)equalities. Such a dependence system has following components:

1. *Same memory location*: $F_S\boldsymbol{x}_S + a_S = F_R\boldsymbol{x}_R + a_R$;
2. *Iteration domains*: $A_S\boldsymbol{x}_S + \boldsymbol{c}_S \geq \mathbf{0}$ and $A_R\boldsymbol{x}_R + \boldsymbol{c}_R \geq \mathbf{0}$;
3. *Precedence order*: $P_S\boldsymbol{x}_S - P_R\boldsymbol{x}_R + \boldsymbol{b} \geq \mathbf{0}$;
   (a) this constraint can be separated into a disjunction of as many parts as there are common loops to both S and R.
   (b) each case corresponds to a common loop depth, and is called *dependence level*.

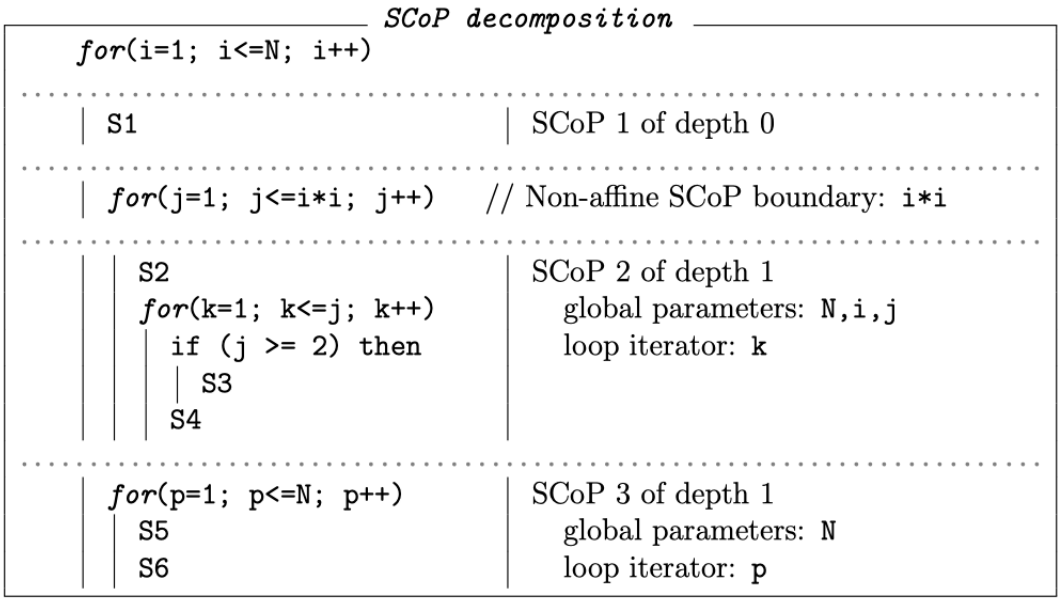

Figure 2: SCoP decomposition

3.4 Legal Transformation Space

Consider the transformations as scheduling functions, the time interval in the target program between the executions of two operations $R(\boldsymbol{x}_R)$ and $S(\boldsymbol{x}_S)$ is:

$$\triangle_{R,S}\begin{pmatrix}\boldsymbol{x}_S\\ \boldsymbol{x}_R\end{pmatrix} = \theta_R(\boldsymbol{x}_R) - \theta_S(\boldsymbol{x}_S). \tag{2}$$

If $\mathscr{D}_{S\delta R}$ is not empty, then:

$$\triangle_{R,S}\begin{pmatrix}\boldsymbol{x}_S\\ \boldsymbol{x}_R\end{pmatrix} - \mathbf{1} \geq \mathbf{0}. \tag{3}$$

The affine scheduling function can be expressed in terms of D and $\boldsymbol{d}$ by applying Farkas Lemma.

> **Lemma 1.** *(Affine form of Farkas Lemma [15]) Let $\mathcal{D}$ be a nonempty polyhedron defined by the inequalities $A\boldsymbol{x} + \boldsymbol{b} \geq \mathbf{0}$. Then any affine function $f(\boldsymbol{x})$ is nonnegative everywhere in $\mathcal{D}$ iff it is a positive affine combination:*
>
> $$f(\boldsymbol{x}) = \boldsymbol{\lambda}_0 + \Lambda^T(A\boldsymbol{x} + \boldsymbol{b}), \text{ with } \boldsymbol{\lambda}_0 \geq \mathbf{0} \text{ and } \Lambda^T \geq 0,$$
>
> *where $\boldsymbol{\lambda}_0$ and Λ^T are called Farkas multipliers.*

Figure 3: Affine form of Farkas Lemma.

According to this Lemma, for each edge in the dependence graph, we can find a positive vector $\boldsymbol{\lambda}_0$ and matrix $\boldsymbol{\Lambda}^T$ such that:

$$T_R\boldsymbol{x}_R + \boldsymbol{t}_R - (T_S\boldsymbol{x}_S + \boldsymbol{t}_S) - \mathbf{1} = \boldsymbol{\lambda}_0 + \boldsymbol{\Lambda}^T\left(D\begin{pmatrix}\boldsymbol{x}_S\\ \boldsymbol{x}_R\end{pmatrix} + \boldsymbol{d}\right), \boldsymbol{\lambda}_0 \geq \mathbf{0}, \boldsymbol{\Lambda} \geq \mathbf{0} \tag{4}$$

3.5 Properties

- There is self-temporal reuse when $x_r \in \ker F$.
- The resue can be exploited if the transformed iteration order follows one of the reuse directions.
- (**??**)*Then we have to find an orthogonal vector to the chosen reuse direction to be the first part of the transformation matrix* T.
- If this partial transformation do not violate dependences, we have many choices for the completion procedure[1] (**??**).
- To generalize, it is easy to consider not only a reuse direction vector, but a reuse direction space. *in order for the transformation function to be instance-wise.*

4 Finding Legal Transformations

TBD

5 Some useful reading

1. Complex transformations such as Tiling canbe achieved using linearning transformations [2].

References

[1] M. Griebl, C. Lengauer, and S. Wetzel, "Code generation in the polytope model," in *Proceedings. 1998 International Conference on Parallel Architectures and Compilation Techniques (Cat. No. 98EX192)*, pp. 106–111, IEEE, 1998.

[2] J. Xue, "On tiling as a loop transformation," *Parallel Processing Letters*, vol. 7, no. 04, pp. 409–424, 1997.

[1] I do not understand what is the completion procedure here? The author refers to [1] which explains completion to a unimodular transformation matrix.