Non-Unimodular Code Generation for Parallel Machines

Jingling Xue
Department of Mathematics, Statistics and Computing Science
University of New England
Armidale, NSW 2351, Australia

Abstract
Unimodular transformations have been used successfully in compiling programs for efficient execution on parallel machines. This paper discusses a number of code generation problems for which non-unimodular transformations are useful.

1 Introduction
Code transformations are important components in parallelising compilers and are used to restructure code to enable the exploitation of parallelism in programs. The importance of unimodular transformations such as loop interchange and loop skewing [2] has been widely recognised. This paper gives a list of code generation problems for which non-unimodular transformations are useful. We show that many code generation problems can be reduced to one of constructing DO loops to scan the intersection of a convex polyhedron and a lattice, which is the image of a convex polyhedron under a non-unimodular loop transformation and is called a polyhedral lattice. A polyhedral lattice is a convex polyhedron if the underlying lattice is \( \mathbb{Z}^n \), i.e., if \( T \) is unimodular.

In [14], we presented a method to construct DO loops to scan a polyhedral lattice. Given a convex polyhedron \( \Phi = \{x \mid Ax \leq b\} \) in \( \mathbb{Z}^n \), which models the iteration space of an \( n \)-deep loop nest, and a non-singular loop transformation \( T \in \mathbb{Z}^{m \times n} \), the problem is to construct DO loops to scan the points in the polyhedral lattice \( \{x \mid AT^{-1}y \leq b\} \cap \mathcal{L}(T) \) in their lexicographical order. Here, \( \mathcal{L}(T) \) denotes the lattice generated by the columns of \( T \). This paper discusses how this method can be used in several code generation problems that arise in compiling for parallel machines. For each code generation problem, we either give \( \Phi \) and \( T \) or the polyhedral lattice \( \{x \mid AT^{-1}y \leq b\} \cap \mathcal{L}(T) \). In many cases, we present the codes that scan the underlying polyhedral lattices.

2 VLSI Processor Array Design
VLSI processor arrays (e.g., systolic arrays and multirate arrays) are special-purpose parallel architectures suitable for many regular iterative algorithms [11].

2.1 Systolic Arrays
The mapping of algorithms to systolic arrays amounts to finding non-singular transformations [9]. For example, the well-known H.T. Kung’s systolic array (Fig. 1) for matrix product is specified by the following non-unimodular matrix:

\[
T = \begin{bmatrix}
1 & 1 & 1 \\
1 & 0 & -1 \\
0 & 1 & -1
\end{bmatrix}
\]

The iteration space is \( \Phi = \{(i, j, k) \mid 0 < i, j, k \leq m\} \), where \( m \) is the matrix size. The DO loops that scan the transformed iteration space provides a complete specification for the systolic array.

Non-unimodularity was also used to advantage in [3] to map algorithms to fixed-size processor arrays.

2.2 Multirate Arrays

![Figure 2: A multirate array where solid dots represent active processors.](image)

Multirate arrays are an extension of systolic arrays where different data variables are propagated with different clocks [7]. One maps algorithms to multirate
arrays by transforming them into so-called sparse uniform recurrence equations each have the form:

\[ \forall p \in J \rightarrow y_k(Mp + a) = f(\cdots, y_j(Mp + a + b), \cdots) \]

where \(J\) is a convex polyhedron and \(M\) is a non-singular matrix that must be synthesised by the design method. Therefore, the domain of such an equation is a polyhedral lattice. For an example taken from [7], the multirate array shown in Fig. 2 is described by the following equation:

\[ y_k \left( \begin{bmatrix} 3 \\ 0 \\ 1 \\ \frac{1}{2} \end{bmatrix} \right) = f(\cdots, y_j \left( \begin{bmatrix} 3 \\ 0 \\ 1 \\ \frac{1}{2} \end{bmatrix} + \begin{bmatrix} 3 \\ 0 \\ 1 \\ 1 \end{bmatrix}, \cdots) \]

In this multirate array, all the processors are simultaneously active every three clock ticks. The variable \(y_k\) moves three times slower than the fastest variable.

3 Improving Data Locality

3.1 Access Normalisation

In [8], a technique was introduced that restructures a loop nest using array indexing functions in such a way that code can be generated with improved data locality and more block transfer fetches.

Consider a simple example:

\[
\begin{align*}
\text{do } x_1 &= 1, N \\
\text{do } x_2 &= 1, N \\
a(2x_1 + 3x_2, x_1 + 3x_2) &= x_1
\end{align*}
\]

Assume that we apply the owner computes rule to determine the computations that a processor is responsible to execute. If we distribute array \(a\) using some standard data distributions such as block or cyclic distribution, some processors will have non-local accesses to \(a\). To eliminate these non-local data accesses completely in the current example, we can choose the indexing function of array \(a\) – a non-unimodular loop transformation

\[
T = \begin{bmatrix} 2 & 3 \\ 1 & 3 \end{bmatrix}, \quad T \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}
\]

- to restructure the original loop nest into the following loop nest:

\[
\begin{align*}
\text{do } y_1 &= 5, 5N \\
\text{do } y_2 &= \max(y_1 - N, \lceil (y_1 + 3)/2 \rceil) + (2y_1 - \\
&\quad \max(y_1 - N, \lceil (y_1 + 3)/2 \rceil)) \text{ mod } 3, \\
&\quad \min(y_1 - 1, \lceil (y_1 + 3N)/2 \rceil), 3 \\
&\quad a(y_1, y_2) = y_1 - y_2
\end{align*}
\]

Now if \(a\) is distributed using block or cyclic distribution as before, there are no non-local accesses to \(a\).

3.2 Loop Tiling

Looping tiling can be used to achieve data locality for machines with multiple levels of the memory hierarchy [6, 12, 13]. The objective is to keep reused data in the cache or register file so as to reduce memory accesses. In general, loop tiling transforms an \(n\)-dimensional loop nest into an \(2n\)-dimensional loop nest. The inner \(n\) loops execute the iterations inside a tile and the outer \(n\) loops enumerate over all the tiles. The shape and size of the tile is chosen so that the data used within the tile can be held within the cache. A tile is an \(n\)-dimensional parallelepiped that can be specified using an integer matrix of size \(n \times n\) whose column vectors are the edge vectors of the parallelepiped [6]. An example given in [10] shows that this matrix must be non-unimodular so as to minimise the amount of data movement. Fig. 3 depicts the tiling of a two-dimensional iteration space using a tile specified by the following non-unimodular matrix:

\[
L = \begin{pmatrix} 2 & 1 \\ 2 & 3 \end{pmatrix}
\]

The iteration space is

\[
P = \{(x_1, x_2) \mid 1 \leq x_1 \leq 6, 1 \leq x_2 \leq 5\}
\]

To enumerate over all the tiles that partition the iteration space, we simply construct a loop nest to scan the polyhedral lattice \(P \cap L(L)\):

\[
\begin{align*}
\text{do } x_1 &= 1, 6 \\
\text{do } x_2 &= 1 + (3x_1 - 1) \text{ mod } 4, 5, 4 \\
&\text{Enumerate the tile with origin } (x_1, x_2)
\end{align*}
\]

The two inner loops for executing the iterations in the tile can be constructed easily and are not given here.

4 Generating Data Transfer Code

For machines with complex memory hierarchies, a substantial performance gain can be achieved if copies of frequently referenced data structures can be stored in local memories [1, 4]. We show that non-unimodular transformations are useful in the generation of data transfer code to move data between the memory levels.

Consider the case of a parallel program for execution in a linear array of \(P\) processors consisting of a local memory private to each processor and a global memory shared by all the processors:

\[
\begin{align*}
doallp \quad p &= 0, P - 1 \\
&\text{do } x_1 = 1, N \\
&\text{do } x_2 = 1, N \\
&\quad \cdots = a(x_1 + p, 2x_2) + a(x_1 + x_2 + p, 2x_2 + 2)
\end{align*}
\]
Figure 4: Part of $a$ accessed by processor $p = 0$ ($N = 3$).

where $p$ is the “processor index”.

Assuming that $a$ is initially stored in global memory, we want to generate the code needed to make part of $a$ accessed by a processor into its local memory.

The two references access the following elements:

\[
\{(y_1, y_2) \mid p + 1 \leq y_1 \leq p + N, 2 \leq y_2 \leq 2N\} \cup \\
\{(y_1, y_2) \mid 2p \leq 2y_1 - y_2 \leq 2p + 2N - 2, 4 \leq y_2 \leq 2N + 2\} \\
\cap L(L)
\]

where

\[
L = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}
\]

To scan this set, which is not a polyhedral lattice (Fig. 4), we apply our method reported in [15] and obtain the following code:

\[
doall p = 0, P - 1 \\
\do y_1 = p + 1, p + N \\
\do y_2 = 2, 2N, 2 \\
\afmp(y_1, y_2) = \text{fetch}(a(y_1, y_2)) \\
\do y_1 = p + N + 1, p + 2N \\
\do y_2 = \max(4, 2y_1 - 2p - 2N + 2), \min(2y_1 - 2p, 2N + 2), 2 \\
\afmp(y_1, y_2) = \text{fetch}(a(y_1, y_2))
\]

5 Constructing SPMD Programs

There are two phases in compiling to machines with a distributed address space [5]. The first phase determines how to decompose the data and computation across the processors. The second phase is to generate the SPMD (Single-Program Multiple Data) program for each processor such that each processor executes its allocated computations and communicates with other processors whenever needed.

5.1 Computation Code

Suppose we want to compile a simple program for a 4-processor distributed memory machine.

\[
\begin{align*}
\text{real } a(40), b(40) \\
\text{decomposition } T(40) \\
\text{align } a, b \text{ with } T \\
\text{distribute } T'(\text{cyclic, block}) \\
\do x_1 = 3, 40 \\
\do x_2 = 1, 40 \\
\quad a(x_1, x_2) = b(x_1 - 2, x_2) + 1
\end{align*}
\]

where arrays $a$ and $b$ are aligned with decomposition $T$, which is then distributed by (cyclic, block).

We can calculate that the following array element of $a$ are stored locally at processor $p$, $0 \leq p \leq 3$:

\[
\{(x_1, x_2) \mid p + 1 \leq x_1 \leq 40, 10p + 1 \leq x_2 \leq 10(p + 1)\} \cap L(p)
\]

where

\[
L(p) = \{ \left[ \begin{array}{cc} 4 & 0 \\ 0 & 1 \end{array} \right] \left[ \begin{array}{c} i \\ j \end{array} \right] : \left[ \begin{array}{c} p + 1 \\ 0 \end{array} \right] \left[ \begin{array}{c} i \\ j \end{array} \right] \in \mathbb{Z}^2 \}
\]

Assuming the owner computes rule, we obtain the computation code.

\[
do x_1 = \max(p + 1, 3) + (p + 1 - \max(p + 1, 3)) \mod 40, 4 \\
\do x_2 = 10p + 1, 10(p + 1) \\
\quad a(x_1, x_2) = b(x_1 - 2, x_2) + 1
\]

5.2 Communication Code

When a processor accesses an array element in another processor in a distributed memory machine, we must generate sends and receives in such a way that the owner executes send and the recipient executes receive.

For illustration purposes, we modify the example in the previous section to get:

\[
\begin{align*}
\text{real } a(40), b(40) \\
\text{decomposition } T(40) \\
\text{align } a, b \text{ with } T \\
\text{distribute } T'(\text{block, block}) \\
\do x_1 = 1, 20 \\
\do x_2 = 1, 20 \\
\quad a(x_1, x_2) = b(2x_1, x_1 + x_2)
\end{align*}
\]

This time, arrays $a$ and $b$ are aligned with decomposition $T$, which is then distributed by (block, block).

For reasons of symmetry, we only discuss how to generate the communication code for sending the elements of $b$ owned by processor $p$ to processor $q$ where those elements are accessed. The set of the elements of $b$ owned by processor $p$, $0 \leq p \leq 3$, is

\[
\{(x_1, x_2) \mid 10p + 1 \leq x_1, x_2 \leq 10(p + 1)\}
\]

which also represents the set of computations allocated to processor $p$ according to the owner computes rule.

The indexing function of array $b$ defines a non-unimodular transformation:

\[
T = \begin{bmatrix} 2 & 0 \\ 1 & 1 \end{bmatrix}, \quad T \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}
\]

The set of the elements of $b$ that are accessed at processor $q$, $0 \leq q \leq 3$, is the image of (1) under the mapping $T$:

\[
\{(y_1, y_2) \mid 10q + 1 \leq y_1, y_2 \leq 10(q + 1)\} \cap L(T)
\]

Let $p \neq q$. The elements of array $b$ to be sent from processor $p$ to processor $q$ is the intersection of (1) and (2). As a polyhedral lattice, it can be scanned by the following loop nest:
do $y_1 = \max(10p+2, 20q+2), \min(10p+10, 20q+20), 2

\begin{align*}
\text{do } y_2 &= \max(10p + 1, [10q + 1 + y_1 / 2]), \\
&\quad \min(10p + 10, [10q + 10 + y_1 / 2])
\end{align*}

send $b(y_1, y_2)$ from $p$ to $q$

6 Conclusion
In this paper, we showed that several code generation problems in compiling for parallel machines can be reduced to one of constructing DO loops to scan a polyhedral lattice, which is the image of a convex polyhedron under a non-unimodular loop transformation. We discussed a number of code generation problems for which non-unimodular transformations must be used.

References


